Oracle® Database JDBC Developer's Guide



ORACLE

Oracle Database JDBC Developer's Guide, 23c

F47013-09

Copyright © 1999, 2024, Oracle and/or its affiliates.

Primary Author: Tulika Das

Contributing Authors: Brian Martin, Venkatasubramaniam Iyer, Elizabeth Hanes Perry, Brian Wright, Thomas Pfaeffle

Contributors: Kuassi Mensah, Douglas Surber, Paul Lo, Ed Shirk, Tong Zhou, Jean de Lavarene, Rajkumar Irudayaraj, Ashok Shivarudraiah, Nirmala Sundarappa, Angela Barone, Rosie Chen, Sunil Kunisetty, Joyce Yang, Mehul Bastawala, Luxi Chidambaran, Vidya Nayak, Srinath Krishnaswamy, Swati Rao, Pankaj Chand, Aman Manglik, Longxing Deng, Magdi Morsi, Ron Peterson, Ekkehard Rohwedder, Catherine Wong, Scott Urman, Jerry Schwarz, Steve Ding, Soulaiman Htite, Anthony Lai, Prabha Krishna, Ellen Siegal, Susan Kraft, Sheryl Maring

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle®, Java, MySQL and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Contents

Preface

Audience	XXV
Documentation Accessibility	XXV
Related Documents	XXV
Conventions	xxvi

Changes in This Release for Oracle Database JDBC Developer's Guide

New Features	XXX
--------------	-----

Part I Overview

1 Introducing JDBC

1.1 O	verview of Oracle JDBC Drivers	1-1
1.2 Cl	noosing the Appropriate Driver	1-3
1.3 Us	se Cases of Oracle JDBC Drivers	1-4
1.4 Fe	eature Differences Between JDBC OCI and Thin Drivers	1-5
1.5 Er	nvironments and Support	1-6
1.5.1	1 Supported JDK and JDBC Versions	1-6
1.5.2	2 JNI and Java Environments	1-7
1.5.3	3 JDBC and IDEs	1-7
1.5.4	4 Availability on Maven Central	1-7
1.6 Fe	eature List	1-9

2 Getting Started

2.1	RDB	MS and JDK Version Compatibility for Oracle JDBC Drivers	2-1
2.2	2.2 Verifying a JDBC Client Installation		
2.2.1 Checking the Environment Variables		Checking the Environment Variables	2-3
2	2.2.2	Ensuring that the Java Code Can Be Compiled and Run	2-4
2	2.2.3	Determining the Version of the JDBC Driver	2-4



	2.2.4	Testing the JDBC and Database Connection	2-5
2	.3 Basi	c Steps in JDBC	2-7
	2.3.1	Importing Packages	2-7
	2.3.2	Opening a Connection to a Database	2-8
	2.3.3	Creating a Statement Object	2-9
	2.3.4	Running a Query and Retrieving a Result Set Object	2-9
	2.3.5	Processing the Result Set Object	2-10
	2.3.6	Closing the Result Set and Statement Objects	2-10
	2.3.7	Making Changes to the Database	2-11
	2.3.8	About Committing Changes	2-13
	2.3	3.8.1 Changing Commit Behavior	2-14
	2.3.9	Closing the Connection	2-15
2	.4 Sam	ple: Connecting, Querying, and Processing the Results	2-15
2	.5 Supp	port for Annotations	2-16
2	.6 Supp	port for Oracle True Cache	2-17
2	.7 Supp	port for the Bequeath Protocol	2-18
2	.8 Supp	port for Invisible Columns	2-19
2	.9 Supp	oort for Verifying JSON Data	2-20
2	.10 Sup	oport for Implicit Results	2-21
2	.11 Sup	port for Lightweight Connection Validation	2-24
2	.12 Sup	oport for Deprioritization of Database Nodes	2-25
2	.13 Sup	port for Oracle Connection Manager in Traffic Director Mode	2-26
	2.13.1	Modes of Running Oracle Connection Manager in Traffic Director Mode	2-27
	2.13.2	Benefits of Oracle Connection Manager in Traffic Director Mode	2-28
2	.14 Sto	red Procedure Calls in JDBC Programs	2-29
	2.14.1	PL/SQL Stored Procedures	2-29
	2.14.2	Java Stored Procedures	2-30
2	.15 Abo	out Processing SQL Exceptions	2-30

Part II Oracle JDBC

3.1 Support for JDBC 4.2 Standard	3-1
3.2 Support for JDBC 4.3 Standard	3-2
Oracle Extensions	
Oracle Extensions 4.1 Overview of Oracle Extensions	4-1

	4.2.2	Sup	port for Oracle Data Types	4-2
	4.2.3	Sup	port for Oracle Objects	4-3
	4.2.4	Sup	port for Schema Naming	4-4
	4.2.5	DML	_ Returning	4-4
	4.2.6	PL/S	SQL Associative Arrays	4-5
4.3	B Orac	cle JD	BC Packages	4-5
	4.3.1	Pacl	kage oracle.sql	4-5
	4.3.2	Pacl	kage oracle.sql.json	4-10
	4.3.3	Pacl	kage oracle.jdbc	4-12
4.4	1 Orac	cle Ch	aracter Data Types Support	4-13
	4.4.1	SQL	CHAR Data Types	4-13
	4.4.2	SQL	NCHAR Data Types	4-13
	4.4.3	Clas	ss oracle.sql.CHAR	4-14
4.5	5 Add	itional	Oracle Type Extensions	4-17
	4.5.1	Orac	cle ROWID Type	4-17
	4.5.2	Orac	cle REF CURSOR Type Category	4-18
	4.5.3	Orac	cle BINARY_FLOAT and BINARY_DOUBLE Types	4-20
	4.5.4	Orac	cle SYS.ANYTYPE and SYS.ANYDATA Types	4-21
	4.5.5	The	oracle.jdbc Package	4-24
	4.	5.5.1	Interface oracle.jdbc.OracleConnection	4-25
	4.	5.5.2	Interface oracle.jdbc.OracleStatement	4-26
	4.	5.5.3	Interface oracle.jdbc.OraclePreparedStatement	4-26
	4.	5.5.4	Interface oracle.jdbc.OracleCallableStatement	4-27
	4.	5.5.5	Interface oracle.jdbc.OracleResultSet	4-27
	4.	5.5.6	Interface oracle.jdbc.OracleResultSetMetaData	4-28
	4.	5.5.7	Class oracle.jdbc.OracleTypes	4-28
4.6	6 DML	Retu	irning	4-32
	4.6.1	Orac	cle-Specific APIs	4-32
	4.6.2	Abo	ut Running DML Returning Statements	4-33
	4.6.3	Exa	mple of DML Returning	4-33
	4.6.4	Limi	tations of DML Returning	4-34
4.7	7 Acce	essing	PL/SQL Associative Arrays	4-35

5 Features Specific to JDBC Thin

5.1	Over	view of JDBC Thin Client	5-1
5.2	Additi	onal Features Supported	5-1
	5.2.1	Default Support for Native XA	5-1
	5.2.2	Support for Transaction Guard	5-2
	5.2.3	Support for Application Continuity	5-2



6 Features Specific to JDBC OCI Driver

6.1	OCI Connection Pooling	6-1
6.2	Transparent Application Failover	6-1
6.3	OCI Native XA	6-1
6.4	OCI Instant Client	6-2
6.5	About Instant Client Light (English)	6-2

7 Server-Side Internal Driver

7.1	Over	view of the Server-Side Internal Driver	7-1
7.2	Coni	necting to the Database	7-1
7.3	Abou	ut Session and Transaction Context	7-3
7.4	Testi	ng JDBC on the Server	7-4
7.5	Load	ling an Application into the Server	7-4
	7.5.1	Using the Loadjava Utility	7-4
	7.5.2	Using the JVM Command Line	7-6

Part III Connection and Security

8 Data Sources and URLs

8.1	Abou	t Data Sources	8-1
8	8.1.1	Overview of Oracle Data Source Support for JNDI	8-1
8	8.1.2	Features and Properties of Data Sources	8-2
8	8.1.3	Creating a Data Source Instance and Connecting	8-5
8	8.1.4	Creating a Data Source Instance, Registering with JNDI, and Connecting	8-5
8	8.1.5	Supported Connection Properties	8-7
8	8.1.6	About Using Roles for SYS Login	8-7
8	8.1.7	Configuring Database Remote Login	8-8
8	8.1.8	Using Bequeath Connection and SYS Logon	8-9
8	8.1.9	Setting Properties for Oracle Performance Extensions	8-9
8	8.1.10	Support for Network Data Compression	8-10
8.2	Data	base URLs and Database Specifiers	8-11
8	8.2.1	Support for Internet Protocol Version 6	8-11
8	8.2.2	Support for HTTPS Proxy Configuration	8-12
8	8.2.3	Database Specifiers	8-13
8	8.2.4	Thin-style Service Name Syntax	8-13
8	8.2.5	Support for Easy Connect Plus	8-14
	8.2	.5.1 Support for TCPS Protocol	8-14
	8.2	.5.2 Support for Multiple Host and Port	8-14



8.2	2.5.3 Support to Pass Connection Properties in the Connection String	8-15
8.2.6	Support for Delay in Connection Retries	8-17
8.2.7	TNSNames Alias Syntax	8-17
8.2.8	LDAP Syntax	8-18

9 JDBC Client-Side Security Features

9.1	Sup	oort for Token-Based Authentication for IAM	9-2
	9.1.1	Using the File System	9-3
	9.1.2	Using the oracle.jdbc.accessToken Connection Property	9-5
	9.1.3	Using the OracleConnectionBuilder Interface	9-6
	9.1.4	Using the OracleDataSource Class	9-6
9.2	Sup	port for Token-Based Authentication for Azure AD	9-7
	9.2.1	Using the File System	9-7
	9.2.2	Using the oracle.jdbc.accessToken Connection Property	9-9
	9.2.3	Using the OracleConnectionBuilder Interface	9-10
	9.2.4	Using the OracleDataSource Class	9-10
9.3	Sup	port for Oracle Advanced Security	9-11
	9.3.1	Overview of Oracle Advanced Security	9-11
	9.3.2	JDBC OCI Driver Support for Oracle Advanced Security	9-12
	9.3.3	JDBC Thin Driver Support for Oracle Advanced Security	9-13
9.4	Sup	port for Login Authentication	9-13
9.5	Sup	port for Strong Authentication	9-14
9.6	Sup	port for Network Encryption and Integrity	9-15
	9.6.1	Overview of JDBC Support for Network Encryption and Integrity	9-15
	9.6.2	JDBC OCI Driver Support for Encryption and Integrity	9-16
	9.6.3	JDBC Thin Driver Support for Encryption and Integrity	9-17
	9.6.4	Setting Encryption and Integrity Parameters in Java	9-18
9.7	Sup	port for TLS	9-20
	9.7.1	Overview of JDBC Support for TLS	9-20
	9.7.2	About Managing Certificates and Wallets	9-22
	9.7.3	About Keys and certificates containers	9-22
	9.7.4	Database Connectivity Over TLS Version 1.2 Using JDBC Thin and JKS	9-22
	9.7.5	Automatic TLS Connection Configuration	9-23
	9.	7.5.1 Provider Resolution	9-23
	9.	7.5.2 Automatic Key Store Type (KSS) Resolution	9-24
	9.7.6	Support for Default TLS Context	9-25
	9.7.7	Support for Key Store Service	9-25
9.8	Sup	port for Kerberos	9-26
	9.8.1	Overview of JDBC Support for Kerberos	9-26
	9.8.2	Configuring Windows to Use Kerberos	9-26



1	9.8.3	Configuring Oracle Database to Use Kerberos	9-27
1	9.8.4	Code Example for Using Kerberos	9-28
1	9.8.5	Support for Kerberos Constrained Delegation	9-32
9	9.8.6	Kerberos Authentication Enhancements	9-33
	9.8	.6.1 Kerberos Authentication Using the User and the Password Properties	9-33
	9.8	.6.2 Kerberos Authentication Using the JAAS Configuration	9-34
9.9	Supp	ort for RADIUS	9-35
1	9.9.1	Overview of JDBC Support for RADIUS	9-35
9	9.9.2	Configuring Oracle Database to Use RADIUS	9-36
1	9.9.3	Code Example for Using RADIUS	9-36
9	9.9.4	Support for Challenge-Response Authentication	9-38
9.10	Sec	ure External Password Store	9-40

10 JDBC Service Provider Extensions

10.1 Cent	ralized Configuration Providers	10-3
10.1.1	Azure App Configuration	10-4
10.1.2	OCI Object Storage	10-6
10.1.3	OCI Database Tools	10-8
10.1.4	Built-in Configuration Providers	10-8
10.2 Reso	purce Providers	10-9
10.3 Trac	e Event Listener Providers	10-10
10.4 JDB	C Extensions for Cloud Vendors	10-11

11 Proxy Authentication

11.1	About Proxy Authentication	11-1
11.2	Types of Proxy Connections	11-2
11.3	Creating Proxy Connections	11-3
11.4	Closing a Proxy Session	11-5
11.5	Caching Proxy Connections	11-5
11.6	Limitations of Proxy Connections	11-6

Part IV Data Access and Manipulation

12 Accessing and Manipulating Oracle Data

12-1
12-1
12-4
12-4



12	2.2.1	Stan	dard Types Versus Oracle Types	12-4
12.2.2 Abo		Abou	It Converting SQL NULL Data	12-5
12	2.2.3	Abou	It Testing for NULLs	12-6
12.3	Resi	ult Set	and Statement Extensions	12-6
12.4	Com	pariso	n of Oracle get and set Methods to Standard JDBC	12-6
12	2.4.1	Stan	dard getObject Method	12-7
12	2.4.2	Orac	le getOracleObject Method	12-7
12	2.4.3	Sum	mary of getObject and getOracleObject Return Types	12-8
12	2.4.4	Othe	r getXXX Methods	12-10
	12.4	1.4.1	Return Types of getXXX Methods	12-11
	12.4	1.4.2	Special Notes about getXXX Methods	12-11
12	2.4.5	Data	Types For Returned Objects from getObject and getXXX	12-12
12	2.4.6	The	setObject and setOracleObject Methods	12-12
12	2.4.7	Othe	r setXXX Methods	12-13
	12.4	1.7.1	Input Data Binding	12-13
	12.4	1.7.2	Method setFixedCHAR for Binding CHAR Data into WHERE Clauses	12-15
12.5	Usin	g Res	ult Set Metadata Extensions	12-16
12.6	Abou	ut Usir	ng SQL CALL and CALL INTO Statements	12-17

13 Java Streams in JDBC

	13.1	Overview of Java Streams		
	13.2	13.2 About Streaming LONG or LONG RAW Columns		13-2
	13	3.2.1	Overview of Streaming LONG or LONG RAW Columns	13-2
	13	3.2.2	LONG RAW Data Conversions	13-3
	13	3.2.3	LONG Data Conversions	13-3
	13	3.2.4	Examples:Streaming LONG RAW Data	13-4
	13	3.2.5	About Avoiding Streaming for LONG or LONG RAW	13-6
	13.3	Abou	ut Streaming CHAR, VARCHAR, or RAW Columns	13-7
13.4 About Streaming LOBs and External Files		13-7		
13.5 Relation Between Data Streaming and Multiple Columns			13-8	
13.6 Closing a Stream		13-10		
13.7 Notes and Precautions on Streams		13-10		
	13	3.7.1	About Streaming Data Precautions	13-10
	13	3.7.2	About Using Streams to Avoid Limits on setBytes and setString	13-11
	13	3.7.3	Relation Between Streaming and Row Prefetching	13-11

14 Working with Oracle Object Types

14.1	About Mapping Oracle Objects	14-1
14.2	About Using the Default STRUCT Class for Oracle Objects	14-2



14.2.1	Ove	rview of Using the Struct Class	14-3
14.2.2		ieving STRUCT Objects and Attributes	14-3
14.2.3		ut Creating STRUCT Objects	14-4
14.2.4		ling STRUCT Objects into Statements	14-4
14.2.5		UCT Automatic Attribute Buffering	14-4
14.3 Abo		ating and Using Custom Object Classes for Oracle Objects	14-5
14.3.1		rview of Creating and Using Custom Object Classes	14-6
14.3.2		ative Advantages of OracleData versus SQLData	14-6
14.3.3	Abo	ut Type Maps for SQLData Implementations	14-7
14.3.4		ut Creating Type Map and Defining Mappings for a SQLData ementation	14-7
14.	3.4.1	Overview of Creating a Type Map and Defining Mappings	14-8
14.	3.4.2	Adding Entries to an Existing Type Map	14-8
14.	3.4.3	Creating a New Type Map	14-9
14.	3.4.4	About Materializing Object Types not Specified in the Type Map	14-9
14.3.5	Abo	ut Reading and Writing Data with a SQLData Implementation	14-10
14.3.6	Abo	ut the OracleData Interface	14-12
14.3.7	Abo	ut Reading and Writing Data with an OracleData Implementation	14-14
14.3.8	Add	itional Uses of OracleData	14-16
14.4 Obje	ect-Typ	be Inheritance	14-17
14.4.1	Abo	ut Creating Subtypes	14-17
14.4.2	Abo	ut Implementing Customized Classes for Subtypes	14-18
14.	4.2.1	About Using OracleData for Type Inheritance Hierarchy	14-19
14.	4.2.2	About UsingSQLData for Type Inheritance Hierarchy	14-21
14.4.3	Abo	ut Retrieving Subtype Objects	14-23
14.4.4	Crea	ating Subtype Objects	14-26
14.4.5	Sen	ding Subtype Objects	14-26
14.4.6	Acce	essing Subtype Data Fields	14-26
14.4.7	Inhe	ritance Metadata Methods	14-28
14.5 Abo	ut Des	scribing an Object Type	14-28
14.5.1	Fund	ctionality for Getting Object Metadata	14-28
14.5.2	Retr	ieving Object Metadata	14-29

15 Working with Large Objects and SecureFiles

15.1	The I	_OB Data Types	15-1
15.2	Persi	stent LOBs	15-2
15.3	Temp	oorary LOBs	15-3
15.4	Data	Interface for LOBs	15-4
15	.4.1	Input	15-4
15	.4.2	Output	15-6
15	.4.3	CallableSatement and IN OUT Parameter	15-7



15.4.4 Size Limitations	15-7
15.5 Locator Interface for LOBs	15-8
15.5.1 LOB prefetching	15-9
15.5.2 LOB Open and Close Operations	15-9
15.6 BFILES	15-10
15.7 JDBC Best Practices for LOB	15-11

16 Using Oracle Object References

16.1	16.1 Oracle Extensions for Object References		16-1
16.2	16.2 Retrieving and Passing an Object Reference		
1	6.2.1	Retrieving an Object Reference from a Result Set	16-2
1	6.2.2	Retrieving an Object Reference from a Callable Statement	16-3
1	6.2.3	Passing an Object Reference to a Prepared Statement	16-3
16.3	Acce	essing and Updating Object Values Through an Object Reference	16-4

17 Working with Oracle Collections

17.1	Orac	le Ext	ensions for Collections	17-1
1	7.1.1	Over	view of Oracle Collections	17-1
1	7.1.2	Choi	ces in Materializing Collections	17-2
1	7.1.3	Crea	ting Collections	17-2
1	7.1.4	Crea	ting Multilevel Collection Types	17-3
17.2	Ove	view o	of Collection Functionality	17-3
17.3	ARR	AY Pe	rformance Extension Methods	17-4
1	7.3.1	Abou	It Accessing oracle.sql.ARRAY Elements as Arrays of Java Primitive	
		Туре	S	17-4
1	7.3.2	ARR	AY Automatic Element Buffering	17-5
1	7.3.3	ARR	AY Automatic Indexing	17-5
17.4	Crea	ting a	nd Using Arrays	17-6
1	7.4.1	Crea	ting ARRAY Objects	17-6
1	7.4.2	Retri	eving an Array and Its Elements	17-7
	17.4	1.2.1	About Retrieving the Array	17-8
	17.4	1.2.2	Data Retrieval Methods	17-8
	17.4	1.2.3	Comparing the Data Retrieval Methods	17-9
	17.4	1.2.4	Retrieving Elements of a Structured Object Array According to a Type	
			Мар	17-10
	17.4	1.2.5	Retrieving a Subset of Array Elements	17-10
	17.4	1.2.6	Retrieving Array Elements into an oracle.sql.Datum Array	17-11
	17.4	1.2.7	About Accessing Multilevel Collection Elements	17-12
1	7.4.3	Pass	ing Arrays to Statement Objects	17-13



18 Result Set

18.1	3.1 Oracle JDBC Implementation Overview for Result Set Support		18-1
18.2	Resu	Iltset Limitations and Downgrade Rules	18-2
18.3	Abou	ut Avoiding Update Conflicts	18-4
18.4	Row	Fetch Size	18-4
18.4.1 Setting the Fetch Size		18-5	
18	8.4.2	Presetting the Fetch Direction	18-5
18.5	3.5 About Refetching Rows		18-5
18.6	18.6 About Viewing Database Changes Made Internally and Externally		18-6
18	8.6.1	Visibility versus Detection of External Changes	18-7
18	8.6.2	Summary of Visibility of Internal and External Changes	18-7
18	8.6.3	Oracle Implementation of Scroll-Sensitive Result Sets	18-8

19 JDBC RowSets

19.1	Overview of JDBC RowSets			
19	9.1.1	RowSet Properties	19-2	
19	9.1.2	Events and Event Listeners	19-3	
19	9.1.3	Command Parameters and Command Execution	19-4	
19	9.1.4	About Traversing RowSets	19-4	
19.2	Abou	ut CachedRowSet	19-6	
19.3	Abou	ut JdbcRowSet	19-9	
19.4	Abou	ut WebRowSet	19-10	
19.5	Abou	ut FilteredRowSet	19-12	
19.6	Abou	ut JoinRowSet	19-14	

20 Globalization Support

20.1	About Providing Globalization Support	20-1
20.2	NCHAR, NVARCHAR2, NCLOB and the defaultNChar Property	20-3
20.3	New Methods for National Character Set Type Data in JDK 6	20-4

Part V Performance and Scalability

21 Statement and Result Set Caching

21.1 Abo	ut Statement Caching	21-1
21.1.1	Basics of Statement Caching	21-2



21.1.2	Implicit Statement Caching	21-2
21.1.3	Explicit Statement Caching	21-3
21.2 Abo	ut Using Statement Caching	21-4
21.2.1	About Enabling and Disabling Statement Caching	21-4
21.2.2	About Closing a Cached Statement	21-6
21.2.3	About Using Implicit Statement Caching	21-7
21.	2.3.1 Methods Used in Statement Allocation and Implicit Statement Caching	21-8
21.2.4	About Using Explicit Statement Caching	21-10
21.	2.4.1 Methods Used to Retrieve Explicitly Cached Statements	21-11
21.3 Abo	ut Reusing Statements Objects	21-11
21.3.1	About Using a Pooled Statement	21-11
21.3.2	About Closing a Pooled Statement	21-12
21.4 Abo	ut Result Set Caching	21-13
21.4.1	Server-Side Result Set Cache	21-13
21.4.2	Client-Side Result Set Cache	21-14
21.	4.2.1 Enabling the Client-Side Result Set Cache	21-14
21.	1.2.2 Benefits of Client-Side Result Set Cache	21-15
21.	1.2.3 Usage Guidelines in JDBC	21-15

22 Performance Extensions

22.1	Upda	te Ba	tching	22-1
22	2.1.1	Over	view of Update Batching	22-1
22	2.1.2	Stan	dard Update Batching	22-2
	22.1.	2.1	About Adding Operations to the Batch	22-2
	22.1.	2.2	About Processing the Batch	22-3
	22.1.	2.3	Row Count per Iteration for Array DMLs	22-4
	22.1.	2.4	About Committing the Changes in the Oracle Implementation of Standard Batching	22-4
	22.1.	2.5	About Clearing the Batch	22-4
	22.1.	2.6	Update Counts in the Oracle Implementation of Standard Batching	22-5
	22.1.	2.7	Error Handling in the Oracle Implementation of Standard Batching	22-6
	22.1.	2.8	About Intermixing Batched Statements and Nonbatched Statements	22-7
	22.1.	2.9	Limitations in the Oracle Implementation of Standard Batching	22-8
22	2.1.3	Pren	nature Batch Flush	22-8
22.2	Additi	onal	Oracle Performance Extensions	22-9
22	2.2.1	Orac	le Row-Prefetching Limitations	22-9
22	2.2.2	Abou	It Defining Column Types	22-11
22	2.2.3	Abou	It Reporting DatabaseMetaData TABLE_REMARKS	22-14



23 JDBC Reactive Extensions

	23.1	Over	view o	of JDBC Reactive Extensions	23-1
	23.2	Aboι	ut Buile	ding an Application with Reactive Extensions	23-2
	23	23.2.1 Opening a Connection Using Asynchronous Methods			23-2
	23	3.2.2	Exec	ution of SQL Statements with Asynchronous Methods	23-3
		23.2	2.2.1	Standard SQL Statement Execution with the executeAsyncOracle Method	23-3
		23.2	2.2.2	DML Statement Execution with the executeUpdateAsyncOracle method	23-5
		23.2	2.2.3	Batch DML Statement Execution with the executeBatchAsyncOracle Method	23-6
		23.2	2.2.4	SQL Query Execution with the executeQueryAsyncOracle Method	23-7
	23.2.4 Read		Abou	It Fetching Row Data with Asynchronous Methods	23-7
			Read	ling LOB Data Using Asynchronous Methods	23-10
			Writi	ng LOB Data Using Asynchronous Methods	23-11
	23	3.2.6	Com	mitting a Transaction Using Asynchronous Methods	23-13
	23	3.2.7	Clos	ing a Connection Using Asynchronous Methods	23-14
	23.3	Thre	ading	Model of Asynchronous Methods	23-15
	23.4	Abou	ut the I	Flow API	23-15
	23.5	Usin	g the I	FlowAdapters Class	23-16
	23.6	Strea	aming	Row Data with the Reactor Library	23-16
	23.7	Strea	aming	Row Data with the RxJava Library	23-18
	23.8	Strea	aming	Row Data with the Akka Streams Library	23-19
	23.9 Limitations of JDBC Reactive Extensions			23-20	

24 Support for Java library for Reactive Streams Ingestion

24.1 Ove	erview of the Java Library for Reactive Streams Ingestion	24-1
24.2 Fea	tures of the Java Library for Reactive Streams Ingestion	24-2
24.2.1	Reactive Streams Ingestion	24-2
24.2.2	Direct Path Load	24-2
24.2.3	Universal Connection Pool	24-3
24.3 Abo	out Reactive Streams Ingestion (RSI) Modes	24-3
24.3.1	Enabling the DataLoad Mode	24-4
24.4 Co	le Samples: Java Library for Reactive Streams Ingestion	24-4
24.4.1	PushPublisher	24-4
24.4.2	Flow.Publisher Dynamic Implementations	24-5
24.4.3	Flow.Publisher Third-Party implementations	24-7
24.5 Lim	itations of Java library for Reactive Streams Ingestion	24-9



25 Support for Pipelined Database Operations

25-1
25-1
25-2
25-3

26 OCI Connection Pooling

26.1	Back	26-1	
26.2	Com	parison Between OCI Driver Connection Pooling and Shared Servers	26-2
26.3	Abou	ut Defining an OCI Connection Pool	26-2
2	6.3.1	Overview of Creating an OCI Connection Pool	26-2
2	6.3.2	Importing the oracle.jdbc.pool and oracle.jdbc.oci Packages	26-3
2	26.3.3 Creating an OCI Connection Pool		26-3
2	26.3.4 Setting the OCI Connection Pool Parameters		26-4
2	6.3.5	Checking the OCI Connection Pool Status	26-5
26.4	Abou	ut Connecting to an OCI Connection Pool	26-6
26.5	Sam	ple Code for OCI Connection Pooling	26-7
26.6	Statement Handling and Caching		26-9
26.7	JNDI and the OCI Connection Pool		26-10

27 Database Resident Connection Pooling

27.1	Overview of Database Resident Connection Pooling	27-1
27.2	Enabling Database Resident Connection Pooling	27-2
2	7.2.1 Enabling DRCP on the Server Side	27-2
2	7.2.2 Enabling DRCP on the Client Side	27-3
27.3	Pooled Server Processes Across Multiple Connection Pools	27-4
27.4	Multi-Pool Support in DRCP	27-4
27.5	Tagging Support in Database Resident Connection Pooling	27-5
27.6	PL/SQL Callback for Session State Fix Up	27-6
27.7	APIs for Using Database Resident Connection Pooling	27-7

28 JDBC Support for Database Sharding

28.1 Ove	erview of Database Sharding for JDBC Users	28-1
28.2 Abo	out Building the Sharding Key	28-3
28.3 API	s for Database Sharding Support	28-5
28.3.1	The OracleShardingKey Interface	28-5
28.3.2	The OracleShardingKeyBuilder Interface	28-6
28.3.3	The OracleConnectionBuilder Interface	28-6



28.3.4	Other New Classes and Methods for Database Sharding Support	28-7
28.4 JDB	C Sharding Example	28-7
28.5 Ove	rview of the Sharding Data Source	28-8
28.5.1	Benefits of the Sharding Data Source	28-9
28.5.2	Example: How to Use the Sharding Data Source	28-9
28.5.3	Limitations of the Sharding Data Source	28-10

29 Oracle Advanced Queuing

29.1	Functionality and Framework of Oracle Advanced Queuing		
29.2	Making Changes to the Database	29-2	
29.3	AQ Asynchronous Event Notification	29-3	
29.4	About Creating Messages	29-5	
29	0.4.1 Creating Messages	29-5	
29.4.2 AQ Message Properties		29-6	
29.4.3 AQ Message Payload		29-7	
29.5	Example: Creating a Message and Setting a Payload	29-7	
29.6	6 Enqueuing Messages		
29.7	Dequeuing Messages		
29.8	Examples: Enqueuing and Dequeuing	29-10	

30 Continuous Query Notification

-				
30.1	Overview of Continuous Query Notification	30-1		
30.2	Overview of Client Initiated Continuous Query Notification	30-2		
30.3	Creating a Registration 30-			
30	0.3.1 Continuous Query Notification Registration Options	30-3		
30.4	Associating a Query with a Registration	30-4		
30.5	Notifying Database Change Events 30-			
30.6	Deleting a Registration	30-5		

Part VI High Availability

31 Transaction Guard for Java

Over	view of Transaction Guard for Java	31-1
Tran	saction Guard Support for XA Transactions	31-1
How	to Use Transaction Guard with XA	31-2
31.4 Transaction Guard for Java APIs		31-3
L.4.1	Retrieving the Logical Transaction Identifiers	31-3
L.4.2	Retrieving the Updated Logical Transaction Identifiers	31-4
	Tran How Tran L.4.1	L.4.1 Retrieving the Logical Transaction Identifiers



31.4.2.1	Registering Event Listeners	31-4
31.4.2.2	Unregistering Event Listeners	31-4
Complete E	Example:Using Transaction Guard APIs	31-4
About Usin	g Server-Side Transaction Guard APIs	31-5
	31.4.2.2 Complete E	 31.4.2.1 Registering Event Listeners 31.4.2.2 Unregistering Event Listeners Complete Example:Using Transaction Guard APIs About Using Server-Side Transaction Guard APIs

32 Application Continuity for Java

32.1	Abou	ut Cor	nfiguring Oracle JDBC for Application Continuity for Java	32-2
3	2.1.1	Sup	port for Concrete Classes with Application Continuity	32-5
3	2.1.2	Abo	ut Using LONG and LONG RAW columns with Application Continuity	32-5
32.2	Abou	ut Cor	nfiguring Oracle Database for Application Continuity for Java	32-6
32.3	Appl	icatior	n Continuity with DRCP	32-7
32.4	Appl	icatior	n Continuity Support for XA Data Source	32-8
32.5	Abou	ut Ider	ntifying Request Boundaries in Application Continuity for Java	32-9
32.6	Supp	oort fo	or Transparent Application Continuity	32-10
3	2.6.1	Sup	port for Session State Stable Cursors	32-11
32.7	Esta	blishir	ng the Initial State Before Application Continuity Replays	32-12
3	2.7.1	No C	Callback	32-12
3	2.7.2	Con	nection Labeling	32-12
3	2.7.3	Con	nection Initialization Callback	32-13
	32.7	7.3.1	Creating an Initialization Callback	32-13
	32.7	7.3.2	Registering an Initialization Callback	32-14
	32.7	7.3.3	Removing or Unregistering an Initialization Callback	32-14
3	2.7.4	Abo	ut Enabling FAILOVER_RESTORE	32-14
32.8	Abou	ut Dela	aying the Reconnection in Application Continuity for Java	32-17
3	2.8.1	Con	figuration Examples Related to Application Continuity for Java	32-17
	32.8	3.1.1	Creating Services on Oracle RAC	32-18
	32.8	3.1.2	Modifying Services on Single-Instance Databases	32-18
32.9	Abou	ut Ret	aining Mutable Values in Application Continuity for Java	32-19
3	2.9.1	Grar	nt and Revoke Interface	32-19
	32.9	9.1.1	Dates and SYS_GUID Syntax	32-19
	32.9	9.1.2	Sequence Syntax	32-20
	32.9	9.1.3	GRANT ALL Statement	32-20
	32.9	9.1.4	Rules for Grants on Mutable Values	32-20
32.10	App	olicatio	on Continuity Statistics	32-20
32.11	Abo	out Dis	sabling Replay in Application Continuity for Java	32-22
3	2.11.1	Hov	w to Disable Replay	32-22
3	2.11.2	Wh	ien to Disable Replay	32-22
	32.1	1.2.1	Application Calls External Systems that Should not Be Repeated	32-23
	32.1	1.2.2	Application Synchronizes Independent Sessions	32-23
	32.1	1.2.3	Application Uses Time at the Middle-tier in the Execution Logic	32-24

32.11.2.4	Application assumes that ROWIds do not change	32-24
32.11.2.5	Application Assumes that Side Effects Execute Once	32-24
32.11.2.6	Application Assumes that Location Values Do not Change	32-24
32.11.3 Diag	nostics and Tracing	32-25
32.11.3.1	Writing Replay Trace to Console	32-25
32.11.3.2	Writing Replay Trace to a File	32-25

33 Oracle JDBC Support for FAN Events

33.1	Overview of Oracle JDBC Support for FAN events	33-1
33.2	Safe Draining APIs for Planned Maintenance	33-2
33.3	Installation and Configuration of Oracle JDBC Driver for FAN Events Support	33-3
33.4	Example of Oracle JDBC Driver FAN support for Planned Maintenance	33-4
33.5	Using Third-Party Connection Pools with Oracle JDBC	33-5

34 Transparent Application Failover

34.1	Overview of Transparent Application Failover	34-1
34.2	Failover Type Events	34-1
34.3	TAF Callbacks	34-2
34.4	Java TAF Callback Interface	34-2
34.5	Comparison of TAF and Fast Connection Failover	34-3

35 Single Client Access Name

35.1	Overview of Single Client Access Name	35-1
35.2	About Configuring the Database Using the SCAN	35-2
35.3	How Connection Load Balancing Works Using the SCAN	35-2
35.4	Version and Backward Compatibility	35-4
35.5	Using the SCAN in a Maximum Availability Architecture Environment	35-5
35.6	Using the SCAN With Oracle Connection Manager	35-6

Part VII Transaction Management

36 Distributed Transactions

36.1	Abou	ut Distributed Transactions	36-1
36	6.1.1	Overview of Distributed Transaction	36-1
36	6.1.2	Distributed Transaction Components and Scenarios	36-2
36	6.1.3	Distributed Transaction Concepts	36-2
36	6.1.4	About Switching Between Global and Local Transactions	36-4



36.1.5	Oracle XA Packages	36-5
36.2 XA C	Components	36-6
36.2.1	XADatasource Interface and Oracle Implementation	36-6
36.2.2	XAConnection Interface and Oracle Implementation	36-7
36.2.3	XAResource Interface and Oracle Implementation	36-8
36.2.4	OracleXAResource Method Functionality and Input Parameters	36-8
36.2.5	Xid Interface and Oracle Implementation	36-13
36.3 Erro	r Handling and Optimizations	36-14
36.3.1	XAException Classes and Methods	36-14
36.3.2	Mapping Between Oracle Errors and XA Errors	36-15
36.3.3	XA Error Handling	36-16
36.3.4	Oracle XA Optimizations	36-16
36.4 Abou	ut Implementing a Distributed Transaction	36-16
36.4.1	Summary of Imports for Oracle XA	36-16
36.4.2	Oracle XA Code Sample	36-17
36.5 Nativ	ve-XA in Oracle JDBC Drivers	36-21
36.5.1	OCI Native XA	36-22
36.5.2	Thin Native XA	36-23

Part VIII Manageability

37-1
37-2
37-2
37-3
37-3
37-4
37-4
38-1
38-2
39-2
39-2
39-3
-



Part IX Appendixes

JD	BC Reference Information	
A.1	Supported SQL-JDBC Data Type Mappings	A-1
A.2	Supported SQL and PL/SQL Data Types	A-3
A.3	About Using PL/SQL Types	A-7
A.4	Using Embedded JDBC Escape Syntax	A-9
	A.4.1 Time and Date Literals	A-10
	A.4.1.1 Date Literals	A-10
	A.4.1.2 Time Literals	A-11
	A.4.1.3 Timestamp Literals	A-11
	A.4.2 Scalar Functions	A-12
	A.4.3 LIKE Escape Characters	A-13
	A.4.4 MATCH_RECOGNIZE Clause	A-13
	A.4.5 Outer Joins	A-14
	A.4.6 Function Call Syntax	A-14
	A.4.7 JDBC Escape Syntax to Oracle SQL Syntax Example	A-14
A.5	Oracle JDBC Notes and Limitations	A-15
	A.5.1 CursorName	A-15
	A.5.2 JDBC Outer Join Escapes	A-15
	A.5.3 IEEE 754 Floating Point Compliance	A-16
	A.5.4 Catalog Arguments to DatabaseMetaData Calls	A-16
	A.5.5 SQLWarning Class	A-16
	A.5.6 Executing DDL Statements	A-16
	A.5.7 Binding Named Parameters	A-16
Ora	acle RAC Fast Application Notification	
B.1	Overview of Oracle RAC Fast Application Notification	B-1
B.2	Installing and Configuring Oracle RAC Fast Application Notification	B-3
B.3	Using Oracle RAC Fast Application Notification	B-3
B.4	Implementing a Connection Pool	B-5
JD	BC Coding Tips	
C.1	JDBC and Multithreading	C-1

		5	
C.2	Perfo	ormance Optimization of JDBC Programs	C-1
	C.2.1	Disabling Auto-Commit Mode	C-2



(C.2.2	Stan	dard Fetch Size and Oracle Row Prefetching	C-3
(C.2.3	Abou	ut Setting the Session Data Unit Size	C-3
	C.2	2.3.1	About Setting the SDU Size for the Database Server	C-3
	C.2	2.3.2	About Setting the SDU Size for JDBC OCI Client	C-4
	C.2	.3.3	About Setting the SDU Size for JDBC Thin Client	C-4
(C.2.4	JDBO	C Update Batching	C-4
(C.2.5	State	ement Caching	C-4
(C.2.6	Марр	ping Between Built-in SQL and Java Types	C-5
C.3	Trans	sactio	n Isolation Levels and Access Modes in JDBC	C-6

D JDBC Error Messages

Gene	eral Structure of JDBC Error Messages	D-1
Gene	eral JDBC Messages	D-1
D.2.1	JDBC Messages Sorted by ORA Number	D-1
0.2.2	JDBC Messages Sorted in Alphabetic Order	D-7
Nativ	e XA Messages	D-12
0.3.1	Native XA Messages Sorted by ORA Number	D-12
0.3.2	Native XA Messages Sorted in Alphabetic Order	D-12
TTC	Messages	D-13
0.4.1	TTC Messages Sorted by ORA Number	D-13
0.4.2	TTC Messages Sorted in Alphabetic Order	D-14
	Gene).2.1).2.2 Nativ).3.1).3.2 TTC).4.1	 D.2.2 JDBC Messages Sorted in Alphabetic Order Native XA Messages D.3.1 Native XA Messages Sorted by ORA Number D.3.2 Native XA Messages Sorted in Alphabetic Order TTC Messages D.4.1 TTC Messages Sorted by ORA Number

E Troubleshooting

E.1	Com	mon F	Problems	E-1
	E.1.1	Mem Varia	ory Consumption for CHAR Columns Defined as OUT or IN/OUT bles	E-1
	E.1.2	Mem	ory Leaks and Running Out of Cursors	E-1
	E.1.3	Oper	ning More than 16 OCI Connections for a Process	E-2
	E.1.4	Using	g statement.cancel	E-2
	E.1.5	Usin	g JDBC with Firewalls	E-4
	E.1.6	Freq	uent Abrupt Disconnection from Server	E-4
	E.1.7	Netw	ork Adapter Cannot Establish Connection	E-4
	E.1	7.1	Oracle Instance Configured with MTS Server Uses Shared Server	E-5
	E.1	7.2	JDBC Thin Driver with NIC Card Supporting Both IPv4 and IPv6	E-6
	E.1	7.3	Sample Application	E-6
E.2	Basic	c Debi	ugging Procedures	E-7
	E.2.1	Orac	le Net Tracing to Trap Network Events	E-8
	E.2	.1.1	Client-Side Tracing	E-8
	E.2	.1.2	Server-Side Tracing	E-10



E.2.2 Third Party Debugging Tools

Index

List of Tables

1-1	Feature Differences Between the JDBC OCI Driver and the JDBC Thin Driver	1-5
1-2	Feature List	1-9
2-1	Import Statements for JDBC Driver	2-7
2-2	Error Messages for Operations Performed When Auto-Commit Mode is ON	2-13
4-1	Key Interfaces and Classes of the oracle.jdbc Package	4-24
8-1	Standard Data Source Properties	8-3
8-2	Oracle Extended Data Source Properties	8-3
8-3	Supported Database Specifiers	8-13
9-1	Client/Server Negotiations for Encryption or Integrity	9-16
9-2	OCI Driver Client Parameters for Encryption and Integrity	9-17
9-3	Thin Driver Client Parameters for Encryption and Integrity	9-17
12-1	Default Mappings Between SQL Types and Java Types	12-2
12-2	getObject and getOracleObject Return Types	12-9
13-1	LONG and LONG RAW Data Conversions	13-4
18-1	Visibility of Internal and External Changes for Oracle JDBC	18-7
19-1	The JDBC and Cached Row Sets Compared	19-9
21-1	Comparing Methods Used in Statement Caching	21-4
21-2	Methods Used in Statement Allocation and Implicit Statement Caching	21-8
21-3	Methods Used to Retrieve Explicitly Cached Statements	21-11
22-1	Valid Column Type Specifications	22-13
23-1	Method Comparison	23-3
23-2	Emission to Multiple Subscribers	23-9
30-1	Continuous Query Notification Registration Options	30-3
35-1	Oracle Client and Oracle Database Version Compatibility for the SCAN	35-4
36-1	Connection Mode Transitions	36-4
36-2	Oracle-XA Error Mapping	36-15
37-1	Supported Database Startup Options	37-2
37-2	Supported Database Shutdown Options	37-3
A-1	Valid SQL Data Type-Java Class Mappings	A-1
A-2	Support for SQL Data Types	A-3
A-3	Support for ANSI-92 SQL Data Types	A-4
A-4	Support for SQL User-Defined Types	A-4
A-5	Support for PL/SQL Data Types	A-5
C-1	Mapping of SQL Data Types to Java Classes that Represent SQL Data Types	C-5
D-1	JDBC Messages Sorted by ORA Number	D-2



D-2	JDBC Messages Sorted in Alphabetic Order	D-7
D-3	Native XA Messages Sorted by ORA Number	D-12
D-4	Native XA Messages Sorted in Alphabetic Order	D-12
D-5	TTC Messages Sorted by ORA Number	D-13
D-6	TTC Messages Sorted in Alphabetic Order	D-14



Preface

This preface introduces you to the *Oracle Database JDBC Developer's Guide* discussing the intended audience, structure, and conventions of this document. A list of related Oracle documents is also provided.

Audience

The Oracle Database JDBC Developer's Guide is intended for developers of Java Database Connectivity (JDBC)-based applications. This book can be read by anyone with an interest in JDBC programming, but assumes at least some prior knowledge of the following:

- Java
- Oracle PL/SQL
- Oracle databases

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

For more information, see the following documents in the Oracle Database documentation set:

- Oracle Database Java Developer's Guide
- Oracle Database Development Guide
- Oracle Database PL/SQL Packages and Types Reference
- Oracle Database PL/SQL Language Reference
- Oracle Database SQL Language Reference

You can also find more information on the following pages:

- http://www.oracle.com/technetwork/documentation/index.html
- http://www.oracle.com/technetwork/java/javase/jdbc/index.htm



Conventions

This section describes the conventions used in the text and code examples of this documentation set. It describes:

- Conventions in Text
- Conventions in Code Examples
- Conventions for Windows Operating Systems

Conventions in Text

We use various conventions in text to help you more quickly identify special terms. The following table describes those conventions and provides examples of their use.

Convention	Meaning	Example
Bold	Bold typeface indicates terms that are defined in the text or terms that appear in a glossary, or both.	When you specify this clause, you create an index-organized table .
Italics	Italic typeface indicates book titles or emphasis.	Oracle Database Concepts
	empirasis.	Ensure that the recovery catalog and target database do <i>not</i> reside on the same disk.
UPPERCASE monospace	Uppercase monospace typeface indicates elements supplied by the system. Such	You can specify this clause only for a NUMBER column.
(fixed-width) font	elements include parameters, privileges, data types, RMAN keywords, SQL	You can back up the database by using the BACKUP command.
	supplied column names, database objects and structures, user names, and roles.	Query the TABLE_NAME column in the USER_TABLES data dictionary view.
		Use the DBMS_STATS.GENERATE_STATS procedure.
lowercase	 executables, file names, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, user names and roles, program units, and parameter values. Note: Some programmatic elements use a mixture of UPPERCASE and lowercase. 	Enter sqlplus to start SQL*Plus.
monospace		The password is specified in the orapwd file.
(fixed-width) font		Back up the data files and control files in the / disk1/oracle/dbs directory.
		The department_id, department_name, and
		location_id columns are in the hr.departments table.
		Set the QUERY REWRITE ENABLED initialization
		parameter to true.
		Connect as oe user.
		The $\ensuremath{\mathtt{JRepUtil}}$ class implements these methods.
lowercase	Lowercase italic monospace font	You can specify the parallel_clause.
italic monospace (fixed-width) font	represents placeholders or variables.	Run <i>old_release</i> .SQL where <i>old_release</i> refers to the release you installed prior to upgrading.



Conventions in Code Examples

Code examples illustrate Java, SQL, and command-line statements. Examples are displayed in a monospace (fixed-width) font and separated from normal text as shown in this example:

SELECT username FROM dba users WHERE username = 'MIGRATE';

The following table describes typographic conventions used in code examples and provides examples of their use.

Convention	Meaning	Example
[]	Brackets enclose one or more optional items. Do not enter the brackets.	DECIMAL (digits [, precision])
{ }	Braces enclose two or more items, one of which is required. Do not enter the braces.	{ENABLE DISABLE}
I	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	{ENABLE DISABLE} [COMPRESS NOCOMPRESS]
	Horizontal ellipsis points indicate either:	
	That we have omitted parts of the code that are not directly related to the	CREATE TABLE AS subquery;
	 example That you can repeat a portion of the code 	<pre>SELECT col1, col2,, coln FROM employees;</pre>
:	Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example.	SQL> SELECT NAME FROM V\$DATAFILE; NAME
		<pre>/fsl/dbs/tbs_01.dbf /fsl/dbs/tbs_02.dbf /fsl/dbs/tbs_09.dbf 9 rows selected.</pre>
Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as shown.	<pre>acctbal NUMBER(11,2); acct CONSTANT NUMBER(4) := 3;</pre>
Italics	Italicized text indicates placeholders or variables for which you must supply particular values.	CONNECT SYSTEM/ <system_password> DB_NAME = database_name</system_password>
UPPERCASE	Uppercase typeface indicates elements supplied by the system. We show these terms in uppercase in order to distinguish them from terms you define. Unless terms appear in brackets, enter them in the order and with the spelling shown. However, because these terms are not case sensitive, you can enter them in lowercase.	<pre>SELECT last_name, employee_id FROM employees; SELECT * FROM USER_TABLES; DROP TABLE hr.employees;</pre>



Convention	Meaning	Example
lowercase	Lowercase typeface indicates programmatic elements that you supply. For example, lowercase indicates names of tables, columns, or files. Note: Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	SELECT last_name, employee_id FROM employees; sqlplus HR/ <i><password></password></i> CREATE USER mjones IDENTIFIED BY ty3MU9

Conventions for Windows Operating Systems

The following table describes conventions for Windows operating systems and provides examples of their use.

Convention	Meaning	Example
Choose Start >	How to start a program.	To start the Database Configuration Assistant, choose Start > Programs > Oracle - <i>HOME_NAME</i> > Configuration and Migration Tools > Database Configuration Assistant.
File and directory names	File and directory names are not case sensitive. The following special characters are not allowed: left angle bracket (<), right angle bracket (>), colon (:), double quotation marks ("), slash (/), pipe (), and dash (-). The special character backslash (\) is treated as an element separator, even when it appears in quotes. If the file name begins with \ then Windows assumes it uses the Universal Naming Convention.	c:\winnt"\"system32 is the same as C:\WINNT\SYSTEM32
C:\>	Represents the Windows command prompt of the current hard disk drive. The escape character in a command prompt is the caret (^). Your prompt reflects the subdirectory in which you are working. Referred to as the <i>command prompt</i> in this manual.	C:\oracle\oradata>
Special characters	The backslash (\) special character is sometimes required as an escape character for the double quotation mark (") special character at the Windows command prompt. Parentheses and the single quotation mark (') do not require an escape character. Refer to your Windows operating system documentation for more information on escape and special characters.	C:\>exp HR/ <password> TABLES=employees QUERY=\"WHERE job_id='SALESMAN' and salary<1600\" C:\>imp SYSTEM/<password> FROM USER=HR TABLES=(employees, dept)</password></password>
HOME_NAME	Represents the Oracle home name. The home name can be up to 16 alphanumeric characters. The only special character allowed in the home name is the underscore.	C:\> net start Oracle <i>HOME_NAME</i> TNSListener



Convention	Meaning	Example
ORACLE_HOME and ORACLE_BASE	In releases prior to Oracle8 <i>i</i> release 8.1.3, when you installed Oracle components, all subdirectories were located under a top level <i>ORACLE_HOME</i> directory that by default used one of the following names:	Go to the ORACLE_BASE\ORACLE_HOME\rdbms\admin directory.
	• C:\orant for Windows NT	
	 C: \orawin98 for Windows 98 This release complies with Optimal Flexible Architecture (OFA) guidelines. All subdirectories are not under a top level ORACLE_HOME directory. There is a top level directory called ORACLE_BASE that by default is C: \oracle. If you install the latest Oracle release on a computer with no other Oracle software installed, then the default setting for the first Oracle home directory is C: \oracle\orann, where nn is the latest release number. The Oracle home directory is located directly under ORACLE_BASE. 	
	All directory path examples in this guide follow OFA conventions.	
	Refer to Oracle Database Platform Guide for Microsoft Windows for additional information about OFA compliance and for information about installing Oracle products in non-OFA compliant directories.	

Changes in This Release for Oracle Database JDBC Developer's Guide

This section contains the changes in this book for Oracle Database Release 23c.

New Features

This section lists the new features for this release:

JDBC Service Provider Extensions

See Also:

JDBC Service Provider Extensions

Enhanced Support for token-based authentication

See Also: Support for Token-Based Authentication for IAM and Support for Token-Based Authentication for Azure AD

Support for RADIUS Challenge-Response Authentication



Kerberos Authentication Enhancements

Support for Kerberos Constrained Delegation

See Also:

Support for Kerberos Constrained Delegation



• Support for Pipelined Database Operations

See Also: Support for Pipelined Database Operations

Support for Data Load Mode in RSI

See Also: About Reactive Streams Ingestion (RSI) Modes

Support for Annotations

See Also: Support for Annotations

• Support for Oracle True Cache

See Also: Support for Oracle True Cache

• Support for the Bequeath Protocol

Support for the Bequeath Protocol

Support for the SQL BOOLEAN Data Type

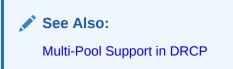
See Also: Class oracle.jdbc.OracleTypes

• Enhanced, Cloud-Ready Diagnosability Features



Support for Multi-Pool DRCP





 Support for Session State Stable (SSS) Cursors with Transparent Application Continuity (TAC)



• Transaction Guard Support during DBMS_ROLLING Operations Transaction Guard support for rolling upgrades, using the DBMS_ROLLING package, ensures that commit outcomes are guaranteed across the entire upgrade process.



Database Development Guide

- Support for Longer Passwords
 Starting with this release, Oracle Database supports passwords up to 1024 bytes
 in length. In previous releases, the Oracle Database password length was up to 30
 bytes. The increased maximum password length provides the following benefits:
 - It accommodates passwords that are used by Oracle Identity Cloud Service (IDCS) and Identity Access Management (IAM). The increase to 1024 bytes enables uniform password rules for all Cloud deployments.
 - The 30-byte limitation was too restrictive when password multi-byte characters used more than 1 byte in an NLS configuration.

The Oracle Database 23c JDBC drivers support this increase in the password length.

• Starting from Oracle Database Release 23c, connection pooling support is implicitly provided to JDBC applications that do not use connection pools. So, even if your application does not close a connection explicitly, Database Resident Connection Pool (DRCP) servers are automatically assigned to and from an application connection at run time, when the application initiates and completes database operations.





Part I Overview

The chapters in this part introduce the concept of Java Database Connectivity (JDBC) and provide an overview of the Oracle implementation of JDBC. This part provides basic information about installation and configuration of the Oracle client with reference to JDBC drivers. This part also covers the basic steps in creating and running any JDBC application.

Part I contains the following chapters:

- Introducing JDBC
- Getting Started



1 Introducing JDBC

Java Database Connectivity (JDBC) is a Java standard that provides the interface for connecting from Java to relational databases. JDBC is based on the X/Open SQL Call Level Interface (CLI). JDBC 4.0 complies with the SQL 2003 standard.

The JDBC standard is defined and implemented through the standard java.sql interfaces. This enables individual providers to implement and extend the standard with their own JDBC drivers. This chapter provides an overview of the Oracle implementation of JDBC, covering the following topics:

- Overview of Oracle JDBC Drivers
- Choosing the Appropriate Driver
- Feature Differences Between JDBC OCI and Thin Drivers
- Environments and Support
- Feature List

1.1 Overview of Oracle JDBC Drivers

In addition to supporting the standard JDBC application programming interfaces (APIs), Oracle drivers have extensions to support Oracle-specific data types and to enhance performance.

Oracle provides the following JDBC drivers:

Thin driver

The JDBC Thin driver is a pure Java, Type IV driver that can be used in applications. It is platform-independent and does not require any additional Oracle software on the client-side. The JDBC Thin driver communicates with the server using Oracle Net Services to access Oracle Database.

The JDBC Thin driver enables a direct connection to the database by providing an implementation of Oracle Net Services on top of Java sockets. The driver supports the TCP/IP protocol and requires a TNS listener on the TCP/IP sockets on the database server.



Oracle recommends you to use the Thin driver unless you have a feature that is supported only by a specific driver.

Oracle Call Interface (OCI) driver

It is used on the client-side with an Oracle client installation. It can be used only with applications.



The JDBC OCI driver is a Type II driver used with Java applications. It requires platform-specific OCI libraries. It supports all installed Oracle Net adapters, including interprocess communication (IPC), named pipes, TCP/IP, and Internetwork Packet Exchange/Sequenced Packet Exchange (IPX/SPX).

The JDBC OCI driver, written in a combination of Java and C, converts JDBC invocations to calls to OCI, using native methods to call C-entry points. These calls communicate with the database using Oracle Net Services.

The JDBC OCI driver uses the OCI libraries, C-entry points, Oracle Net, core libraries, and other necessary files on the client computer where it is installed.

OCI is an API that enables you to create applications that use the native procedures or function calls of a third-generation language to access Oracle Database and control all phases of the SQL statement processing.

• Server-side Thin driver

It is functionally similar to the client-side Thin driver. However, it is used for code that runs on the database server and needs to access another session either on the same server or on a remote server on any tier.

The JDBC server-side Thin driver offers the same functionality as the JDBC Thin driver that runs on the client-side. However, the JDBC server-side Thin driver runs inside Oracle Database and accesses a remote database or a different session on the same database for use with Java in the database.

This driver is useful in the following scenarios:

- Accessing a remote database server from an Oracle Database instance acting as a middle tier
- Accessing an Oracle Database session from inside another, such as from a Java stored procedure

The use of JDBC Thin driver from a client application or from inside a server does not affect the code.

• Server-side internal driver

It is used for code that runs on the database server and accesses the same session. That is, the code runs and accesses data from a single Oracle session.

The JDBC server-side internal driver supports any Java code that runs inside Oracle Database, such as in a Java stored procedure, and accesses the same database. It lets the Oracle Java Virtual Machine (Oracle JVM) to communicate directly with the SQL engine for use with Java in the database.

The JDBC server-side internal driver, the Oracle JVM, the database, and the SQL engine all run within the same address space, and therefore, the issue of network round-trips is irrelevant. The programs access the SQL engine by using function calls.

Note:

The server-side internal driver does not support the cancel and setQueryTimeout methods of the Statement class.



The JDBC server-side internal driver is fully consistent with the client-side drivers and supports the same features and extensions.

The following figure illustrates the architecture of Oracle JDBC drivers and Oracle Database.

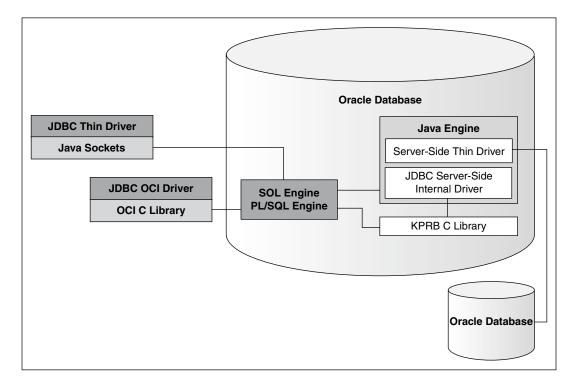


Figure 1-1 Architecture of Oracle JDBC Drivers and Oracle Database

Related Topics

- Features Specific to JDBC Thin
- Features Specific to JDBC OCI Driver
- Server-Side Internal Driver

1.2 Choosing the Appropriate Driver

Consider the following when choosing a JDBC driver for your application or applet:

- In general, unless you need OCI-specific features, such as support for non-TCP/IP networks, use the JDBC Thin driver.
- If you want maximum portability and performance, then use the JDBC Thin driver. You can connect to Oracle Database from an application using the JDBC Thin driver.
- If you want to use Lightweight Directory Access Protocol (LDAP) over Transport Layer Security (TLS), then use the JDBC Thin driver.
- If you are writing a client application for an Oracle client environment and need OCIdriver-specific features, such as support for non-TCP/IP networks, then use the JDBC OCI driver.



- For code that runs in the database server and needs to access a remote database or another session within the same database instance, use the JDBC server-side Thin driver.
- If your code runs inside the database server and needs to access data locally within the session, then use the JDBC server-side internal driver to access that server.

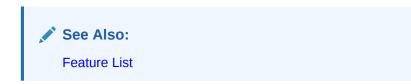
1.3 Use Cases of Oracle JDBC Drivers

This section describes the use cases for Oracle JDBC drivers.

JDBC Thin Driver or Type 4 Client Driver

You must use the JDBC Thin driver in your client-side Java applications for accessing Oracle Database over the TCP/IP protocol, for both tcp and tcps. This is the most widely used driver that Oracle recommends to use because it offers a range of features as mentioned in the following non-exhaustive list:

- Row count per iteration
- Support for promoting a local transaction to a global transaction
- Transaction Guard
- Transparent Application Continuity and Application Continuity
- Support for the Reactive Streams Ingestion library
- JDBC Reactive Extensions



JDBC OCI Driver or Type 2 Client Driver

You must use the JDBC OCI driver in your client-side Java applications only if your applications use any of the following features that are dependent on the platform-specific OCI libraries:

Bequeath protocol

This procol lets you use the local connections without going through the listener, which is typically used by the Database Administrators to perform various administrative operations; however, other non-administrative users too can use this protocol.



Using Bequeath Connection and SYS Logon

OS Authentication

The JDBC OCI driver supports OS Authentication on Linux when the client and the server are on the same computer. On Window domains, it supports OS Authentication even across multiple computers.



• Transparent Application Failover (TAF) that supports failover of read transactions.



Server-Side Internal Driver or Type 2 Server-Side Driver

You must use this driver for Java code that runs on the JVM embedded in the Database server and accesses the same Database session. It means that the code runs and accesses data from a single Database session. This driver is also known as the JDBC KPRB Driver.

This driver is built in the JVM embedded in the Database, also known as Oracle JVM, and it is not a fully-featured driver.

Server-Side Thin Driver or Type 4 Server-Side Driver

You must use this driver in the following scenarios:

- For accessing a remote database server from an Oracle Database instance acting as a middle tier
- For accessing an Oracle Database session from within another session, such as from a Java stored procedure

This driver is built in the Oracle JVM, and it is not a fully-featured driver.

1.4 Feature Differences Between JDBC OCI and Thin Drivers

The following table lists the features that are specific either to the JDBC OCI or the JDBC Thin driver in Oracle Database Release 23c.

JDBC OCI Driver	JDBC Thin Driver
OCI connection pooling	NA
Transparent Application Failover (TAF)	NA
NA	Support for row count per iteration for array DML
NA	SHA-2 Support in Oracle Advanced Security
oraaccess.xml configuration file settings	NA
NA	Oracle Advanced Queuing
NA	Support for the O7L_MR client ability
NA	Support for promoting a local transaction to a global transaction
NA	Java Data Source for Sharded Databases Access
NA	Java Library for Reactive Streams Ingestion
NA	JDBC Reactive Extensions
NA	Native JSON Type Support

Table 1-1Feature Differences Between the JDBC OCI Driver and the JDBC ThinDriver



- The OCI optimized fetch feature is internal to the JDBC OCI driver and not applicable to the JDBC Thin driver.
- Some JDBC OCI driver features, inherited from the OCI library, are not available in the Thin JDBC driver.

1.5 Environments and Support

This section provides a brief description of the tools and environments that you need to run a JDBC application.

- Supported JDK and JDBC Versions
- JNI and Java Environments
- JDBC and IDEs
- Availability on Maven Central

1.5.1 Supported JDK and JDBC Versions

In Oracle Database 23c, all the JDBC drivers are compatible with JDK 8, JDK 11, and JDK 17, and the <code>ojdbc8.jar</code> and <code>ojdbc11.jar</code> files provide the support to these JDK versions.

When to Use ojdbc8.jar File

Use the <code>ojdbc8.jar</code> file when you want JDBC 4.2 features and need to compile your code with JDK 8 and JDK 11.

When to Use ojdbc11.jar File

Use the <code>ojdbc11.jar</code> file when you want JDBC 4.3 features and need to compile your code with JDK 11 and JDK 17.

Related Topics

RDBMS and JDK Version Compatibility for Oracle JDBC Drivers
 Oracle Database Release 23c JDBC drivers are certified with all the supported
 Oracle Database releases (23c, 21c, and 19c).

See Also:

JDBC FAQ Page



1.5.2 JNI and Java Environments

The JDBC OCI driver uses the standard Java Native Interface (JNI) to call OCI C libraries. You can use the JDBC OCI driver with Java Virtual Machines (JVMs), in particular, with Microsoft and IBM JVMs.

1.5.3 JDBC and IDEs

The Oracle JDeveloper Suite provides developers with a single, integrated set of products to build, debug, and deploy component-based database applications for the Internet. The Oracle JDeveloper environment contains integrated support for JDBC, including the JDBC Thin driver and the native OCI driver. The database component of Oracle JDeveloper uses the JDBC drivers to manage the connection between the application running on the client and the server.

1.5.4 Availability on Maven Central

All supported releases of the Oracle JDBC drivers, including 21.1.0.0, 19.9.0.0, 19.8.0.0, 19.6.0.0, 19.3.0.0, and 18.3.0.0, are available on Maven Central. So, you can consider Maven Central as a distribution center for the Oracle JDBC drivers and companion JAR files.

Group IDs for JDBC Drivers and Companion JAR Files on Maven Central

All Oracle Database artifacts on Maven Central reside under the same umbrella com.oracle.database as shown in the following image:

com/oracle/database

../ ha/ jdbc/ nls/ observability/ security/ xml/

You can find the Oracle Database Artifacts under their specific focus area. For example, JDBC, XML, security, high-availability (HA), NLS, observability, and so on. The following table lists the group IDs of the JDBC drivers and the companion JAR files:



Group ID	Corresponding JAR Files
com.oracle.database.jdbc	ojdbc11.jar,ojdbc10.jar,ojdbc8.jar, ojdbc6.jar,ojdbc5.jar,ucp.jar, ojdbc10dms.jar,ojdbc8dms.jar, ojdbc6dms.jar,ojdbc45dms.jar
com.oracle.database.jdbc.debug	ojdbc11_g.jar,ojdbc10_g.jar, ojdbc8_g.jar,ojdbc6_g.jar, ojdbc5_g.jar,ojdbc11dms_g.jar, ojdbc10dms_g.jar,ojdbc8dms_g.jar, ojdbc6dms_g.jar,ojdbc5dms_g.jar
com.oracle.database.security	osdt_core.jar,osdt_cert.jar, oraclepki.jar
com.oracle.database.ha	ons.jar,simplefan.jar
com.oracle.database.nls	orail8n.jar
com.oracle.database.xml	xdb.jar,xdb6.jar,xmlparserv2.jar
com.oracle.database.observability	dms.jar

- The ojdbc8dms.jar and ojdbc11dms.jar files provide complete support for the Dynamic Monitoring System (DMS) and limited support for the java.util.logging package.
- xdb6.jar is a legacy name. The new name is xdb.jar.

Managing Dependencies on Maven Central with GAVs

You can manage the JDBC and UCP dependencies in the pom.xml file of your project by using the corresponding group ID, artifact ID, and the version (GAV), as defined in this section. For example, the following GAV pulls the ojdbc10.jar, ucp.jar, oraclepki.jar, osdt_core.jar, osdt_cert.jar, ons.jar, and simplefan.jar from the 19.3 release:

```
<dependency>
```

```
<groupId>com.oracle.database.jdbc</groupId>
    <artifactId>ojdbc10</artifactId>
    <version>19.3.0.0</version>
</dependency>
```

Similarly, the following GAV pulls the orail8n.jar file from the 19.3.0.0 release:

```
<dependency>
    <groupId>com.oracle.database.nls</groupId>
    <artifactId>orai18n</artifactId>
    <version>19.3.0.0</version>
    </dependency>
```



See Also: JDBC FAQ Page

1.6 Feature List

This section lists the supported features and the corresponding versions in which they were first supported in the JDBC OCI driver and the JDBC Thin driver.

Feature	JDBC OCI	JDBC Thin
TimeZone Patching	11.2	11.2
Secure LOB Support	11.2	11.2
LOB prefetch Support	11.2	11.2
Network Connection Pool	NA	11.2
Column Security Support	NA	11.2
XMLType Queue Support (AQ)	NA	11.2
Notification Grouping (AQ and DCN)	NA	11.2
SimpleFAN	11.2	11.2
Application Continuity	12.2	12.1
Transaction Guard	12.2	12.1
SQL Statement Translation	NA	12.1
Database Resident Connection Pooling	12.1	12.1
SHA-2 Support in Oracle Advanced Security	NA	12.1
Invisible Columns Support	12.1	12.1
Support for PL/SQL Package Types as Parameters	12.1	12.1
Support for Monitoring of Database Operations	12.1	12.1
Support for Increased Length Limit for Various Data Types	12.1	12.1
Implicit Results Support	12.1	12.1
Support for row count per iteration for array DML	NA	12.1
oraaccess.xml configuration file settings	12.1	NA
Transparent Application Continuity	NA	18c
Support for verifying JSON Data	18c	18c
Support for Lightweight Connection Validation	NA	18c
Support for REF CURSOR as IN bind variables	18c	18c
Support for Key Store Service	NA	18c
Easy Connect Plus (Easy Connect Naming Syntax Improvements)	NA	19c
Java Library for Reactive Streams Ingestion	NA	21c
Java Data Source for Sharded Databases Access	NA	21c
JDBC Support for Native JSON Data Type	NA	21c
JDBC Reactive Extensions	NA	21c

Table 1-2Feature List



Table 1-2	(Cont.)	Feature List
-----------	---------	--------------

Feature	JDBC OCI	JDBC Thin
JDBC Pipelining Support	23c	23c

- The following features of JDBC drivers were introduced in releases earlier than release 11.2:
 - NLS Support
 - New Statement Caching API
 - Row Prefetch
 - Java Native Interface
 - Native LOB
 - Associative Arrays/index-by-table
 - Implicit Statement Caching
 - Explicit Statement Caching
 - Temporary LOBs
 - Object Type Inheritance
 - Multilevel Collections
 - oracle.jdbc Interfaces
 - Native XA
 - OCI Connection Pooling
 - Transparent Application Failover
 - Implicit Connection Cache
 - Fast Connection Failover
 - Connection Wrapping
 - DMS
 - Service Names in URLs
 - Set Statement Parameters by Name
 - End-to-End Tracing
 - Web RowSet
 - Proxy Authentication
 - Run-time Connection Load Balancing
 - Extended setXXX and getXXX methods for LOBs
 - XA Connection Cache
 - DML Returning
 - JSR 114 RowSets
 - SSL/TLS Encryption
 - SSL/TLS Authentication
 - AES Encryption
 - SHA1 Hash



- Radius Authentication
- Kerberos Authentication
- ANYDATA and ANYTYPE types
- Native AQ
- Query Change Notification
- Database start up and shut down
- Factory methods for data types
- Buffer Cache
- Secure File LOBs
- Diagnosability
- Server Result Cache
- Universal Connection Pool
- The ConnectionCacheImpl connection cache feature is deprecated since Oracle Database 10g.
- The Implicit Connection Cache feature is desupported now.



2 Getting Started

This chapter discusses the compatibility of Oracle Java Database Connectivity (JDBC) driver versions, database versions, and Java Development Kit (JDK) versions.

It also describes the basics of testing a client installation and configuration and running a simple application. This chapter contains the following sections:

- Version Compatibility for Oracle JDBC Drivers
- Verifying a JDBC Client Installation
- Basic Steps in JDBC
- Sample: Connecting_ Querying_ and Processing the Results
- Support for Annotations
- Support for Oracle True Cache
- Support for the Bequeath Protocol
- Support for Invisible Columns
- Support for Verifying JSON Data
- Support for Implicit Results
- Support for Lightweight Connection Validation
- Support for Deprioritization of Database Nodes
- Support for Oracle Connection Manager in Traffic Director Mode
- Stored Procedure Calls in JDBC Programs
- About Processing SQL Exceptions

2.1 RDBMS and JDK Version Compatibility for Oracle JDBC Drivers

Oracle Database Release 23c JDBC drivers are certified with all the supported Oracle Database releases (23c, 21c, and 19c).

The following table describes the JDBC and Oracle Database interoperability matrix or the certification matrix:

JDBC Driver Version	Database 23.x	Database 21.x	Database 19.x
JDBC 23	Yes	Yes	Yes
JDBC 21.x	Yes	Yes	Yes
JDBC 19.x	Yes	Yes	Yes

Oracle JDBC Drivers are always compliant to the latest JDK version for every new release. For some versions, JDBC drivers support multiple JDK versions. The following table



describes the release-specific JDBC JAR files and supported JDK versions for various Oracle Database versions:



Oracle JDBC Version	Release-Specific JDBC JAR File with Supported JDK Versions
23.x	ojdbc11.jar with JDK 11, JDK 17, JDK 19, and JDK 21 ojdbc8.jar with JDK 8 and JDK 11
21.x	ojdbc11.jar with JDK 11, JDK 17, and JDK 19 ojdbc8.jar with JDK 8 and JDK 11
19.x	ojdbc10.jar with JDK 11 and JDK 17 ojdbc8.jar with JDK 8, JDK 11, JDK 17, and JDK 19

Related Topics

- Oracle Universal Connection Pool Developer's Guide
- Oracle JDBC FAQ

2.2 Verifying a JDBC Client Installation

This section describes the steps that you must perform to verify a JDBC client installation.

- Checking the Environment Variables
- Ensuring that the Java Code Can Be Compiled and Run
- Determining the Version of the JDBC Driver
- Testing the JDBC and Database Connection

Note:

- If you use the JDBC Thin driver, then there is no additional installation on the client computer. If you use the JDBC Oracle Call Interface (OCI) driver, then you must also install the Oracle client software. This includes Oracle Net and the OCI libraries.
- The JDBC Thin driver requires a TCP/IP listener to be running on the computer, where the database is installed.

This section describes the steps for verifying an Oracle client installation of the JDBC drivers, assuming that you have already installed the driver of your choice. Installation



of an Oracle JDBC driver is platform-specific. You must follow the installation instructions for the driver you want to install in your platform-specific documentation.

2.2.1 Checking the Environment Variables

This section describes the environment variables that you must set for the JDBC OCI driver and the JDBC Thin driver, focusing on Solaris, Linux, and Microsoft Windows platforms.

JDBC Thin Driver

You must set the CLASSPATH environment variable for using the JDBC Thin driver. Include the following in the CLASSPATH environment variable:

```
jdbc/lib/ojdbc11.jar
jlib/orai18n.jar
```

Note:

If you use the JTA features and the JNDI features, then you must specify jta.jar and jndi.jar in your CLASSPATH environment variable.

JDBC OCI Driver

You must set the CLASSPATH environment variable for using the JDBC OCI driver. Include the following in the CLASSPATH environment variable:

ORACLE_HOME/jdbc/lib/ojdbc11.jar
ORACLE_HOME/jlib/orai18n.jar

Note:

If you use the JTA features and the JNDI features, then you must specify jta.jar and jndi.jar in your CLASSPATH environment variable.

To use the JDBC OCI driver, you must also set the following value for the library path environment variable:

• On Solaris or Linux, set the LD LIBRARY PATH environment variable as follows:

ORACLE_HOME/lib

This directory contains the libocijdbc11.so shared object library.

• On Microsoft Windows, set the PATH environment variable as follows:

 $\textit{ORACLE_HOME}\bin$

This directory contains the ocijdbc11.dll dynamic link library.

All of the JDBC OCI demonstration programs can be run in the Instant Client mode by including the JDBC OCI Instant Client data shared library on the library path environment variable.



Setting Permission for the Server-Side Thin Driver

To use the JDBC server-side Thin driver, you must set permissions because the JDBC server-side Thin driver opens a socket for its connection to the database. So, Oracle Database enforces the Java security model and performs a check for a SocketPermission object.

The following is an example of how the permission can be granted for the user HR:

```
CREATE ROLE jdbcthin;
CALL dbms_java.grant_permission('JDBCTHIN', 'java.net.SocketPermission', '*',
'connect');
GRANT jdbcthin TO HR;
```

Note that JDBCTHIN in the grant_permission call must be in uppercase. The asterisk (*) is a pattern. You can restrict the user by granting permission to connect to only specific computers or ports.

Related Topics

- Features Specific to JDBC OCI Driver
- Oracle Database Java Developer's Guide

2.2.2 Ensuring that the Java Code Can Be Compiled and Run

To further ensure that Java is set up properly on your client system, go to the samples directory under the <code>ORACLE_HOME/jdbc/demo</code> directory. Now, type the following commands on the command line, one after the other, to see if the Java compiler and the Java interpreter run without error:

javac

java

Each of the preceding commands should display a list of options and parameters and then exit. Ideally, verify that you can compile and run a simple test program, such as jdbc/demo/samples/generic/SelectExample.

2.2.3 Determining the Version of the JDBC Driver

It is very important to use the correct version of the JDBC driver in your applications.

Use the following commands to determine the version of the JDBC driver:

- java -jar ojdbc8.jar
- java -jar ojdbc11.jar

You can also call the getDriverVersion method of the OracleDatabaseMetaData class, as shown in the following sample code:

```
import java.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.OracleDataSource;
class JDBCVersion
{
    public static void main (String args[]) throws SQLException
```



```
{
    OracleDataSource ods = new OracleDataSource();
    ods.setURL("jdbc:oracle:thin:HR/<password>@<host>:<port>:<service>");
    Connection conn = ods.getConnection();
    // Create Oracle DatabaseMetaData object
    DatabaseMetaData meta = conn.getMetaData();
    // gets driver info:
    System.out.println("JDBC driver version is " + meta.getDriverVersion());
}
```

2.2.4 Testing the JDBC and Database Connection

The samples directory contains sample programs for a particular Oracle JDBC driver. One of the programs, JdbcCheckup.java, is designed to test JDBC and the database connection. The program queries for the user name, password, and the name of the database to which you want to connect. The program connects to the database, queries for the string "Hello World", and prints it to the screen.

Go to the samples directory, and compile and run the JdbcCheckup.java program. If the results of the query print without error, then your Java and JDBC installations are correct.

Although JdbcCheckup.java is a simple program, it demonstrates several important functions by performing the following:

- Imports the necessary Java classes, including JDBC classes
- Creates a DataSource instance
- Connects to the database
- Runs a simple query
- Prints the query results to your screen

The JdbcCheckup.java program, which uses the JDBC OCI driver, is as follows:

```
/*
\star This sample can be used to check the JDBC installation.
 * Just run it and provide the connect information. It will select
 * "Hello World" from the database.
 */
// You need to import the java.sql and JDBC packages to use JDBC
import java.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.OracleDataSource;
// We import java.io to be able to read from the command line
import java.io.*;
class JdbcCheckup
{
 public static void main(String args[]) throws SQLException, IOException
  {
    // Prompt the user for connect information
    System.out.println("Please enter information to test connection to
                          the database");
```



```
String user;
 String password;
 String database;
 user = readEntry("user: ");
 int slash index = user.indexOf('/');
 if (slash index != -1)
  {
   password = user.substring(slash index + 1);
   user = user.substring(0, slash index);
  }
 else
   password = readEntry("password: ");
 database = readEntry("database(a TNSNAME entry): ");
 System.out.print("Connecting to the database...");
 System.out.flush();
 System.out.println("Connecting...");
 // Open an OracleDataSource and get a connection
 OracleDataSource ods = new OracleDataSource();
 ods.setURL("jdbc:oracle:oci:@" + database);
 ods.setUser(user);
 ods.setPassword(password);
 Connection conn = ods.getConnection();
  System.out.println("connected.");
  // Create a statement
 Statement stmt = conn.createStatement();
  // Do the SQL "Hello World" thing
 ResultSet rset = stmt.executeQuery("select 'Hello World' from dual");
 while (rset.next())
   System.out.println(rset.getString(1));
  // close the result set, the statement and the connection
 rset.close();
 stmt.close();
 conn.close();
 System.out.println("Your JDBC installation is correct.");
}
// Utility function to read a line from standard input
static String readEntry(String prompt)
{
 try
  {
   StringBuffer buffer = new StringBuffer();
   System.out.print(prompt);
   System.out.flush();
   int c = System.in.read();
   while (c != ' n' \& c != -1)
     buffer.append((char)c);
     c = System.in.read();
    }
   return buffer.toString().trim();
  catch (IOException e)
  {
   return "";
  }
```



2.3 Basic Steps in JDBC

}

After verifying the JDBC client installation, you can start creating your JDBC applications. When using Oracle JDBC drivers, you must include certain driver-specific information in your programs. This section describes, in the form of a tutorial, where and how to add the information. The tutorial guides you through the steps to create code that connects to and queries a database from the client.

You must write code to perform the following tasks:

- 1. Importing Packages
- 2. Opening a Connection to a Database
- 3. Creating a Statement Object
- 4. Running a Query and Retrieving a Result Set Object
- 5. Processing the Result Set Object
- 6. Closing the Result Set and Statement Objects
- 7. Making Changes to the Database
- 8. About Committing Changes
- 9. Closing the Connection

Note:

You must supply Oracle driver-specific information for the first three tasks that enable your program to use the JDBC application programming interface (API) to access a database. For the other tasks, you can use standard JDBC Java code, as you would for any Java application.

2.3.1 Importing Packages

Regardless of which Oracle JDBC driver you use, include the import statements shown in Table 2-1 at the beginning of your program using the following syntax:

import <package_name>;

Table 2-1 Import Statements for JDBC Driver

Import statement	Provides	
<pre>import java.sql.*;</pre>	Standard JDBC packages.	
<pre>import java.math.*;</pre>	The BigDecimal and BigInteger classes. You can omit this package if you are not going to use these classes in your application.	



Import statement	Provides
<pre>import oracle.jdbc.*;</pre>	Oracle extensions to JDBC. This is optional.
<pre>import oracle.jdbc.pool.*;</pre>	OracleDataSource.
<pre>import oracle.sql.*;</pre>	Oracle type extensions. This is optional.

Table 2-1 (Cont.) Import Statements for JDBC Driver

The Oracle packages listed as optional provide access to the extended functionality provided by Oracle JDBC drivers, but are not required for the example presented in this section.

Note:

It is better to import only the classes your application needs, rather than using the wildcard asterisk (*). This guide uses the asterisk (*) for simplicity, but this is not the recommended way of importing classes and interfaces.

2.3.2 Opening a Connection to a Database

First, you must create an OracleDataSource instance. Then, open a connection to the database using the OracleDataSource.getConnection method. The properties of the retrieved connection are derived from the OracleDataSource instance. If you set the URL connection property, then all other properties, including TNSEntryName,

DatabaseName, ServiceName, ServerName, PortNumber, Network Protocol, and driver type are ignored.

Specifying a Database URL, User Name, and Password

The following code sets the URL, user name, and password for a data source:

```
OracleDataSource ods = new OracleDataSource();
ods.setURL(url);
ods.setUser(user);
ods.setPassword(password);
```

The following example connects user HR with password hr to a database with service orcl through port 5221 of the host myhost, using the JDBC Thin driver:

```
OracleDataSource ods = new OracleDataSource();
String url = "jdbc:oracle:thin:@myhost:5221/orcl";
ods.setURL(url);
ods.setUser("HR");
ods.setPassword("hr");
Connection conn = ods.getConnection();
```

Note:

The user name and password specified in the arguments override any user name and password specified in the URL.



Specifying a Database URL that Includes User Name and Password

The following example connects user HR with password hr to a database host whose Transparent Network Substrate (TNS) entry is myTNSEntry, using the JDBC Oracle Call Interface (OCI) driver. In this case, the URL includes the user name and password and is the only input parameter.

```
String url = "jdbc:oracle:oci:HR/<password>@myTNSEntry");
ods.setURL(url);
Connection conn = ods.getConnection();
```

If you want to connect using the Thin driver, then you must specify the port number. For example, if you want to connect to the database on the host myhost that has a TCP/IP listener on port 5221 and the service identifier is orcl, then provide the following code:

```
String URL = "jdbc:oracle:thin:HR/<password>@myhost:5221/orcl");
ods.setURL(URL);
Connection conn = ods.getConnection();
```

Related Topics

- Data Sources and URLs
- Data Sources and URLs

2.3.3 Creating a Statement Object

Once you connect to the database and, in the process, create a Connection object, the next step is to create a Statement object. The createStatement method of the JDBC Connection object returns an object of the JDBC Statement type. To continue the example from the previous section, where the Connection object conn was created, here is an example of how to create the Statement object:

Statement stmt = conn.createStatement();

2.3.4 Running a Query and Retrieving a Result Set Object

To query the database, use the <code>executeQuery</code> method of the <code>Statement</code> object. This method takes a SQL statement as input and returns a JDBC <code>ResultSet</code> object.



- The method used to execute a Statement object depends on the type of SQL statement being executed. If the Statement object represents a SQL query returning a ResultSet object, the executeQuery method should be used. If the SQL is known to be a DDL statement or a DML statement returning an update count, the executeUpdate method should be used. If the type of the SQL statement is not known, the execute method should be used.
- In case of a standard JDBC driver, if the SQL string being executed does not return a ResultSet object, then the executeQuery method throws a SQLException exception. In case of an Oracle JDBC driver, the executeQuery method does not throw a SQLException exception even if the SQL string being executed does not return a ResultSet object.

To continue the example, once you create the Statement object stmt, the next step is to run a query that returns a ResultSet object with the contents of the first_name column of a table of employees named EMPLOYEES:

ResultSet rset = stmt.executeQuery ("SELECT first name FROM employees");

2.3.5 Processing the Result Set Object

Once you run your query, use the next() method of the ResultSet object to iterate through the results. This method steps through the result set row by row, detecting the end of the result set when it is reached.

To pull data out of the result set as you iterate through it, use the appropriate getXXX methods of the ResultSet object, where XXX corresponds to a Java data type.

For example, the following code will iterate through the ResultSet object, rset, from the previous section and will retrieve and print each employee name:

```
while (rset.next())
    System.out.println (rset.getString(1));
```

The next() method returns false when it reaches the end of the result set. The employee names are materialized as Java String values.

2.3.6 Closing the Result Set and Statement Objects

You must explicitly close the ResultSet and Statement objects after you finish using them. This applies to all ResultSet and Statement objects you create when using Oracle JDBC drivers. The drivers do not have finalizer methods. The cleanup routines are performed by the close method of the ResultSet and Statement classes. If you do not explicitly close the ResultSet and Statement objects, serious memory leaks could occur. You could also run out of cursors in the database. Closing both the result set and the statement releases the corresponding cursor in the database. If you close only the result set, then the cursor is not released.



For example, if your ResultSet object is rset and your Statement object is stmt, then close the result set and statement with the following lines of code:

rset.close();
stmt.close();

When you close a Statement object that a given Connection object creates, the connection itself remains open.

Note:

Typically, you should put close statements in a finally clause.

2.3.7 Making Changes to the Database

DML Operations

To perform DML (Data Manipulation Language) operations, such as INSERT or UPDATE operations, you can create either a Statement object or a PreparedStatement object. PreparedStatement objects enable you to run a statement with varying sets of input parameters. The prepareStatement method of the JDBC Connection object lets you define a statement that takes variable bind parameters and returns a JDBC PreparedStatement object with your statement definition.

Use the setXXX methods on the PreparedStatement object to bind data to the prepared statement to be sent to the database.

The following example shows how to use a prepared statement to run INSERT operations that add two rows to the EMPLOYEES table.

```
// Prepare to insert new names in the EMPLOYEES table
PreparedStatement pstmt = null;
trv{
   pstmt = conn.prepareStatement ("insert into EMPLOYEES (EMPLOYEE ID, FIRST NAME)
values (?, ?)");
   // Add LESLIE as employee number 1500
   pstmt.setInt (1, 1500); // The first ? is for EMPLOYEE ID
   pstmt.setString (2, "LESLIE"); // The second ? is for FIRST NAME
   // Do the insertion
   pstmt.execute();
   // Add MARSHA as employee number 507
   pstmt.setInt (1, 507); // The first ? is for EMPLOYEE_ID
   pstmt.setString (2, "MARSHA"); // The second ? is for FIRST_NAME
   // Do the insertion
   pstmt.execute();
}
finally{
       if(pstmt!=null)
   // Close the statement
   pstmt.close();
}
```



DDL Operations

To perform data definition language (DDL) operations, you must create a *Statement* object. The following example shows how to create a table in the database:

```
//create table EMPLOYEES with columns EMPLOYEE_ID and FIRST_NAME
String query;
Statement stmt=null;

try{
    query="create table EMPLOYEES " +
        "(EMPLOYEE_ID int, " +
        "FIRST_NAME varchar(50))";
    stmt = conn.createStatement();
    stmt.executeUpdate(query);
    }
finally{
    //close the Statement object
    stmt.close();
  }
```

Note:

You can also use a PreparedStatement object to perform DDL operations. However, you should not use a PreparedStatement object because the useful part of such an object is that it can have parameters and a DDL operation does not have any parameters.

Also, due to a Database limitation, if you use a PreparedStatement object for a DDL operation, then it only works for the first time it is executed. So, you should use only Statement objects for DDL operations.

The following example shows how to prepare your DDL statements before any reexecution:

```
11
Statement stmt = null;
PreparedStatement pstmt = null;
try{
   pstmt = conn.prepareStatement ("insert into EMPLOYEES (EMPLOYEE ID,
FIRST NAME) values (?, ?)");
   stmt = conn.createStatement("truncate table EMPLOYEES");
   // Add LESLIE as employee number 1500
   pstmt.setInt (1, 1500); // The first ? is for EMPLOYEE ID
   pstmt.setString (2, "LESLIE"); // The second ? is for FIRST NAME
   pstmt.execute();
   stmt.executeUpdate();
   // Add MARSHA as employee number 507
   pstmt.setInt (1, 507); // The first ? is for EMPLOYEE ID
   pstmt.setString (2, "MARSHA"); // The second ? is for FIRST_NAME
   pstmt.execute();
   stmt.executeUpdate();
finally{
```



```
if(pstmt!=null)
    // Close the statement
    pstmt.close();
}
```

Related Topics

- The setObject and setOracleObject Methods
- Other setXXX Methods

2.3.8 About Committing Changes

By default, data manipulation language (DML) operations are committed automatically as soon as they are run. This is known as the auto-commit mode. If auto-commit mode is on and you perform a COMMIT or ROLLBACK operation using the commit or rollback method on a connection object, then you get the following error messages:

Table 2-2 Error Messages for Operations Performed When Auto-Commit Mode is ON

Operation	Error Messages
COMMIT	Could not commit with auto-commit set on
ROLLBACK	Could not rollback with auto-commit set on

If a SQLException is raised during a COMMIT or ROLLBACK operation with the error messages as mentioned in the preceding table, then check the auto-commit status of the connection because you get an exception when these operations are performed on a connection that has auto-commit value set to true.

This exception is raised for any one of the following cases:

- When auto-commit status is set to true and commit or rollback method is called
- When the default status of auto-commit is not changed and commit or rollback method is called
- When the value of the COMMIT_ON_ACCEPT_CHANGES property is true and commit or rollback method is called after calling the acceptChanges method on a rowset

However, you can disable auto-commit mode with the following method call on the Connection object:

conn.setAutoCommit(false);

If you disable the auto-commit mode, then you must manually commit or roll back changes with the appropriate method call on the Connection object:

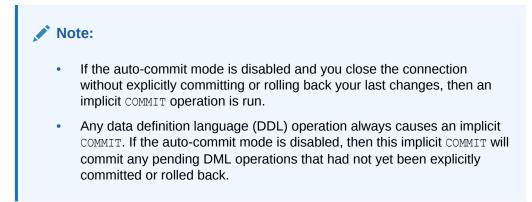
```
conn.commit();
```

or:

conn.rollback();

A COMMIT or ROLLBACK operation affects all DML statements run since the last COMMIT or ROLLBACK.





Related Topics

Disabling Auto-Commit Mode

2.3.8.1 Changing Commit Behavior

When a transaction updates the database, it generates a redo entry corresponding to this update. Oracle Database buffers this redo in memory until the completion of the transaction. When you commit the transaction, the Log Writer (LGWR) process writes the redo entry for the commit to disk, along with the accumulated redo entries of all changes in the transaction. By default, Oracle Database writes the redo to disk before the call returns to the client. This behavior introduces latency in the commit because the application must wait for the redo entry to be persisted on disk.

If your application requires very high transaction throughput and you are willing to trade commit durability for lower commit latency, then you can change the behavior of the default COMMIT operation, depending on the needs of your application. You can change the behavior of the COMMIT operation with the following options:

- WAIT
- NOWAIT
- WRITEBATCH
- WRITEIMMED

These options let you control two different aspects of the commit phase:

- Whether the COMMIT call should wait for the server to process it or not. This is achieved by using the WAIT or NOWAIT option.
- Whether the Log Writer should batch the call or not. This is achieved by using the WRITEIMMED or WRITEBATCH option.

You can also combine different options together. For example, if you want the COMMIT call to return without waiting for the server to process it and also the log writer to process the commits in batch, then you can use the NOWAIT and WRITEBATCH options together. For example:

```
((OracleConnection)conn).commit(
    EnumSet.of(
    OracleConnection.CommitOption.WRITEBATCH,
    OracleConnection.CommitOption.NOWAIT));
```



you cannot use the WAIT and NOWAIT options together because they have opposite meanings. If you do so, then the JDBC driver will throw an exception. The same applies to the WRITEIMMED and WRITEBATCH options.

2.3.9 Closing the Connection

You must close the connection to the database after you have performed all the required operations and no longer require the connection. You can close the connection by using the close method of the Connection object, as follows:

```
conn.close();
```

Note:

Typically, you should put close statements in a finally clause.

2.4 Sample: Connecting, Querying, and Processing the Results

The steps in the preceding sections are illustrated in the following example, which uses the Oracle JDBC Thin driver to create a data source, connects to the database, creates a Statement object, runs a query, and processes the result set.

Note that the code for creating the Statement object, running the query, returning and processing the ResultSet object, and closing the statement and connection uses the standard JDBC API.

```
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.sql.SQLException;
import oracle.jdbc.pool.OracleDataSource;
class JdbcTest
{
   public static void main (String args []) throws SQLException
   {
OracleDataSource ods = null;
Connection conn = null;
Statement stmt = null;
ResultSet rset = null;
      // Create DataSource and connect to the local database
      ods = new OracleDataSource();
      ods.setURL("jdbc:oracle:thin:@localhost:5221/orcl");
      ods.setUser("HR");
      ods.setPassword("hr");
      conn = ods.getConnection();
```



```
{
    // Query the employee names
    stmt = conn.createStatement ();
    rset = stmt.executeQuery ("SELECT first_name FROM employees");
    // Print the name out
    while (rset.next ())
        System.out.println (rset.getString (1));
    }
    //Close the result set, statement, and the connection
finally{
    if(rset!=null) rset.close();
    if(stmt!=null) stmt.close();
    if(conn!=null) conn.close();
    }
    }
}
```

If you want to adapt the code for the OCI driver, then replace the call to the OracleDataSource.setURL method with the following:

ods.setURL("jdbc:oracle:oci:@MyHostString");

where, MyHostString is an entry in the TNSNAMES.ORA file.

2.5 Support for Annotations

Annotations are a lightweight declarative facility for developers to centrally register usage properties for database schema objects.

You can add annotations to schema objects when you create new objects (using the CREATE statement) or modify existing objects (using the ALTER statement). An individual annotation has a name and an optional value, both of which are free-form text fields. A schema object can have multiple annotations.



Starting from Oracle Database Release 23c, the JDBC drivers support annotations. You can use the following methods to work with annotations:

```
oracle.jdbc.AdditionalDatabaseMetaData.getAnnotations
(java.lang.String objectName, java.lang.String domainName,
java.lang.String domainOwner) throws java.sql.SQLException
```

```
oracle.jdbc.OracleResultSetMetaData.getAnnotations
(java.lang.String objectName, java.lang.String columnName,
java.lang.String domainName, java.lang.String domainOwner) throws
java.sql.SQLException
```



These methods return the annotations associated with the specified table or view. If there is no annotation available for the specified object, then it returns null.

2.6 Support for Oracle True Cache

Oracle True Cache is an in-memory, consistent, and automatically managed cache for Oracle Database. It satisfies queries by using data only from its buffer cache.

Note:

This feature is available only to the customers, who are a part of Oracle Database 23c Beta 3 program.

True Cache is a fully functional, read-only replication of the primary database, which is mostly diskless. It exploits the fact that the applications rarely need the most current data, and can use the cached data instead. Queries that use the cached data, can be issued to a True Cache instance that is in the middle tier. So, the applications need to maintain two data sources, one to the primary database and the other to the True Cache instance.

See Also:

Oracle Database Oracle True Cache User's Guide

When configured, the JDBC driver can execute queries on both the True Cache instance and the primary database. Applications maintain only one logical connection, which is aware of both the primary database and the True Cache instance. A query is executed on the True Cache instance if the JDBC driver logical connection is in a read-only mode, otherwise, the query is executed on the primary database. This improves the scalability and the application-response-time because the number of queries sent to the primary database is reduced.

For enabling the True Cache functionality, you must set the value of the new oracle.jdbc.useTrueCacheDriverConnection property to true. Once you enable the True Cache functionality, the JDBC driver uses the standard

<code>java.sql.Connection.setReadOnly(boolean)</code> and <code>java.sql.Connection.isReadOnly()</code> methods to mark a connection as read-only. By default, the read-only mode is <code>false</code> for a connection.

The following code snippet shows how to use the True Cache functionality:

```
...
OracleDataSource ods = new oracle.jdbc.pool.OracleDataSource();
ods.setURL(DB_URL);
ods.setUser(DB_USER);
ods.setPassword(DB_PASSWORD);
Properties props = new Properties();
props.setProperty("oracle.jdbc.useTrueCacheDriverConnection", "true");
ods.setConnectionProperties(props);
// this is a True Cache driver connection and it can be used to execute
// queries on both the primary database and a True Cache instance
Connection conn = ods.getConnection();
Statement stmt = conn.createStatement();
```



```
// Default value of connection read-only flag is false which means
// the SQL_QUERY1 is executed on the primary database
ResultSet rs = stmt.executeQuery(SQL_QUERY1);
// set the read-only flag to true, in order to execute SQL_QUERY2
// on a True Cache instance
conn.setReadOnly(true);
ResultSet rs1 = stmt.executeQuery(SQL_QUERY2);
....
```

🖍 See Also:

- setReadOnly(boolean) method
- isReadOnly method

2.7 Support for the Bequeath Protocol

Starting from Oracle Database Release 23c, the JDBC thin driver supports the Bequeath protocol (BEQ) for applications running on Linux platforms.

To connect to the Database using the Bequeath Protocol, you must set the value of the ORACLE_HOME variable, so that the driver can locate the Oracle server process executable. Typically, the ORACLE_HOME variable points to the database installation location, that is, /var/lib/oracle/dbhome. You can reset the location in the following two ways:

- In the connection URL
- In the environment of the current application

The second mandatory variable, which you must enable, is the <code>ORACLE_SID</code>. Similar to setting the <code>ORACLE_HOME</code> variable, you can set the <code>ORACLE_SID</code> in the connection URL or in the current application environment. To establish a bequeath connection, the BEQ protocol must be enabled, which is the default setting in the authentication services property.

The following example shows how you can set the <code>ORACLE_HOME</code> variable and the <code>ORACLE_SID</code> in the connection URL:

jdbc:oracle:thin:@(DESCRIPTION=(LOCAL=YES)(ADDRESS=(PROTOCOL=beq)) (ENVS=ORACLE HOME=/var/lib/oracle/dbhome,ORACLE SID=oraclesid))

See Also: Oracle JDBC Java API Reference



2.8 Support for Invisible Columns

Starting from this release, Oracle Database supports invisible columns. Using this feature, you can add a column to the table in hidden mode and make it visible later. JDBC provides APIs to retrieve information about invisible columns. To get information about whether a column is invisible or not, you can use the <code>isColumnInvisible</code> method available in the <code>oracle.jdbc.OracleResultSetMetaData</code> interface in the following way:

Example

```
Connection conn = DriverManager.getConnection(jdbcURL, user, password);
Statement stmt = conn.createStatement ();
stmt.executeQuery ("create table hiddenColsTable (a varchar(20), b int invisible)");
stmt.executeUpdate("insert into hiddenColsTable (a,b ) values('somedata',1)");
stmt.executeUpdate("insert into hiddenColsTable (a,b) values('newdata',2)");
System.out.println ("Invisible columns information");
try
{
      ResultSet rset = stmt.executeQuery("SELECT a, b FROM hiddenColsTable");
      OracleResultSetMetaData rsmd = (OracleResultSetMetaData)rset.getMetaData();
      while (rset.next())
        System.out.println("column1 value:" + rset.getString(1));
        System.out.println("Visibility:" + rsmd.isColumnInvisible(1));
        System.out.println("column2 value:" + rset.getInt(2));
        System.out.println("Visibility:" + rsmd.isColumnInvisible(2));
      }
}
catch (Exception ex)
{
       System.out.println("Exception :" + ex);
       ex.printStackTrace();
}
```

Alternatively, you can also use the getColumns method available in the oracle.jdbc.OracleDatabaseMetaData class to retrieve information about invisible columns.

Example

```
Connection conn = DriverManager.getConnection(jdbcURL, user, password);
Statement stmt = conn.createStatement ();
stmt.executeQuery ("create table hiddenColsTable (a varchar(20), b int invisible)");
stmt.executeUpdate("insert into hiddenColsTable (a,b) values('somedata',1)");
stmt.executeUpdate("insert into hiddenColsTable (a,b) values('newdata',2)");
System.out.println ("getColumns for table with invisible columns");
try
{
    DatabaseMetaData dbmd = conn.getMetaData();
    ResultSet rs = dbmd.getColumns(null, "HR", "hiddenColsTable", null);
    OracleResultSetMetaData rsmd = (OracleResultSetMetaData)rs.getMetaData();
    int colCount = rsmd.getColumnCount();
    System.out.println("colCount: " + colCount);
    String[] columnNames = new String [colCount];
```



```
for (int i = 0; i < colCount; ++i)</pre>
      {
        columnNames[i] = rsmd.getColumnName (i + 1);
      }
     while (rs.next())
      {
        for (int i = 0; i < colCount; ++i)</pre>
          System.out.println(columnNames[i] +":" +rs.getString
(columnNames[i]));
     }
}
catch (Exception ex)
{
     System.out.println("Exception: " + ex);
     ex.printStackTrace();
}
```

The server-side internal driver, kprb does not support fetching information about invisible columns.

2.9 Support for Verifying JSON Data

Starting from Oracle Database Release 18c, JDBC drivers can verify whether a column returned in the ResultSet is a JSON column or not. To get information about whether a column is JSON or not, you can use the isColumnJSON method available in the oracle.jdbc.OracleResultSetMetaData interface in the following way:

Example 2-1 Example

```
. . .
 public void test (Connection conn)
      throws Exception{
    try {
      show ("tkpjb26776242 - start");
      createTable(conn);
      String sql = "SELECT col1, col2, col3, col4, col5, col6, col7,
col8 FROM tkpjb26776242 tab";
      Statement stmt = conn.createStatement();
      ResultSet rs = stmt.executeQuery(sql);
      ResultSetMetaData rsmd = rs.getMetaData();
      OracleResultSetMetaData orsmd = (OracleResultSetMetaData) rsmd;
      int colCnt = orsmd.getColumnCount();
      show("Table has " + colCnt + " columns.");
      for (int i = 1; i \leq colCnt; i++) {
        String columnName = orsmd.getColumnName(i);
```



```
String typeName = orsmd.getColumnTypeName(i);
       boolean invisible = orsmd.isColumnInvisible(i);
       boolean json = orsmd.isColumnJSON(i);
       show(columnName + " " + typeName + (invisible?" INVISIBLE":"") +
(json?"
         JSON":""));
      }
     rs.close();
     stmt.close();
     show ("tkpjb26776242 - end");
   }
   finally {
     dropTable(conn);
   }
 }
 private void createTable(Connection conn) throws Exception{
   String sql = " create table tkpjb26776242 tab ( "
                 + " coll clob, "
                + " col2 clob , "
                 + " col3 clob INVISIBLE, "
                 + " col4 clob INVISIBLE, "
                 + " col5 varchar2(200), "
                 + " col6 varchar2(200), "
                 + " col7 varchar2(200) INVISIBLE, "
                 + " col8 varchar2(200) INVISIBLE, "
                 + " check (col2 IS JSON), "
                 + " check (col4 IS JSON), "
                 + " check (col6 IS JSON), "
                 + " check (col8 IS JSON))";
   Util.doSQL(conn, sql);
 }
 private void dropTable (Connection conn) throws Exception {
   String sql = " drop table tkpjb26776242 tab";
   Util.trySQL(conn, sql);
 }
    . . .
```

2.10 Support for Implicit Results

Oracle Database supports results of SQL statements executed in a stored procedure to be returned implicitly to the client applications without the need to explicitly use a REF CURSOR.

You can use the following methods to retrieve and process the implicit results returned by PL/SQL procedures or blocks:

Method	Description	
getMoreResults	Checks if there are more results available in the result set	



Method	Description	
getMoreResults(int)	Checks if there are more results available in the result set, like the overloaded method. This method accepts an int parameter that can have one of the following values:	
	• KEEP_CURRENT_RESULT	
	• CLOSE_ALL_RESULTS	
	CLOSE_CURRENT_RESULT	
getResultSet	Iteratively retrieves each implicit result from an executed PL/SQL statement	

- The server-side internal driver, kprb does not support fetching information about implicit results.
- Only SELECT queries can be returned implicitly.
- Applications retrieve each result set sequentially, but can fetch rows from any result set independent of the sequence.

Suppose you have a procedure called foo as the following:

```
create procedure foo as
  c1 sys_refcursor;
  c2 sys_refcursor;
begin
  open c1 for select * from hr.employees;
  dbms_sql.return_result(c1); --return to client
  -- open 1 more cursor
  open c2 for select * from hr.departments;
  dbms_sql.return_result (c2); --return to client
end;
```

The following code snippet demonstrates how to retrieve the implicit results returned by PL/SQL procedures using the getMoreResults methods:

Example 1

```
String sql = "begin foo; end;";
...
Connection conn = DriverManager.getConnection(jdbcURL, user, password);
try {
    Statement stmt = conn.createStatement ();
    stmt.executeQuery (sql);
    while (stmt.getMoreResults())
    {
        ResultSet rs = stmt.getResultSet();
        System.out.println("ResultSet");
        while (rs.next())
        {
            /* get results */
        }
```



}

Suppose you have another procedure called foo as the following:

```
create or replace procedure foo asc1 sys_refcursor; c2 sys_refcursor; c3
sys_refcursor; begin open c1 for 'select * from hr.employees';
dbms_sql.return_result (c1);-- cursor 2open c2 for 'select * from hr.departments';
dbms_sql.return_result (c2);-- cursor 3open c3 for 'select first_name from
hr.employees';
dbms_sql.return_result (c3); end;
```

The following code snippet demonstrates how to retrieve the implicit results returned by PL/SQL procedures using the getMoreResults(int) methods:

Example 2

```
String sql = "begin foo; end;";
. . .
Connection conn = DriverManager.getConnection(jdbcURL, user, password);
try {
        Statement stmt = conn.createStatement ();
        stmt.executeQuery (sql);
        ResultSet rs = null;
        boolean retval = stmt.getMoreResults(Statement.KEEP CURRENT RESULT))
        if (retval)
        {
            rs = stmt.getResultSet();
            System.out.println("ResultSet");
            while (rs.next())
            {
                /* get results */
            }
        }
        /* closes open results */
        retval = stmt.getMoreResults(Statement.CLOSE ALL RESULTS);
        if (retval)
        {
            System.out.println("More ResultSet available");
            rs = stmt.getResultSet();
            System.out.println("ResultSet");
            while (rs.next())
                /* get results */
            }
        }
        /* close current result set */
        retval = stmt.getMoreResults(Statement.CLOSE CURRENT RESULT);
        if(retval)
        {
            System.out.println("More ResultSet available");
            rs = stmt.getResultSet();
            while (rs.next())
            {
                /* get Results */
```

2.11 Support for Lightweight Connection Validation

Starting from Oracle Database Release 18c, JDBC Thin driver supports lightweight connection validation. Lightweight connection validation enables JDBC applications to verify connection validity by sending a zero length NS data packet that does not require a round-trip to the database.

For the releases of Oracle Database earlier than 18c, when you call the <code>isValid(timeout)</code> method to test the validity of a connection, Oracle JDBC driver uses a ping-pong protocol, which is an expensive operation as it makes a full round-trip to the database. Since Oracle Database Release 18c, the <code>isValid(timeout)</code> method instead sends an empty packet to the database and does not wait to receive it back. So, connection validation is faster, which results in better application performance.

Lightweight connection validation is disabled by default. To enable this feature, you must set the oracle.jdbc.defaultConnectionValidation connection property value to SOCKET. If this property is set, then the JDBC driver performs lightweight connection validation, when you call the isValid(timeout) method.

Note:

}

- Lightweight connection validation checks only the underlying socket health. When the isValid(timeout) method returns true, that is, if a connection is termed as valid, this validation only guarantees that the server is not unreachable (dead socket). It does not provide any status about the server processes, like whether they are running or not. However, by default, that is, when lightweight connection validation is not enabled, the isValid(timeout) method does check whether the network between the client and the server is intact or not.
- Only the JDBC Thin driver supports this feature.

New APIs for Lightweight Connection Validation

• oracle.jdbc.defaultConnectionValidation

This connection property specifies the level of connection validation. The possible values for this property are: NONE, LOCAL, SOCKET, NETWORK, SERVER, and COMPLETE. These values are case-sensitive, and setting any value other than these values throws an exception. The default value is NETWORK.

 public boolean isValid(ConnectionValidation validation_level, int timeout) throws SQLException

The new variation of the existing isValid(timeout) method accepts two parameters: level of validation (validation_level) and timeout. The first parameter specifies the level of connection validation.



Example 2-2 Example of Lightweight Connection Validation

The following code snippet demonstrates how to implement lightweight connection mechanism:

```
. . .
    OracleDataSource ods = new OracleDataSource();
    ods.setURL(url);
    ods.setUser(user);
    ods.setPassword(password);
    Connection conn = ods.getConnection();
    trv{
          boolean isValid = ((OracleConnection)conn).
isValid(ConnectionValidation.SOCKET,timeout);
          System.out.println("Connection isValid = "+isValid);
    }
    catch (Exception ex)
   {
         System.out.println("Exception :" + ex);
         ex.printStackTrace();
    }
    . . .
          . . .
```

2.12 Support for Deprioritization of Database Nodes

Starting from Oracle Database 12c Release 2 (12.2.0.1), JDBC drivers support deprioritization of database nodes. When a node fails, JDBC deprioritizes it for the next 10 minutes, which is the default expiry time. For example, if there are three nodes A, B, C, and node A is down, then connections are allocated first from nodes B and C, and then from node A. After the default expiry time, node A is no longer deprioritized, that is, connections are allocated from all the three nodes on availability basis. Also, during the default expiry time, if a connection attempt to node A succeeds, then node A is no longer considered to be a deprioritized node. You can specify the default expiry time for deprioritization using the oracle.net.DOWN HOSTS TIMEOUT system property.

For example, in the following URL, scan_listener0 has ip1, ip2, and ip3 IP addresses configured, after retrieving its IP addresses. Now, if ip1 is deprioritized, then the order of trying IP addresses will be ip2, ip3, and then ip1. If all IP addresses are unavailable, then the whole host is tried last, after trying node 1 and node 2.

```
(DESCRIPTION=
    (ADDRESS=(PROTOCOL=tcp)(HOST=node_0)(PORT=1528))
    (CONNECT_DATA=(SERVICE_NAME=cdb3))
)
)
```

2.13 Support for Oracle Connection Manager in Traffic Director Mode

Starting from Oracle Database Release 18c, JDBC Drivers support Oracle Connection Manager in Traffic Director Mode, which is a proxy server that resides between the Database clients and the Database instances.

A JDBC client connects to the Oracle Connection Manager in Traffic Director Mode, which in turn connects to the target Oracle Database. The client sends requests in the form of Two-Task Common (TTC) messages that Oracle Connection Manager in Traffic Director Mode intercepts, parses, and then relays to the appropriate target database. Once the responses arrive from the database, Oracle Connection Manager in Traffic Director Mode transfers the responses back to the clients through TTC messages.

The following image illustrates the architecture of Oracle Connection Manager in Traffic Director Mode:

Figure 2-1 Architecture of Oracle Connection Manager in Traffic Director Mode





See Also:

- Oracle Database Net Services Administrator's Guide for more information about configuring the cman.ora file to set up Oracle Connection Manager in Traffic Director Mode
- Oracle Database Net Services Reference for more information about Oracle Connection Manager in Traffic Director Mode parameters

This chapter describes the following concepts:

- Modes of Running Oracle Connection Manager in Traffic Director Mode
- Benefits of Oracle Connection Manager in Traffic Director Mode

2.13.1 Modes of Running Oracle Connection Manager in Traffic Director Mode

You can run Oracle Connection Manager in Traffic Director Mode in the following connection modes:

Pooled connection mode

The pooled connection mode uses a new feature called Proxy Resident Connection Pooling, which is a proxy-enabled mode of Database Resident Connection Pooling (DRCP). The Proxy Resident Connection Pooling reduces the connection load on the database as it multiplexes a large number of client connections over a fewer number of database connections. Any application using Oracle Database 12c Release 1 (12.1) JDBC drivers and later can use this connection mode.

Note:

The pooled connection mode yields best results when you use it with clients using DRCP-aware connection pools.

Nonpooled or dedicated connection mode

You can use the nonpooled or dedicated connection mode with applications using any supported Oracle Database JDBC driver. However, some capabilities, such as connection multiplexing, are not available in this mode.

Related Topics

Overview of Database Resident Connection Pooling

See Also:

- Database Admin Guide
- Universal Connection Pool Developer's Guide



2.13.2 Benefits of Oracle Connection Manager in Traffic Director Mode

This section describes how Oracle Connection Manager in Traffic Director Mode provides benefits to your applications.

- **Transparent performance enhancements:** Oracle Connection Manager in Traffic Director Mode auto enables statement caching, row prefetching, and result set caching during both pooled and nonpooled modes.
- **Connection multiplexing:** Using Proxy Resident Connection Pooling (PRCP), Oracle Connection Manager in Traffic Director Mode (for the pooled mode) provides transparent connection-time load balancing and run-time load balancing with the database. If you use multiple instances of Oracle Connection Manager in Traffic Director Mode, then you can increase the scalability of your application through the implementation of client-side connection-time load balancing or a load balancer like BIG-IP or NGINX.
- Zero application downtime: Oracle Connection Manager in Traffic Director Mode provides zero application downtime during planned database maintenance as well as unplanned database outages. For Unplanned database outages, it offers zero application downtime for read-mostly workloads. For planned database maintenance or pluggable database (PDB) relocation, it uses different techniques for the pooled mode and the nonpooled mode, as described in this section.
 - Pooled mode:

For planned outages, Oracle Connection Manager in Traffic Director Mode responds to the Oracle Notification Service (ONS) events. It uses database connections from the proxy resident connection pool and redirects the requests to the appropriate databases. When the requests complete, it drains the connections from the pool.

For PDB relocations, Oracle Connection Manager in Traffic Director Mode uses an in-band client notification mechanism, which works even when ONS is not configured. This feature is available only for Oracle Database release 18c and later.

Nonpooled or dedicated mode

In this mode, Oracle Connection Manager in Traffic Director Mode leverages either of the following features of Oracle Database:

- * Continuous application availability to stop the service at the request boundary
- * Transparent Application Failover (TAF) to reconnect and restore simple states
- **High Availability:** Oracle Connection Manager in Traffic Director Mode implements the following techniques to avoid single point of failure, and in turn, assures high availability:
 - Multiple instances of Oracle Connection Manager in Traffic Director Mode use a load balancer or client-side load balancing in the connection string.
 - Oracle Connection Manager in Traffic Director Mode instances support rolling upgrade.
 - For planned outages, Oracle Connection Manager in Traffic Director Mode implements graceful close of existing connections from client.



- Oracle Connection Manager in Traffic Director Mode sends in-band notifications to Oracle Database release 18c and later clients, and ONS notifications to all supported clients prior to Oracle Database release 18c.
- **Security:** Oracle Connection Manager in Traffic Director Mode provides security to your applications in the following ways:
 - It supports the Transmission Control Protocol Secure (TCPS) protocol.
 - It creates a firewall based on the IP address, service name, and Transport Layer Security (TLS) wallets.
 - It provides protection against denial-of-service and fuzzing attacks.
 - It provides secure tunneling of database traffic across Oracle Database on-premises and Oracle Cloud.
- **Tenant isolation:** Oracle Connection Manager in Traffic Director Mode provides tenant isolation for increased memory and enhanced processing power.

2.14 Stored Procedure Calls in JDBC Programs

This section describes how Oracle JDBC drivers support the following kinds of stored procedures:

- PL/SQL Stored Procedures
- Java Stored Procedures

2.14.1 PL/SQL Stored Procedures

JDBC supports the invocation of PL/SQL procedures/functions and anonymous blocks, using either JDBC escape syntax or PL/SQL block syntax. The following PL/SQL calls would work with any Oracle JDBC driver:

As an example of using the Oracle syntax, here is a PL/SQL code snippet that creates a stored function. The PL/SQL function gets a character sequence and concatenates a suffix to it:

```
create or replace function foo (val1 char)
return char as
begin
   return val1 || 'suffix';
end;
```

The function invocation in your JDBC program should look like the following:

```
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:oracle:oci:@<hoststring>");
ods.setUser("HR");
```



```
ods.setPassword("hr");
Connection conn = ods.getConnection();
CallableStatement cs = conn.prepareCall ("begin ? := foo(?); end;");
cs.registerOutParameter(1,Types.CHAR);
cs.setString(2, "aa");
cs.execute();
String result = cs.getString(1);
```

2.14.2 Java Stored Procedures

You can use JDBC to call Java stored procedures through the SQL interface. The syntax for calling Java stored procedures is the same as the syntax for calling PL/SQL stored procedures, presuming they have been properly published. That is, you have written call specifications to publish them to the Oracle data dictionary. Applications can call Java stored procedures using the Native Java Interface for direct invocation of static Java methods.

2.15 About Processing SQL Exceptions

To handle error conditions, Oracle JDBC drivers throw SQL exceptions, producing instances of the <code>java.sql.SQLException</code> class or its subclass. Errors can originate either in the JDBC driver or in the database itself. Resulting messages describe the error and identify the method that threw the error. Additional run-time information can also be appended.

JDBC 3.0 defines only a single exception, SQLException. However, there are large categories of errors and it is useful to distinguish them. Therefore, in JDBC 4.0, a set of subclasses of the SQLException exception is introduced to identify the different categories of errors.

Basic exception handling can include retrieving the error message, retrieving the error code, retrieving the SQL state, and printing the stack trace. The SQLException class includes functionality to retrieve all of this information, when available.

Retrieving Error Information

You can retrieve basic error information with the following methods of the SQLException class:

- getMessage class includes functionality to retrieve all of this information, when available.
- getErrorCode class includes functionality to retrieve all of this information, when available.
- getSQLState class includes functionality to retrieve all of this information, when available.

The following example prints output from a getMessage method call:

```
catch(SQLException e)
{
   System.out.println("exception: " + e.getMessage());
}
```

This would print the output, such as the following, for an error originating in the JDBC driver:



```
exception: Invalid column type
```

Note: Error message text is available in alternative languages and character sets supported by Oracle.

Printing the Stack Trace

The SQLException class provides the printStackTrace() method for printing a stack trace. This method prints the stack trace of the Throwable object to the standard error stream. You can also specify a java.io.PrintStream object or java.io.PrintWriter object for output.

The following code fragment illustrates how you can catch SQL exceptions and print the stack trace.

```
try { <some code> }
catch(SQLException e) { e.printStackTrace (); }
```

To illustrate how the JDBC drivers handle errors, assume the following code uses an incorrect column index:

```
// Iterate through the result and print the employee names
// of the code
try {
  while (rset.next ())
     System.out.println (rset.getString (5)); // incorrect column index
}
catch(SQLException e) { e.printStackTrace (); }
```

Assuming the column index is incorrect, running the program would produce the following error text:

```
java.sql.SQLException: Invalid column index
at oracle.jdbc.OracleDriver.OracleResultSetImpl.getDate(OracleResultSetImpl.java:1556)
at Employee.main(Employee.java:41)
```

Related Topics

- JDBC Error Messages
- Oracle Database Error Messages Reference



Part II Oracle JDBC

This part includes chapters that discuss the different Java Database Connectivity (JDBC) versions that Oracle Database 23c supports. It also includes chapters that cover features specific to JDBC Thin driver, JDBC Oracle Call Interface (OCI) driver, and the server-side internal driver.

Part II contains the following chapters:

- JDBC Standards Support
- Oracle Extensions
- Features Specific to JDBC Thin
- Features Specific to JDBC OCI Driver
- Server-Side Internal Driver



3 JDBC Standards Support

Oracle Java Database Connectivity (JDBC) drivers support different versions of the JDBC standard features. These features are provided through the oracle.jdbc and oracle.sql packages. These packages support JDK 8, JDK 11, and JDK 17.

This chapter discusses the JDBC standards support in Oracle JDBC drivers for the most recent releases. It contains the following sections:

- Support for JDBC 4.2 Standard
- Support for JDBC 4.3 Standard

3.1 Support for JDBC 4.2 Standard

Oracle Database Release 23c JDBC drivers provide support for JDBC 4.2 standard through JDK 8. This section lists some of the important methods available in this release.

The %Large% Methods

This release of Oracle JDBC drivers support the following methods introduced in JDBC 4.2 standard, which deal with long values:

- executeLargeBatch()
- executeLargeUpdate(String sql)
- executeLargeUpdate(String sql, int autoGeneratedKeys)
- executeLargeUpdate(String sql, int[] columnIndexes)
- executeLargeUpdate(String sql, String[] columnNames)
- getLargeMaxRows()
- getLargeUpdateCount()
- setLargeMaxRows(long max)

These new methods are available as part of the java.sql.Statement interface. The %Large% methods are identical to the corresponding *non-large* methods, except that they work with long values instead of int values. For example, the executeUpdate method returns the number of rows updated as an int value, whereas, the executeLargeUpdate method returns the number of rows updated as a long value. If the number of rows is greater than the value of Integer.MAX_VALUE, then your application must use the executeLargeUpdate method.

The following code snippet shows how to use the executeLargeUpdate (String sql) method:

```
Statement stmt = conn.createStatement();
stmt.executeQuery("create table BloggersData (FIRST_NAME varchar(100), ID
int)");
long updateCount = stmt.executeLargeUpdate("insert into BloggersData
```



```
(FIRST_NAME,ID) values('John',1)");
...
```

The SQLType Methods

This release of Oracle JDBC drivers support the following methods introduced in JDBC 4.2 standard, which take SQLType parameters:

• setObject

The setObject method sets the value of the designated parameter for the specified object. This method is similar to the setObject(int parameterIndex, Object x, SQLType targetSqlType, int scaleOrLength) method, except that it assumes a scale of zero. The default implementation of this method throws SQLFeatureNotSupportedException.

```
void setObject(int parameterIndex, java.lang.Object x, SQLType
targetSqlType) throws SQLException
```

updateObject

The updateObject method takes the column index as a parameter and updates the designated column with an Object value.

registerOutParameter

The registerOutParameter method registers a specified parameter to be of JDBC type SQLType.

The following code snippet shows how to use the setObject method:

```
int empId = 100;
connection.prepareStatement("SELECT FIRST_NAME, LAST_NAME FROM
EMPLOYEES WHERE EMPNO = ?");
preparedStatement.setObject(1, Integer.valueOf(empId),
OracleType.NUMBER);
```

See Also: JDBC 4.2 Documentation

3.2 Support for JDBC 4.3 Standard

Starting from Release 19c, Oracle Database JDBC drivers provide support for Standard JDBC 4.3 features through JDK 11 and JDK 17. This section describes some of the important APIs available in this release.

Sharding Support

Sharding is a data tier architecture, where data is horizontally partitioned across independent databases. Each database in such a configuration is called a shard. All



shards together make up a single logical database, which is referred to as a sharded database (SDB). You can use the DatabaseMetaData.supportsSharding method to determine whether a JDBC Driver supports sharding or not.

Sharding support includes addition of the following APIs:

- javax.sql.XAConnectionBuilder Interface
 This is a builder interface created from a XADataSource object, which you can use to establish a connection to the database that the data source object represents.
- java.sql.ShardingKey Interface This interface is used to indicate that the current object represents a Sharding Key. You can create a ShardingKey instance using the ShardingKeyBuilder interface.
- java.sql.ShardingKeyBuilder Interface
 This is a builder interface created from a DataSource or XADataSource object, used to create a ShardingKey with subkeys of supported data types.

See Also: Overview of Database Sharding for JDBC Users

Enhancements to the java.sql.Connection Interface

The following methods have been added to the java.sql.Connection Interface:

- default void beginRequest throws SQLException
- default void endRequest throws SQLException
- default void setShardingKey(ShardingKey shardingKey) throws SQLException
- default void setShardingKey(ShardingKey shardingKey, ShardingKey superShardingKey) throws SQLException
- default void setShardingKeyIfValid(ShardingKey shardingKey, int timeout) throws SQLException
- default void setShardingKeyIfValid(ShardingKey shardingKey, ShardingKey superShardingKey, int timeout) throws SQLException

Enhancements to the java.sql.DatabaseMetaData Interface

The following methods have been added to the java.sql.DatabaseMetaData Interface:

default boolean supportsSharding() throws SQLException

Enhancements to the java.sql.Statement Interface

The following methods have been added to the java.sql.Statement Interface:

- default String enquoteIdentifier(String identifier, Boolean alwaysQuote) throws SQLException
- default String enquoteLiteral(String val) throws SQLException
- default String enquoteNCharLiteral(String val) throws SQLException



• default boolean isSimpleIdentifier(String identifier) throws SQLException

See Also: Oracle Database JDBC Java API Reference

4 Oracle Extensions

Oracle provides Java classes and interfaces that extend the Java Database Connectivity (JDBC) standard implementation, enabling you to access and manipulate Oracle data types and use Oracle performance extensions. This chapter provides an overview of the classes and interfaces provided by Oracle that extend the JDBC standard implementation. It also describes some of the key support features of the extensions.

This chapter contains the following sections:

- Overview of Oracle Extensions
- Features of the Oracle Extensions
- Oracle JDBC Packages
- Oracle Character Data Types Support
- Additional Oracle Type Extensions
- DML Returning
- Accessing PL/SQL Associative Arrays

Related Topics

• Performance Extensions

4.1 Overview of Oracle Extensions

Beyond standard features, Oracle JDBC drivers provide Oracle-specific type extensions and performance extensions. These extensions are provided through the following Java packages:

• oracle.sql

Provides classes that represent SQL data in Oracle format

oracle.jdbc

Provides interfaces to support database access and updates in Oracle type formats

Related Topics

Oracle JDBC Packages

4.2 Features of the Oracle Extensions

The Oracle extensions to JDBC include a number of features that enhance your ability to work with Oracle Databases. These include the following:

- Support for Pipelined Database Operations
- Database Management Using JDBC
- Support for Oracle Data Types



- Support for Oracle Objects
- Support for Schema Naming
- DML Returning
- About Accessing PL/SQL Associative Arrays

4.2.1 Database Management Using JDBC

Starting from Oracle Database 11g Release 1, the oracle.jdbc.OracleConnection interface has two JDBC methods, startup and shutdown, which enable you to start up and shut down an Oracle Database instance.

Note:

My Oracle Support Note 335754.1 announces the desupport of the oracle.jdbc.driver.* package in Oracle Database 11*g* JDBC drivers. In other words, Oracle Database 10*g* Release 2 was the last database to support this package and any API depending on the oracle.jdbc.driver.* package will fail to compile in the current release of the Database. You must remove such APIs and migrate to the standard APIs. For example, if your code uses the oracle.jdbc.CustomDatum and

oracle.jdbc.CustomDatumFactory interfaces, then you must replace them with the java.sql.Struct or java.sql.SQLData interfaces.

Related Topics

Database Administration

4.2.2 Support for Oracle Data Types

One of the features of the Oracle JDBC extensions is the type support in the oracle.sql package. This package includes classes that are an exact representation of the data in Oracle format. Keep the following important points in mind, when you use oracle.sql types in your program:

- For numeric type of data, the conversion to standard Java types does not guarantee to retain full precision due to limitations of the data conversion process. Use the BigDecimal type to minimize any data loss issues.
- For certain data types, the conversion to standard Java types can be dependent on the system settings and your program may not run as expected. This is a known limitation while converting data from oracle.sql types to standard Java types.
- If the functionalities of your program is limited to reading data from one table and writing the same to another table, then for numeric and date data, oracle.sql types are slightly faster as compared to standard Java types. But, if your program involves even a simple data manipulation operation like compare or print, then standard Java types are faster.
- oracle.sql.CHAR is not an exact representation of the data in Oracle format. oracle.sql.CHAR is constructed from java.lang.String. There is no advantage



of using oracle.sql.CHAR because java.lang.String is always faster and represents the same character sets, excluding a couple of desupported character sets.

Note:

Oracle strongly recommends you to use standard Java types and convert any existing oracle.sql type of data to standard Java types. Internally, the Oracle JDBC drivers strive to maximize the performance of Java standard types. oracle.sql types are supported *only* for backward compatibility and their use is discouraged.

Related Topics

• Package oracle.sql

The oracle.sql package supports direct access to data in SQL format. This package consists primarily of classes that provide Java mappings to SQL data types and their support classes. Essentially, the classes act as Java containers for SQL data.

- Oracle Character Data Types Support
- Additional Oracle Type Extensions

4.2.3 Support for Oracle Objects

Oracle JDBC supports the use of structured objects in the database, where an object data type is a user-defined type with nested attributes. For example, a user application could define an Employee object type, where each Employee object has a firstname attribute (character string), a lastname attribute (character string), and an employeenumber attribute (integer).

Oracle JDBC supports Oracle object data types. When you work with Oracle object data types in a Java application, you must consider the following:

- How to map between Oracle object data types and Java classes
- How to store Oracle object attributes in corresponding Java objects
- How to convert attribute data between SQL and Java formats
- How to access data

Oracle objects can be mapped either to the weak java.sql.Struct type or to strongly typed customized classes. These strong types are referred to as custom Java classes, which must implement either the standard java.sql.SQLData interface or the Oracle extension oracle.jdbc.OracleData interface. Each interface specifies methods to convert data between SQL and Java.

Note:

Starting from Oracle Database 12c Release 1 (12.1), the OracleData interface has replaced the ORAData interface.



Oracle recommends the use of the Oracle JVM Web Service Call-Out Utility to create custom Java classes to correspond to your Oracle objects.

Related Topics

- Working with Oracle Object Types
- Oracle Database Java Developer's Guide

4.2.4 Support for Schema Naming

Oracle object data type classes have the ability to accept and return fully qualified schema names. A fully qualified schema name has this syntax:

```
{[schema_name].}[sql_type_name]
```

Where, *schema_name* is the name of the schema and *sql_type_name* is the SQL type name of the object. *schema_name* and *sql_type_name* are separated by a period (.).

To specify an object type in JDBC, use its fully qualified name. It is not necessary to enter a schema name if the type name is in the current naming space, that is, the current schema. Schema naming follows these rules:

- Both the schema name and the type name may or may not be within quotation marks. However, if the SQL type name has a period in it, such as CORPORATE.EMPLOYEE, the type name must be quoted.
- The JDBC driver looks for the first period in the object name that is not within quotation marks and uses the string before the period as the schema name and the string following the period as the type name. If no period is found, then the JDBC driver takes the current schema as default. That is, you can specify only the type name, without indicating a schema, instead of specifying the fully qualified name if the object type name belongs to the current schema. This also explains why you must put the type name within quotation marks if the type name has a dot in it.

For example, assume that user HR creates a type called person.address and then wants to use it in his session. HR may want to skip the schema name and pass in person.address to the JDBC driver. In this case, if person.address is not within quotation marks, then the period is detected and the JDBC driver mistakenly interprets person as the schema name and address as the type name.

 JDBC passes the object type name string to the database unchanged. That is, the JDBC driver does not change the character case even if the object type name is within quotation marks.

For example, if HR.PersonType is passed to the JDBC driver as an object type name, then the JDBC driver passes the string to the database unchanged. As another example, if there is white space between characters in the type name string, then the JDBC driver will not remove the white space.

4.2.5 DML Returning

Oracle Database supports the use of the RETURNING clause with data manipulation language (DML) statements. This enables you to combine two SQL statements into one. Both the Oracle JDBC Oracle Call Interface (OCI) driver and the Oracle JDBC Thin driver support DML returning.



See Also: "DML Returning"

4.2.6 PL/SQL Associative Arrays

Oracle JDBC drivers enable JDBC applications to make PL/SQL calls with Associative Array parameters. Oracle JDBC drivers support PL/SQL Associative Arrays of scalar data types

See Also:

"Accessing PL/SQL Associative Arrays"

4.3 Oracle JDBC Packages

This section describes the following Java packages, which support the Oracle JDBC extensions:

- Package oracle.sql
- Package oracle.sql.json
- Package oracle.jdbc

4.3.1 Package oracle.sql

The oracle.sql package supports direct access to data in SQL format. This package consists primarily of classes that provide Java mappings to SQL data types and their support classes. Essentially, the classes act as Java containers for SQL data.

Each of the oracle.sql.* data type classes extends oracle.sql.Datum, a superclass that encapsulates functionality common to all the data types. Some of the classes are for JDBC 2.0-compliant data types. These classes, implement standard JDBC 2.0 interfaces in the java.sql package, as well as extending the oracle.sql.Datum class.

The LONG and LONG RAW SQL types and REF CURSOR type category have no oracle.sql.* classes. Use standard JDBC functionality for these types. For example, retrieve LONG or LONG RAW data as input streams using the standard JDBC result set and callable statement methods getBinaryStream and getCharacterStream. Use the getCursor method for REF CURSOR types.

Note:

Oracle recommends the use of standard JDBC types or Java types whenever possible. The types in the package oracle.sql.* are provided primarily for backward compatibility or for support of a few Oracle specific features such as OPAQUE, OracleData, TIMESTAMPTZ, and so on.



General oracle.sql.* Data Type Support

Each of the Oracle data type classes provides, among other things, the following:

- Data storage as Java byte arrays for SQL data
- A getBytes() method, which returns the SQL data as a byte array
- A toJdbc() method that converts the data into an object of a corresponding Java class as defined in the JDBC specification

The JDBC driver does not convert Oracle-specific data types that are not part of the JDBC specification, such as BFILE. The driver returns the object in the corresponding oracle.sql.* format.

- Appropriate xxxValue methods to convert SQL data to Java type. For example, stringValue, intValue, booleanValue, dateValue, and bigDecimalValue
- Additional conversion methods, getXXX and setXXX, as appropriate, for the functionality of the data type, such as methods in the large object (LOB) classes that get the data as a stream and methods in the REF class that get and set object data through the object reference.

Overview of Class oracle.sql.STRUCT

oracle.sql.STRUCT class is the Oracle implementation of java.sql.Struct interface. This class is a value class and you should not change the contents of the class after construction. This class, as with all oracle.sql.* data type classes, is a subclass of the oracle.sql.Datum class.

Note:

Starting from Oracle Database 12c Release 1 (12.1), the oracle.sql.STRUCT class is deprecated and replaced with the oracle.jdbc.OracleStruct interface, which is a part of the oracle.jdbc package. Oracle strongly recommends you to use the methods available in the java.sql package, where possible, for standard compatibility and methods available in the oracle.jdbc package for Oracle specific extensions. Refer to MoS Note 1364193.1 for more information about the oracle.jdbc.OracleStruct interface.

Overview of Class oracle.sql.REF

The oracle.sql.REF class is the generic class that supports Oracle object references. This class, as with all oracle.sql.* data type classes, is a subclass of the oracle.sql.Datum class.



Note:

Starting from Oracle Database 12c Release 1 (12.1), the oracle.sql.REF class is deprecated and replaced with the oracle.jdbc.OracleRef interface, which is a part of the oracle.jdbc package. Oracle strongly recommends you to use the methods available in the java.sql package, where possible, for standard compatibility and methods available in the oracle.jdbc package for Oracle specific extensions. Refer to MoS Note 1364193.1 for more information about the oracle.jdbc.OracleRef interface.

The REF class has methods to retrieve and pass object references. However, selecting an object reference retrieves only a pointer to an object. This does not materialize the object itself. But the REF class also includes methods to retrieve and pass the object data. You cannot create REF objects in your JDBC application. You can only retrieve existing REF objects from the database.

You should use the JDBC standard type, java.sql.Ref, and the JDBC standard methods in preference to using oracle.sql.REF. If you want your code to be more portable, then you must use the standard type because only the Oracle JDBC drivers will use instances of oracle.sql.REF type.

Overview of Classes oracle.sql.BLOB, oracle.sql.CLOB, oracle.sql.BFILE

Binary large objects (BLOBs), character large objects (CLOBs), and binary files (BFILEs) are for data items that are too large to store directly in a database table. Instead, the database table stores a locator that points to the location of the actual data.

Note:

- Starting from Oracle Database 12c Release 1 (12.1), the oracle.sql.BLOB and Oracle.sql.CLOB classes are deprecated and replaced with the oracle.jdbc.OracleBlob and oracle.jdbc.OracleClob interfaces respectively, which are a part of the oracle.jdbc package. Oracle strongly recommends you to use the methods available in the java.sql package, where possible, for standard compatibility and methods available in the oracle.jdbc package for Oracle specific extensions. Refer to MoS Note 1364193.1 for more information about the oracle.jdbc.OracleBlob and oracle.jdbc.OracleClob interfaces.
- oracle.sql.BFILE is an Oracle proprietary extension and there is no JDBC standard equivalent.

The oracle.sql package supports these data types in several ways:

- BLOBs point to large unstructured binary data items and are supported by the oracle.sql.BLOB class.
- CLOBs point to large character data items and are supported by the oracle.sql.CLOB class.
- BFILEs point to the content of external files (operating system files) and are supported by the oracle.sql.BFILE class. BFiles are read-only.



You can select a BLOB, CLOB, or BFILE locator from the database using a standard SELECT statement. However, you receive only the locator, and not the data. Additional steps are necessary to retrieve the data.

Overview of Classes oracle.sql.DATE, oracle.sql.NUMBER, and oracle.sql.RAW

These classes hold primitive SQL data types in Oracle native representation. In most cases, these types are not used internally by the drivers and you should use the standard JDBC types instead.

Java Double and Float NaN values do not have an equivalent Oracle NUMBER representation. For example, for Oracle BINARY_FLOAT and BINARY_DOUBLE data types, negative zero is coerced to positive zero and all NaNs are coerced to the canonical one. So, a NullPointerException is thrown whenever a Double.NaN value or a Float.NaN value is converted into an Oracle NUMBER using the oracle.sql.NUMBER class. For instance, the following code throws a NullPointerException:

oracle.sql.NUMBER n = new oracle.sql.NUMBER(Double.NaN); System.out.println(n.doubleValue()); // throws NullPointerException

Overview of Classes oracle.sql.TIMESTAMP, oracle.sql.TIMESTAMPTZ, and oracle.sql.TIMESTAMPLTZ

The JDBC drivers support the following date/time data types:

- TIMESTAMP (TIMESTAMP)
- TIMESTAMP WITH TIME ZONE (TIMESTAMPTZ)
- TIMESTAMP WITH LOCAL TIME ZONE (TIMESTAMPLTZ)

The JDBC drivers allow conversions between DATE and date/time data types. For example, you can access a TIMESTAMP WITH TIME ZONE column as a DATE value.

The JDBC drivers support the most popular time zone names used in the industry as well as most of the time zone names defined in the JDK. Time zones are specified by using the java.util.TimeZone class.

Note:

- Do not use TimeZone.getTimeZone to create time zone objects. The Oracle time zone data types support more time zone names than JDK.
- If a result set contains a TIMESTAMPLTZ column followed by a LONG column, then reading the LONG column results in an error.

The following code shows how the TimeZone and Calendar objects are created for US PACIFIC, which is a time zone name not defined in JDK:

```
TimeZone tz = TimeZone.getDefault();
tz.setID("US_PACIFIC");
GregorianCalendar gcal = new GregorianCalendar(tz);
```

The following Java classes represent the SQL date/time types:

oracle.sql.TIMESTAMP



- oracle.sql.TIMESTAMPTZ
- oracle.sql.TIMESTAMPLTZ

Before accessing TIMESTAMP WITH LOCAL TIME ZONE data, call the

OracleConnection.setSessionTimeZone (String regionName) method to set the session time zone. When this method is called, the JDBC driver sets the session time zone of the connection and saves the session time zone so that any TIMESTAMP WITH LOCAL TIME ZONE data accessed through JDBC can be adjusted using the session time zone.

Note:

TIMESTAMP WITH TIME ZONE and TIMESTAMP WITH LOCAL TIME ZONE types can be represented as standard java.sql.Timestamp type. The byte representation of TIMESTAMP WITH TIME ZONE and TIMESTAMP WITH LOCAL TIME ZONE types to java.sql.Timestamp is straight forward. This is because the internal format of TIMESTAMP WITH TIME ZONE and TIMESTAMP WITH LOCAL TIME ZONE data types is GMT, and java.sql.Timestamp type objects internally use a milliseconds time value that is the number of milliseconds since EPOCH. However, the String representation of these data types requires time zone information that is obtained dynamically from the server and cached on the client side.

In earlier versions of JDBC drivers, the cache of time zone was shared across different connections. This used to cause problems sometimes due to incompatibility in various time zones. Starting from Oracle Database 11 Release 2 version of JDBC drivers, the time zone cache is based on the time zone version supplied by the database. This newly designed cache avoids any issues related to version incompatibility of time zones.

Overview of Class oracle.sql.OPAQUE

The oracle.sql.OPAQUE class provides the name and characteristics of the OPAQUE type and any attributes. The OPAQUE type provides access only to the uninterrupted bytes of the instance.

Note:

Starting from Oracle Database 12c Release 1 (12.1), the oracle.sql.OPAQUE class is deprecated and replaced with the oracle.jdbc.OracleOpaque interface, which is a part of the oracle.jdbc package. Oracle recommends you to use the methods available in the java.sql package, where possible, for standard compatibility and methods available in the oracle.jdbc package for Oracle specific extensions. Refer to MoS Note 1364193.1 for more information about the oracle.jdbc.OracleOpaque interface.

Related Topics

- Oracle Database SQL Language Reference
- JDBC Java API Reference



Working with Large Objects and SecureFiles

Large Objects (LOBs) are a set of data types that are designed to hold large amounts of data. This chapter describes how to use Java Database Connectivity (JDBC) to access and manipulate LOBs and SecureFiles using either the data interface or the locator interface.

4.3.2 Package oracle.sql.json

Starting with release 21c, Oracle Database provides a native JSON SQL type in the database. The oracle.sql.json package provides functionality to work with the JSON type values.

Specifically, you can use the oracle.sql.json package to perform the following tasks:

- Store and retrieve JSON type values in the database
- Read, create, and modify JSON type values
- Encode or decode JSON type values in the same binary JSON storage format as used by the database
- Convert JSON type values to and from JSON text
- Bind and access JSON type values using the JSON-P interfaces like javax.json.*

See Also: JDBC Java API Reference

The following example shows how to insert, get, and modify JSON type values:

```
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import oracle.jdbc.OracleConnection;
import oracle.jdbc.pool.OracleDataSource;
import oracle.sql.json.OracleJsonFactory;
import oracle.sql.json.OracleJsonObject;
public class JsonExample {
 public static void main(String[] args) throws SQLException {
    OracleDataSource ds = new OracleDataSource();
    ds.setURL("jdbc:oracle:thin:@myhost:1521:orcl");
    ds.setUser(<user>);
    ds.setPassword(<password>);
    OracleConnection con = (OracleConnection) ds.getConnection();
    // create a table with a JSON column and insert one value
    Statement stmt = con.createStatement();
    stmt.executeUpdate("CREATE TABLE fruit (data JSON)");
    stmt.executeUpdate("INSERT INTO fruit VALUES
```



```
('{"name":"pear","count":10}')");
    // create another JSON object
   OracleJsonFactory factory = new OracleJsonFactory();
    OracleJsonObject orange = factory.createObject();
    orange.put("name", "orange");
    orange.put("count", 12);
    // insert the orange object
    PreparedStatement pstmt = con.prepareStatement("INSERT INTO fruit VALUES
(:1)");
    pstmt.setObject(1, orange, OracleType.JSON);
    pstmt.executeUpdate();
   pstmt.close();
    // retrieve the pear object
    ResultSet rs = stmt.executeQuery("SELECT data FROM fruit f WHERE
f.data.name = 'pear'");
    rs.next();
    OracleJsonObject pear = rs.getObject(1, OracleJsonObject.class);
    int count = pear.getInt("count");
    // create a modifiable copy of the pear object
   pear = factory.createObject(pear);
    pear.put("count", count + 1);
   pear.put("color", "green");
    // update the pear object
    pstmt = con.prepareStatement("UPDATE fruit f SET data = :1 WHERE
f.data.name = 'pear');
    pstmt.setObject(1, pear, OracleType.JSON);
    pstmt.executeUpdate();
   pstmt.close();
   rs.close();
    stmt.close();
    con.close();
  }
}
```

Compatibility with Client Libraries Prior to Release 21c

If you do not upgrade the earlier versions of your client library to 21c, then your application treats the JSON type column in the database as a String, BLOB, or CLOB column. So, to query the JSON type column, you must use the query methods for those data types. However, when you use the BLOB APIs like the getBlob method, they return an error, even after you upgrade the client library. So, Oracle recommends that for BLOB data type, you use the getBytes method or the getBinaryStream method, which will return text data with UTF-8 JSON encoding. The following code snippets show how to query a JSON type column for BLOB data type:

Example: Using the getBinaryStream Method

public static void fetchStream(ResultSet rs) throws SQLException, IOException



```
{
  InputStream is = rs.getBinaryStream("JCOL");
 ByteArrayOutputStream baos = new ByteArrayOutputStream();
 int n = -1;
 byte[] buffer = new byte[1024];
 while ((n = is.read(buffer)) != -1) {
   baos.write(buffer, 0, n);
  }
 is.close();
 byte[] bytes = baos.toByteArray();
 System.out.println(new String(bytes, StandardCharsets.UTF 8));
}
public static void example1 (Connection con) throws SQLException,
IOException
{
  Statement stmt = con.createStatement();
 ResultSet rs = stmt.executeQuery("select jcol from jtab where rownum
< 2");
 while (rs.next()) {
    fetchStream(rs);
  }
}
```

Example: Using the getBytes Method

```
public static void fetchString(ResultSet rs) throws SQLException
{
    byte[] utf8 = rs.getBytes("JCOL");
    System.out.println(new String(utf8, StandardCharsets.UTF_8));
}
public static void example2(Connection con) throws SQLException
{
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery("select jcol from jtab where rownum
    < 2");
    while (rs.next()){
        fetchString(rs);
    }
}</pre>
```

4.3.3 Package oracle.jdbc

The interfaces of the oracle.jdbc package define the Oracle extensions to the interfaces in java.sql. These extensions provide access to Oracle SQL-format data and other Oracle-specific functionality, including Oracle performance enhancements.



See Also: "The oracle.jdbc Package"

4.4 Oracle Character Data Types Support

Oracle character data types include the SQL CHAR and NCHAR data types. The following sections describe how these data types can be accessed using the oracle.sql.* classes:

- SQL CHAR Data Types
- SQL NCHAR Data Types
- Class oracle.sql.CHAR

4.4.1 SQL CHAR Data Types

The SQL CHAR data types include CHAR, VARCHAR2, and CLOB. These data types let you store character data in the database character set encoding scheme. The character set of the database is established when you create the database.

4.4.2 SQL NCHAR Data Types

The SQL NCHAR data types were created for Globalization Support. The SQL NCHAR data types include NCHAR, NVARCHAR2, and NCLOB. These data types enable you to store Unicode data in the database NCHAR character set encoding. The NCHAR character set, which never changes, is established when you create the database.

Note:

Because the UnicodeStream class is deprecated in favor of the CharacterStream class, the setUnicodeStream and getUnicodeStream methods are not supported for NCHAR data type access. Use the setCharacterStream method and the getCharacterStream method if you want to use stream access.

The usage of SQL NCHAR data types is similar to that of the SQL CHAR data types. JDBC uses the same classes and methods to access SQL NCHAR data types that are used for the corresponding SQL CHAR data types. Therefore, there are no separate, corresponding classes defined in the oracle.sql package for SQL NCHAR data types. Similarly, there is no separate, corresponding constant defined in the oracle.jdbc.OracleTypes class for SQL NCHAR data types.

See Also:

"NCHAR_ NVARCHAR2_ NCLOB and the defaultNChar Property"



Note:

The setFormOfUse method must be called before the registerOutParameter method is called in order to avoid unpredictable results.

The following code shows how to access SQL NCHAR data:

```
11
// Table TEST has the following columns:
// - NUMBER
// - NVARCHAR2
// - NCHAR
11
oracle.jdbc.OraclePreparedStatement pstmt =
  (oracle.jdbc.OraclePreparedStatement)
conn.prepareStatement("insert into TEST values(?, ?, ?)");
11
// oracle.jdbc.OraclePreparedStatement.FORM NCHAR should be used for all NCHAR,
// NVARCHAR2 and NCLOB data types.
11
                                      // NUMBER column
pstmt.setInt(1, 1);
pstmt.setNString(2, myUnicodeString1); // NVARCHAR2 column
pstmt.setNString(3, myUnicodeString2); // NCHAR column
pstmt.execute();
```

4.4.3 Class oracle.sql.CHAR

The oracle.sql.CHAR class is used by Oracle JDBC in handling and converting character data. This class provides the Globalization Support functionality to convert character data. This class has two key attributes: Globalization Support character set and the character data. The Globalization Support character set defines the encoding of the character data. It is a parameter that is always passed when a CHAR object is constructed. Without the Globalization Support character set information, the data bytes in the CHAR object are meaningless. The oracle.sql.CHAR class is used for both SQL CHAR and SQL NCHAR data types.

Note:

In versions of Oracle JDBC drivers prior to 10g Release 1, there were performance advantages to using the oracle.SQL.CHAR. Starting from Oracle Database 10g, there are no longer any such advantages. In fact, optimum performance is achieved using the java.lang.String. All Oracle JDBC drivers handle all character data in the Java UCS2 character set. Using the oracle.sql.CHAR does not prevent conversions between the database character set and UCS2 character set.

The only remaining use of the oracle.sql.CHAR class is to handle character data in the form of raw bytes encoded in an Oracle Globalization Support character set. All character data retrieved from Oracle Database should be accessed using the



java.lang.String class. When processing byte data from another source, you can use an oracle.sql.CHAR to convert the bytes to java.lang.String.

To convert an oracle.sql.CHAR, you must provide the data bytes and an oracle.sql.CharacterSet instance that represents the Globalization Support character set used to encode the data bytes.

The CHAR objects that are Oracle object attributes are returned in the database character set.

JDBC application code rarely needs to construct CHAR objects directly, because the JDBC driver automatically creates CHAR objects, when it is needed to create them on those rare occasions.

To construct a CHAR object, you must provide character set information to the CHAR object by way of an instance of the CharacterSet class. Each instance of this class represents one of the Globalization Support character sets that Oracle supports. A CharacterSet instance encapsulates methods and attributes of the character set, mainly involving functionality to convert to or from other character sets.

Constructing an oracle.sql.CHAR Object

Follow these general steps to construct a CHAR object:

1. Create a CharacterSet object by calling the static CharacterSet.make method.

This method is a factory for the character set instance. The make method takes an integer as input, which corresponds to a character set ID that Oracle supports. For example:

Each character set that Oracle supports has a unique, predefined Oracle ID.

2. Construct a CHAR object.

Pass a string, or the bytes that represent the string, to the constructor along with the CharacterSet object that indicates how to interpret the bytes based on the character set. For example:

```
String mystring = "teststring";
...
CHAR mychar = new CHAR(teststring, mycharset);
```

There are multiple constructors for CHAR, which can take a String, a byte array, or an object as input along with the CharacterSet object. In the case of a String, the string is converted to the character set indicated by the CharacterSet object before being placed into the CHAR object.



Note:

- The CharacterSet object cannot be a null value.
- The CharacterSet class is an abstract class, therefore it has no constructor. The only way to create instances is to use the make method.
- The server recognizes the special value CharacterSet.DEFAULT_CHARSET as the database character set. For the client, this value is not meaningful.
- Oracle does not intend or recommend that users extend the CharacterSet class.

oracle.sql.CHAR Conversion Methods

The CHAR class provides the following methods for translating character data to strings:

getString

This method converts the sequence of characters represented by the CHAR object to a string, returning a Java String object. If you enter an invalid OracleID, then the character set will not be recognized and the getString method will throw a SQLException exception.

toString

This method is identical to the getString method. But if you enter an invalid OracleID, then the character set will not be recognized and the toString method will return a hexadecimal representation of the CHAR data and will *not* throw a SQLException exception.

getStringWithReplacement

This method is identical to the getString method, except a default replacement character replaces characters that have no unicode representation in the CHAR object character set. This default character varies from character set to character set, but is often a question mark (?).

The database server and the client, or application running on the client, can use different character sets. When you use the methods of the CHAR class to transfer data between the server and the client, the JDBC drivers must convert the data from the server character set to the client character set or vice versa. To convert the data, the drivers use Globalization Support.

See Also: Globalization Support



4.5 Additional Oracle Type Extensions

Oracle JDBC drivers support the Oracle-specific BFILE and ROWID data types and REF CURSOR types, which are not part of the standard JDBC specification. This section describes the ROWID and REF CURSOR type extensions. The ROWID is supported as a Java string, and REF CURSOR types are supported as JDBC result sets.

This section covers the following topics:

- Oracle ROWID Type
- Oracle REF CURSOR Type Category
- Oracle BINARY_FLOAT and BINARY_DOUBLE Types
- Oracle SYS.ANYTYPE and SYS.ANYDATA Types
- The oracle.jdbc Package

4.5.1 Oracle ROWID Type

A ROWID is an identification tag unique for each row of an Oracle Database table. The ROWID can be thought of as a virtual column, containing the ID for each row.

The oracle.sql.ROWID class is supplied as a container for ROWID SQL data type.

ROWIDs provide functionality similar to the getCursorName method specified in the java.sql.ResultSet interface and the setCursorName method specified in the java.sql.Statement interface.

If you include the ROWID pseudo-column in a query, then you can retrieve the ROWIDs with the result set getString method. You can also bind a ROWID to a PreparedStatement parameter with the setString method. This enables in-place updating, as in the example that follows.

Note:

Use the oracle.sql.ROWID class, only when you are using J2SE 5.0. For JSE 6, you should use the standard java.sql.RowId interface instead.

Example

The following example shows how to access and manipulate ROWID data:

Note:

The following example works only with JSE 6.

```
Statement stmt = conn.createStatement();
```

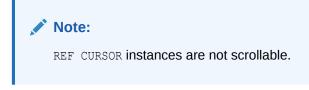
// Query the employee names with "FOR UPDATE" to lock the rows.



```
// Select the ROWID to identify the rows to be updated.
ResultSet rset =
   stmt.executeQuery ("SELECT first name, rowid FROM employees FOR UPDATE");
// Prepare a statement to update the first name column at a given ROWID
PreparedStatement pstmt =
  conn.prepareStatement ("UPDATE employees SET first name = ? WHERE rowid
= ?");
// Loop through the results of the query
while (rset.next ())
{
   String ename = rset.getString (1);
RowId rowid = rset.getROWID(2); // Get the ROWID as a String
   pstmt.setString (1, ename.toLowerCase ());
   pstmt.setROWID (2, rowid); // Pass ROWID to the update statement
   pstmt.executeUpdate (); // Do the update
}
```

4.5.2 Oracle REF CURSOR Type Category

A cursor variable holds the memory location of a query work area, rather than the contents of the area. Declaring a cursor variable creates a pointer. In SQL, a pointer has the data type REF *x*, where REF is short for REFERENCE and *x* represents the entity being referenced. A REF CURSOR, then, identifies a reference to a cursor variable. Because many cursor variables might exist to point to many work areas, REF CURSOR can be thought of as a category or data type specifier that identifies many different types of cursor variables. Starting from Oracle Database Release 18 *c*, JDBC drivers support REF CURSOR as IN bind variables.



Perform the following steps to create a cursor variable:

1. Identify a type that belongs to the REF CURSOR category. For example:

DECLARE TYPE DeptCursorTyp IS REF CURSOR

2. Then, create the cursor variable by declaring it to be of the type DeptCursorTyp:

```
dept_cv DeptCursorTyp - - declare cursor variable
...
```

REF CURSOR, then, is a category of data types, rather than a particular data type.

Stored procedures can accept or return cursor variables of the REF CURSOR category. This output is equivalent to a database cursor or a JDBC result set. A REF CURSOR essentially encapsulates the results of a query.

In JDBC, a REF CURSOR can be accessed as follows:



- **1.** Use a JDBC callable statement or a prepared statement to call a stored procedure.
- 2. The stored procedure accepts or returns a REF CURSOR.
- 3. The Java application casts the callable statement or prepared statement to an Oracle callable statement or Oracle prepared statement.
- 4. The Java application uses the setCursor method of the OraclePreparedStatement interface or the getCursor method of the OracleCallableStatement interface to materialize the REF CURSOR as a JDBC ResultSet object.
- 5. The result set is processed as requested.

Note:

- The cursor associated with a REF CURSOR is closed whenever the statement object that produced the REF CURSOR is closed.
- Unlike in past releases, the cursor associated with a REF CURSOR is not closed when the result set object in which the REF CURSOR was materialized is closed.

Example

This example shows how to access REF CURSOR data.

```
. . .
    // Prepare a PL/SQL call
   CallableStatement cstmt =
      conn.prepareCall ("DECLARE rc sys refcursor; curid NUMBER; BEGIN open
rc FOR SELECT empno FROM emp order by empno; ? := rc; END;");
    cstmt.registerOutParameter (1, OracleTypes.CURSOR);
    cstmt.execute ();
   ResultSet rset = (ResultSet)cstmt.getObject (1);
    if (rset.next ()) {
      show (rset.getString ("empno"));
    }
   CallableStatement cstmt2 =
     conn.prepareCall ("DECLARE rc sys refcursor; v1 NUMBER; BEGIN
rc := ?; fetch rc INTO v1; ? := v1; END;");
    ((OracleCallableStatement)call2).setCursor(1, rset);
    cstmt2.registerOutParameter (2, OracleTypes.INTEGER);
    cstmt2.execute();
    int empno = cstmt2.getInt(2);
    show("Fetch in PL/SQL empno=" + empno);
    // Dump the cursor
    while (rset.next ())
```



```
show (rset.getString ("empno"));
// Close all the resources
rset.close();
cstmt.close();
cstmt2.close();
...
```

In the preceding example:

- Two CallableStatement objects cstmt1 and cstmt2 are created using the prepareCall method of the Connection class.
- The cstmt2 callable statement uses REF CURSOR as input parameter.
- The callable statements implement PL/SQL procedure that returns a REF CURSOR.
- As always, the output parameter of the callable statement must be registered to define its type. Use the type code <code>OracleTypes.CURSOR</code> for a <code>REF</code> CURSOR.
- The callable statements are run, returning the REF CURSOR or sending the REF CURSOR as input bind.

4.5.3 Oracle BINARY_FLOAT and BINARY_DOUBLE Types

The Oracle BINARY_FLOAT and BINARY_DOUBLE types are used to store IEEE 574 float and double data. These correspond to the Java float and double scalar types with the exception of negative zero and NaN.

See Also:

Oracle Database SQL Language Reference

If you include a BINARY_DOUBLE column in a query, then the data is retrieved from the database in the binary format. Also, the getDouble method will return the data in the binary format. In contrast, for a NUMBER data type column, the number bits are returned and converted to the Java double data type.

Note:

The Oracle representation for the SQL FLOAT, DOUBLE PRECISION, and REAL data types use the Oracle NUMBER representation. The BINARY_FLOAT and BINARY_DOUBLE data types can be regarded as proprietary types.

A call to the JDBC standard setDouble(int, double) method of the PreparedStatement interface converts the Java double argument to Oracle NUMBER style bits and send them to the database. In contrast, the setBinaryDouble(int, double) method of the oracle.jdbc.OraclePreparedStatement interface converts the data to the internal binary bits and sends them to the database.



You must ensure that the data format used matches the type of the target parameter of the PreparedStatement interface. This will result in correct data and least use of CPU. If you use setBinaryDouble for a NUMBER parameter, then the binary bits are sent to the server and converted to NUMBER format. The data will be correct, but server CPU load will be increased. If you use setDouble for a BINARY_DOUBLE parameter, then the data will first be converted to NUMBER bits on the client and sent to the server, where it will be converted back to binary format. This will increase the CPU load on both client and server and can result in data corruption as well.

The SetFloatAndDoubleUseBinary connection property when set to true causes the JDBC standard APIs, setFloat(int, float), setDouble(int, double), and all the variations, to send internal binary bits instead of NUBMER bits.

Note:

Although this section largely discusses **BINARY_DOUBLE**, the same is true for **BINARY_FLOAT** as well.

4.5.4 Oracle SYS.ANYTYPE and SYS.ANYDATA Types

Oracle Database 12c Release 1 (12.1) provides a Java interface to access the SYS.ANYTYPE and SYS.ANYDATA Oracle types.

See Also:

For information about these Oracle types, refer Oracle Database PL/SQL Packages and Types Reference

An instance of the SYS.ANYTYPE type contains a type description of any SQL type, persistent or transient, named or unnamed, including object types and collection types. You can use the oracle.sql.TypeDescriptor class to access the SYS.ANYTYPE type. An ANYTYPE instance can be retrieved from a PL/SQL procedure or a SQL SELECT statement where SYS.ANYTYPE is used as a column type. To retrieve an ANYTYPE instance from the database, use the getObject method. This method returns an instance of the TypeDescriptor.

The retrieved ANYTYPE instance could be any of the following:

- Transient object type
- Transient predefined type
- Persistent object type
- Persistent predefined type

Example 4-1 Accessing SYS.ANYTYPE Type

The following code snippet illustrates how to retrieve an instance of ANYTYPE from the database:

ResultSet rs = stmt.executeQuery("select anytype column from my table");



```
TypeDescriptor td = (TypeDescriptor)rs.getObject(1);
short typeCode = td.getInternalTypeCode();
if(typeCode == TypeDescriptor.TYPECODE_OBJECT)
{
    // check if it's a transient type
    if(td.isTransientType())
    {
        AttributeDescriptor[] attributes =
    ((StructDescriptor)td).getAttributesDescriptor();
        for(int i=0; i<attributes.length; i++)
           System.out.println(attributes[i].getAttributeName());
    }
    else
    {      System.out.println(td.getTypeName()); }}
...
```

Example 4-2 Creating a Transient Object Type Through PL/SQL and Retrieving Through JDBC

This example provides a code snippet illustrating how to retrieve a transient object type through JDBC.

```
OracleCallableStatement cstmt = (OracleCallableStatement)conn.prepareCall
  ("BEGIN ? := transient_obj_type (); END;");
  cstmt.registerOutParameter(1,OracleTypes.OPAQUE,"SYS.ANYTYPE");
  cstmt.execute();
  TypeDescriptor obj = (TypeDescriptor)cstmt.getObject(1);
  if(!obj.isTransient())
   System.out.println("This must be a JDBC bug");
  cstmt.close();
  return obj;
  ...
```

Example 4-3 Calling a PL/SQL Stored Procedure That Takes an ANYTPE as IN Parameter

The following code snippet illustrates how to call a PL/SQL stored procedure that takes an ANYTYPE as IN parameter:

```
CallableStatement cstmt = conn.prepareCall("BEGIN ? := dumpanytype(?); END;");
cstmt.registerOutParameter(1,OracleTypes.VARCHAR);
// obj is the instance of TypeDescriptor that you have retrieved
cstmt.setObject(2,obj);
cstmt.execute();
String str = (String)cstmt.getObject(1);
...
```

The oracle.sql.ANYDATA class enables you to access SYS.ANYDATA instances from the database. An instance of this class can be obtained from any valid instance of oracle.sql.Datum class. The convertDatum factory method takes an instance of Datum and returns an instance of ANYDATA. The syntax for this factory method is as follows:

public static ANYDATA convertDatum(Datum datum) throws SQLException

The following is sample code for creating an instance of oracle.sql.ANYDATA:



```
// struct is a valid instance of oracle.sql.STRUCT that either comes from the
// database or has been constructed in Java.
ANYDATA myAnyData = ANYDATA.convertDatum(struct);
```

Example 4-4 Accessing an Instance of ANYDATA from the Database

```
...
// anydata_table has been created as:
// CREATE TABLE anydata_tab (data SYS.ANYDATA)
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("select data from my_anydata_tab");
while(rs.next())
{
    ANYDATA anydata = (ANYDATA)rs.getObject(1);
    if(!anydata.isNull())
    {
      TypeDescriptor td = anydata.getTypeDescriptor();
      if(td.getTypeCode() == OracleType.TYPECODE_OBJECT)
        STRUCT struct = (STRUCT)anydata.accessDatum();
    }
}
....
```

Example 4-5 Inserting an Object as ANYDATA in a Database Table

Consider the following table and object type definition:

CREATE TABLE anydata_tab (id NUMBER, data SYS.ANYDATA)

CREATE OR REPLACE TYPE employee AS OBJECT (employee_id NUMBER, first_name VARCHAR2(10))

You can create an instance of the EMPLOYEE SQL object type and to insert it into anydata table in the following way:

```
...
PreparedStatement pstmt = conn.prepareStatement("insert into anydata_table values
(?,?)");
Struct myEmployeeStr = conn.createStruct("EMPLOYEE", new Object[]{1120, "Papageno"});
ANYDATA anyda = ANYDATA.convertDatum(myEmployeeStr);
pstmt.setInt(1,123);
pstmt.setObject(2,anyda);
pstmt.executeUpdate();
...
```

Example 4-6 Selecting an ANYDATA Column from a Database Table

```
...
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("select data from anydata_table");
while(rs.next())
{
    ANYDATA obj = (ANYDATA)rs.getObject(1);
    TypeDescriptor td = obj.getTypeDescriptor();
}
rs.close();
stmt.close();
...
```



4.5.5 The oracle.jdbc Package

The interfaces of the oracle.jdbc package define the Oracle extensions to the interfaces in java.sql. These extensions provide access to SQL-format data as described in this chapter. They also provide access to other Oracle-specific functionality, including Oracle performance enhancements.

For the oracle.jdbc package, Table 4-1 lists key interfaces and classes used for connections, statements, and result sets.

Name	Interface or Class	Key Functionality
OracleDriver	Class	Implements java.sql.Driver
OracleConnection	Interface	Provides methods to start and stop an Oracle Database instance and to return Oracle statement objects and methods to set Oracle performance extensions for any statement run in the current connection.
		<pre>Implements java.sql.Connection.</pre>
OracleStatement	Interface	Provides methods to set Oracle performance extensions for individual statement.
		Is a supertype of OraclePreparedStatement and OracleCallableStatement.
		Implements java.sql.Statement.
OraclePreparedStatement	Interface	Provides setXXX methods to bind oracle.sql.* types into a prepared statement.
		Provides getMetaData method to get the metadata from the prepared statements without executing the SELECT statements.
		<pre>Implements java.sql.PreparedStatement.</pre>
		Extends OracleStatement.
		ls a supertype of OracleCallableStatement.
OracleCallableStatement	Interface	Provides getXXX methods to retrieve data in oracle.sql format and setXXX methods to bind oracle.sql.* types into a callable statement.
		<pre>Implements java.sql.CallableStatement.</pre>
		Extends OraclePreparedStatement.
OracleResultSet	Interface	Provides getXXX methods to retrieve data in oracle.sql format.
		Implements java.sql.ResultSet.

Table 4-1 Key Interfaces and Classes of the oracle.jdbc Package



Name	Interface or Class	Key Functionality
OracleResultSetMetaData	Interface	Provides methods to get metadata information about Oracle result sets, such as column names and data types.
		lmplements java.sql.ResultSetMetaData.
OracleDatabaseMetaData	Class	Provides methods to get metadata information about the database, such as database product name and version, table information, and default transaction isolation level.
		lmplements java.sql.DatabaseMetaData).
OracleTypes	Class	Defines integer constants used to identify SQL types.
		For standard types, it uses the same values as the standard java.sql.Types class. In addition, it adds constants for Oracle extended types.
OracleArray	Interface	Includes functionality to retrieve the array as a whole, retrieve a subset of the array elements, and retrieve the SQL base type name of the array elements.
OracleStruct	Interface	
OracleClob	Interface	
OracleBlob	Interface	
OracleRef	Interface	
OracleOpaque	Interface	

Table 4-1 (Cont.) Key Interfaces and Classes of the oracle.jdbc Package

This section covers the following topics:

- Interface oracle.jdbc.OracleConnection
- Interface oracle.jdbc.OracleStatement
- Interface oracle.jdbc.OraclePreparedStatement
- Interface oracle.jdbc.OracleCallableStatement
- Interface oracle.jdbc.OracleResultSet
- Interface oracle.jdbc.OracleResultSetMetaData
- Class oracle.jdbc.OracleTypes

4.5.5.1 Interface oracle.jdbc.OracleConnection

This interface extends standard JDBC connection functionality to create and return Oracle statement objects, set flags and options for Oracle performance extensions, support type maps for Oracle objects, and support client identifiers.



In Oracle Database 11g Release 1, new methods were added to this interface that enable the starting up and shutting down of an Oracle Database instance. Also, for better visibility and clarity, all connection properties are defined as constants in the OracleConnection interface.

This interface also defines factory methods for constructing oracle.sql data values like DATE and NUMBER. Remember the following points while using factory methods:

- All code that constructs instances of the oracle.sql types should use the Oracle extension factory methods. For example, ARRAY, BFILE, DATE, INTERVALDS, NUMBER, STRUCT, TIME, TIMESTAMP, and so on.
- All code that constructs instances of the standard types should use the JDBC 4.0 standard factory methods. For example, CLOB, BLOB, NCLOB, and so on.
- There are no factory methods for CHAR, JAVA_STRUCT, ArrayDescriptor, and StructDescriptor. These types are for internal driver use only.

Note:

Prior to Oracle Database 11*g* Release 1, you had to construct ArrayDescriptors and StructDescriptors for passing as arguments to the ARRAY and STRUCT class constructors. The new ARRAY and Struct factory methods do not have any descriptor arguments. The driver still uses descriptors internally, but you do not need to create them.

Client Identifiers

In a connection pooling environment, the client identifier can be used to identify the lightweight user using the database session currently. A client identifier can also be used to share the Globally Accessed Application Context between different database sessions. The client identifier set in a database session is audited when database auditing is turned on.

💉 See Also:

Oracle Database JDBC Java API Reference for more information

4.5.5.2 Interface oracle.jdbc.OracleStatement

This interface extends standard JDBC statement functionality and is the superinterface of the <code>OraclePreparedStatement</code> and <code>OracleCallableStatement</code> classes. Extended functionality includes support for setting flags and options for Oracle performance extensions on a statement-by-statement basis, as opposed to the <code>OracleConnection</code> interface that sets these on a connectionwide basis.

4.5.5.3 Interface oracle.jdbc.OraclePreparedStatement

This interface extends the OracleStatement interface and extends standard JDBC prepared statement functionality. Also, the oracle.jdbc.OraclePreparedStatement



interface is extended by the OracleCallableStatement interface. Extended functionality consists of the following:

- set XXX methods for binding oracle.sql.* types and objects to prepared statements
- getMetaData method to get the metadata from the prepared statements without executing the SELECT statements
- Methods to support Oracle performance extensions on a statement-by-statement basis

Note:

Do not use the PreparedStatement interface to create a trigger that refers to a:NEW or :OLD column. Use Statement instead. Using PreparedStatement will cause execution to fail with the message java.sql.SQLException: Missing IN or OUT parameter at index:: 1.

4.5.5.4 Interface oracle.jdbc.OracleCallableStatement

This interface extends the <code>OraclePreparedStatement</code> interface, which extends the <code>OracleStatement</code> interface and incorporates standard JDBC callable statement functionality.

Note:

Do not use the CallableStatement interface to create a trigger that refers to a:NEW or :OLD column. Use Statement instead; using CallableStatement will cause execution to fail with the message java.sql.SQLException: Missing IN or OUT parameter at index::1

Note:

- The setXXX(String,...) and registerOutParameter(String,...) methods can be used only if all binds are procedure or function parameters only. The statement can contain no other binds and the parameter binds must be indicated with a question mark (?) and not :XX.
- If you are using setXXX(int,...) or setXXXAtName(String,...) method, then any output parameter is bound with registerOutParameter(int,...) and not registerOutParameter(String,...), which is for named parameter notation.

4.5.5.5 Interface oracle.jdbc.OracleResultSet

This interface extends standard JDBC result set functionality, implementing getXXX methods for retrieving data into oracle.sql.* objects.



4.5.5.6 Interface oracle.jdbc.OracleResultSetMetaData

This interface extends standard JDBC result set metadata functionality to retrieve information about Oracle result set objects.



4.5.5.7 Class oracle.jdbc.OracleTypes

The <code>OracleTypes</code> class defines constants that JDBC uses to identify SQL types. Each variable in this class has a constant integer value. The <code>oracle.jdbc.OracleTypes</code> class duplicates the type code definitions of the standard Java <code>java.sql.Types</code> class and contains these additional type codes for Oracle extensions:

- OracleTypes.BFILE
- OracleTypes.ROWID
- OracleTypes.CURSOR (for REF CURSOR types)
- OracleTypes.CHAR_BYTES (for calling setNull and setCHAR methods on the same column)

As in java.sql.Types, all the variable names in oracle.jdbc.OracleTypes are also in uppercase text. JDBC uses the SQL types identified by the elements of the OracleTypes class in the following three main areas:

- Registering output parameters
- Using the setNull method
- Supporting the new BOOLEAN data type

Registering Output Parameters

The type codes in java.sql.Types or oracle.jdbc.OracleTypes identify the SQL types of the output parameters in the registerOutParameter method of the java.sql.CallableStatement and oracle.jdbc.OracleCallableStatement interfaces.

These are the forms that the registerOutputParameter method can take for the CallableStatement and OracleCallableStatement interfaces

cs.registerOutParameter(int index, int sqlType); cs.registerOutParameter(int index, int sqlType, String sql_name); cs.registerOutParameter(int index, int sqlType, int scale);

In these signatures, index represents the parameter index, sqlType is the type code for the SQL data type, sql_name is the name given to the data type, for user-defined types, when sqlType is a STRUCT, REF, or ARRAY type code, and scale represents the



number of digits to the right of the decimal point, when sqlType is a NUMERIC or DECIMAL type code.

The following example uses a CallableStatement interface to call a procedure named charout, which returns a CHAR data type. Note the use of the OracleTypes.CHAR type code in the registerOutParameter method.

```
CallableStatement cs = conn.prepareCall ("BEGIN charout (?); END;");
cs.registerOutParameter (1, OracleTypes.CHAR);
cs.execute ();
System.out.println ("Out argument is: " + cs.getString (1));
```

The next example uses a CallableStatement interface to call structout, which returns a STRUCT data type. The form of registerOutParameter requires you to specify the type code, Types.STRUCT or OracleTypes.STRUCT, as well as the SQL name, EMPLOYEE.

The example assumes that no type mapping has been declared for the EMPLOYEE type, so it is retrieved into a STRUCT data type. To retrieve the value of EMPLOYEE as an oracle.sql.STRUCT object, the statement object cs is cast to OracleCallableStatement and the Oracle extension getSTRUCT method is invoked.

```
CallableStatement cs = conn.prepareCall ("BEGIN structout (?); END;");
cs.registerOutParameter (1, OracleTypes.STRUCT, "EMPLOYEE");
cs.execute ();
```

```
// get the value into a STRUCT because it
// is assumed that no type map has been defined
STRUCT emp = ((OracleCallableStatement)cs).getSTRUCT (1);
```

Using the setNull Method

The type codes in Types and OracleTypes identify the SQL type of the data item, which the setNull method sets to NULL. The setNull method can be found in the java.sql.PreparedStatement and oracle.jdbc.OraclePreparedStatement interfaces.

These are the forms that the setNull method can take for the PreparedStatement and OraclePreparedStatement objects:

```
ps.setNull(int index, int sqlType);
ps.setNull(int index, int sqlType, String sql name);
```

In these signatures, index represents the parameter index, sqlType is the type code for the SQL data type, and sql_name is the name given to the data type, for user-defined types, when sqlType is a STRUCT, REF, or ARRAY type code. If you enter an invalid sqlType, a ParameterTypeConflict exception is thrown.

The following example uses a prepared statement to insert a null value into the database. Note the use of OracleTypes.NUMERIC to identify the numeric object set to NULL. Alternatively, Types.NUMERIC can be used.

```
PreparedStatement pstmt =
    conn.prepareStatement ("INSERT INTO num_table VALUES (?)");
pstmt.setNull (1, OracleTypes.NUMERIC);
pstmt.execute ();
```

In this example, the prepared statement inserts a NULL STRUCT object of type EMPLOYEE into the database.



pstmt.execute ();

You can also use the <code>OracleTypes.CHAR_BYTES</code> type with the <code>setNull</code> method, if you also want to call the <code>setCHAR</code> method on the same column. For example:

```
ps.setCHAR(n, aCHAR);
ps.addBatch();
ps.setNull(n, OracleTypes.CHAR_BYTES);
ps.addBatch();
```

In this preceding example, any other type, apart from the <code>OracleTypes.CHAR_BYTES</code> type, will cause extra round trips to the Database. Alternatively, you can also write your code without using the <code>setNull</code> method. For example, you can also write your code as shown in the following example:

```
ps.setCHAR(n, null);
```

Supporting the BOOLEAN Data Type

Starting from Release 23c, Oracle Database supports the BOOLEAN data type in compliance with the ISO SQL standard. The current release of JDBC Thin driver provides support for the newly introduced BOOLEAN data type, with the introduction of a new type BOOLEAN in oracle.jdbc.OracleType.

The following table lists the APIs to work with this new type:

Operation	API Used	Code Example
Insert	java.sql.PreparedStat ement.setBoolean or jav a.sql.PreparedStateme	Example:
	nt.setObject	<pre>String query = "INSERT INTO BoolTable values (?) "; PreparedStatement pstmt = con.prepareStatement(query); pstmt.setBoolean(1, true); pstmt.execute();</pre>
Fetch	java.sql.ResultSet.ge tBoolean or any equivalent method	<pre> Statement stmt = con.createStatement(); ResultSet rs = stmt.executeQuery("select * from BoolTable"); while(rs.next()) { System.out.print("Value: " + rs.getBoolean("booleanColumn")); }</pre>



Operation	API Used	Code Example
Fetch column metadata	java.sql.ResultSetMet aData	<pre> try (OraclePreparedStatement stmt = (OraclePreparedStatement) conn.prepareStatement("SELECT ? < ?")) { ResultSetMetaData rsmd = stmt.getMetaData(); System.out.print(rsmd.getColumnTyp e(1) + " "); System.out.println(rsmd.getColumnT ypeName(1) + " "); }</pre>
		The expected output of the preceding code snippet is: 16 BOOLEAN

Note:

If the columns with which the values are being compared to or the column into which values are inserted, are not of the appropriate types, then an implicit conversion may take place. If the value cannot be converted, an exception is thrown.

Supported and Unsupported Conversions for the BOOLEAN Type

Oracle Database allows implicit conversion from character and number types of data to BOOLEAN type, and BOOLEAN type to character and number types. So, you can also use external types for these types, while the Database takes care of implicit conversions for the binds and the client takes care of the implicit conversions for the defines.

The following table lists the supported conversions:

Java Type	Value	BOOLOEAN Value
boolean	TRUE or FALSE	TRUE or FALSE
String	"TRUE" or "FALSE" (case- insensitive)	TRUE or FALSE
char,int,long	Non-zero value or zero-value	TRUE or FALSE

The following table lists the String literals to represent "True" and "False":

True (case-insensitive)	False (case-insensitive)
"true"	"false"
"yes"	"no"



True (case-insensitive)	False (case-insensitive)	
"on"	"off"	
"1"	" () "	
"t"	"f"	
"Y"	"n"	

Conversion between the NUMBER and BOOLEAN types with non-zero scale returns an error. Also, when converting from string to BOOLEAN, if the string literal value is not any of the values listed in the preceding table, then an error is returned.

4.6 DML Returning

The DML returning feature provides more functionality compared to retrieval of autogenerated keys. It can be used to retrieve not only auto-generated keys, but also other columns or values that the application may use.

Note:

- The server-side internal driver does not support DML returning and retrieval of auto-generated keys.
- You cannot use both DML returning and retrieval of auto-generated keys in the same statement.

The following sections explain the support for DML returning:

- Oracle-Specific APIs
- About Running DML Returning Statements
- Example of DML Returning
- Limitations of DML Returning

4.6.1 Oracle-Specific APIs

The OraclePreparedStatement interface is enhanced with Oracle-specific application programming interfaces (APIs) to support DML returning. The registerReturnParameter and getReturnResultSet methods have been added to the oracle.jdbc.OraclePreparedStatement interface, to register parameters that are returned and data retrieved by DML returning.

The registerReturnParameter method is used to register the return parameter for DML returning. The method throws a SQLException instance if an error occurs. You must pass a positive integer specifying the index of the return parameter. You also must specify the type of the return parameter. You can also specify the maximum bytes or characters of the return parameter. This method can be used only with char or RAW types. You can also specify the fully qualified name of a SQL structure type.



Note:

If you do not know the maximum size of the return parameters, then you should use registerReturnParameter(int paramIndex, int externalType), which picks the default maximum size. If you know the maximum size of return parameters, using registerReturnParameter(int paramIndex, int externalType, int maxSize) can reduce memory consumption.

The getReturnResultSet method fetches the data returned from DML returning and returns it as a ResultSet object. The method throws a SQLException exception if an error occurs.

Note:

The Oracle-specific APIs for the DML returning feature are in <code>ojdbc6.jar</code> for Java Development Kit (JDK) 6.0 and in <code>ojdbc7.jar</code> for JDK 7.

4.6.2 About Running DML Returning Statements

Before running a DML returning statement, the JDBC application must call one or more of the registerReturnParameter methods. The method provides the JDBC drivers with information, such as type and size, of the return parameters. The DML returning statement is then processed using one of the standard JDBC APIs, executeUpdate or execute. You can then fetch the returned parameters as a ResultSet object using the getReturnResultSet method of the oracle.jdbc.OraclePreparedStatement interface.

In order to read the values in the ResultSet object, the underlying Statement object must be open. When the underlying Statement object is closed, the returned ResultSet object is also closed. This is consistent with ResultSet objects that are retrieved by processing SQL query statements.

When a DML returning statement is run, the concurrency of the ResultSet object returned by the getReturnResultSet method must be CONCUR_READ_ONLY and the type of the ResultSet object must be TYPE_FORWARD_ONLY or TYPE_SCROLL_INSENSITIVE.

4.6.3 Example of DML Returning

This section provides two code examples of DML returning.

The following code example illustrates the use of DML returning. In this example, assume that the maximum size of the name column is 100 characters. Because the maximum size of the name column is known, the registerReturnParameter(int paramIndex, int externalType, int maxSize) method is used.

```
...
OraclePreparedStatement pstmt = (OraclePreparedStatement)conn.prepareStatement(
    "delete from tabl where age < ? returning name into ?");
pstmt.setInt(1,18);
/** register returned parameter
    * in this case the maximum size of name is 100 chars
```



```
*/
pstmt.registerReturnParameter(2, OracleTypes.VARCHAR, 100);
// process the DML returning statement
count = pstmt.executeUpdate();
if (count>0)
{
    ResultSet rset = pstmt.getReturnResultSet(); //rest is not null and not empty
    while(rset.next())
    {
        String name = rset.getString(1);
        ...
    }
}
...
```

The following code example also illustrates the use of DML returning. However, in this case, the maximum size of the return parameters is not known. Therefore, the registerReturnParameter(int paramIndex, int externalType) method is used.

```
. . .
OraclePreparedStatement pstmt = (OraclePreparedStatement)conn.prepareStatement(
  "insert into lobtab values (100, empty clob()) returning col1, col2
into ?, ?");
// register return parameters
pstmt.registerReturnParameter(1, OracleTypes.INTEGER);
pstmt.registerReturnParameter(2, OracleTypes.CLOB);
// process the DML returning SQL statement
pstmt.executeUpdate();
ResultSet rset = pstmt.getReturnResultSet();
int r;
CLOB clob;
if (rset.next())
{
  r = rset.getInt(1);
 System.out.println(r);
 clob = (CLOB)rset.getClob(2);
}
. . .
```

4.6.4 Limitations of DML Returning

When using DML returning, be aware of the following:

- It is unspecified what the getReturnResultSet method returns when it is invoked more than once. You should not rely on any specific action in this regard.
- The ResultSet objects returned from the execution of DML returning statements do not support the ResultSetMetaData type. Therefore, the applications must know the information of return parameters before running DML returning statements.
- Streams are not supported with DML returning.
- DML returning cannot be combined with batch update.
- You cannot use both the auto-generated key feature and the DML returning feature in a single SQL DML statement. For example, the following is not allowed:



```
PreparedStatement pstmt = conn.prepareStatement('insert into orders (?, ?, ?)
returning order_id into ?");
pstmt.setInt(1, seq01.NEXTVAL);
pstmt.setInt(2, 100);
pstmt.setInt(3, 966431502);
pstmt.registerReturnParam(4, OracleTypes.INTEGER);
pstmt.executeUpdate;
ResultSet rset = pstmt.getGeneratedKeys;
....
```

4.7 Accessing PL/SQL Associative Arrays

Oracle JDBC drivers enable JDBC applications to make PL/SQL calls with Associative Arrays parameters. This section describes the methods to access Associative Arrays.

In PL/SQL, an Associative Array is a set of key-value pairs, where the keys may be PLS_INTEGERS or Strings. The keys may have any value and need not be dense. From a client application, you can work only with PLS_INTEGER or BINARY_INTEGER keys.

Note:

. . .

- Associative Arrays were previously known as index-by tables.
- The PLS INTEGER and BINARY INTEGER are identical data types.
- When you use String data types, the size is limited to the size in PL/SQL that is 32767 characters. For the server-side internal driver, the limits are lower.

The previous release of Oracle JDBC drivers provided support only for PL/SQL Associative Arrays of Scalar data types. Also, the support was restricted only to the values of the key-value pairs of the Arrays. Starting from Release 18c, Oracle Database supports accessing both the keys (indexes) and values of Associative Arrays, and also provides support for Associative Arrays of object types. Use the following methods to achieve the new functionalities:

Array createOracleArray(String arrayTypeName,

Object elements)

throws SQLException

ARRAY createARRAY(String typeName,

Object elements)

throws SQLException



Note:

You can use the createOracleArray method and the createARRAY method for Associative Arrays on Oracle Database release 12c and later releases. If you are using an earlier release of Oracle Database, then you should continue using the following deprecated APIs:

- oracle.jdbc.OraclePreparedStatement.setPlsqlIndexTable
- oracle.jdbc.OracleCallableStatement.getPlsqlIndexTable
- oracle.jdbc.OracleCallableStatement.getOraclePlsqlIndexTabl
 e
- oracle.jdbc.OracleCallableStatement.registerIndexTableOutPa rameter

In both the preceding methods, the second parameter can either be a java.util.Map<Integer, ?> that holds the key-value pairs of the Associative
Arrays, or it can only be an array of values. If it is an array of values, then the
JDBC driver defaults the indexes to 0,1,2 and so on. If it is
java.util.Map<Integer, ?>, then the JDBC driver does not default the keys.
They remain as specified in the Map, and can be sparse and negative.

Map<?, ?> oracle.jdbc.OracleArray.getJavaMap();

This method returns a Map<?, ?> for the data types in the Associative Array and null for Nested Tables and VARRAYs.

See Also:

- Oracle Database JDBC Java API Reference
- Oracle Database PL/SQL Language Reference for more information
 about Associative Arrays



Features Specific to JDBC Thin

This chapter introduces the Java Database Connectivity (JDBC) Thin client and covers the following features supported only by the JDBC Thin driver:

- Overview of JDBC Thin Client
- Additional Features Supported

5.1 Overview of JDBC Thin Client

The JDBC Thin client is a pure Java, Type IV driver. It is lightweight and easy to install. It provides high performance, comparable to the performance provided by the JDBC Oracle Call Interface (OCI) driver. The JDBC Thin driver is written entirely in Java, and therefore, it is platform-independent. Also, this driver does not require any additional Oracle software on the client-side.

The JDBC Thin driver communicates with the server using TTC, a protocol developed by Oracle to access data from Oracle Database. It can be used for application servers. The driver allows a direct connection to the database by providing an implementation of TCP/IP that implements Oracle Net and TTC on top of Java sockets. Both of these protocols are lightweight implementation versions of their counterparts on the server. The Oracle Net protocol runs over TCP/IP only.

The JDBC Thin driver can be used on both the client-side and the server-side. On the clientside, drivers can be used in Java applications that run either on the client or in the middle tier of a three-tier configuration. On the server-side, this driver is used to access a remote Oracle Database instance or another session on the same database.

5.2 Additional Features Supported

The JDBC Thin driver supports all standard JDBC features. The JDBC Thin driver also provides support for the following additional features:

- Default Support for Native XA
- Support for Transaction Guard
- Support for Application Continuity

5.2.1 Default Support for Native XA

Similar to the JDBC OCI driver, the JDBC Thin driver also provides support for Native XA. However, the JDBC Thin driver provides support for Native XA by default. This is unlike the case of the JDBC OCI driver, in which the support for Native XA is not enabled by default.



See Also: "Native-XA in Oracle JDBC Drivers"

5.2.2 Support for Transaction Guard

Transaction Guard feature provides a generic infrastructure for at-most-once execution during planned and unplanned outages and duplicate submissions. Transaction Guard feature (along with Application Continuity feature) provides transparent session recovery and replay of SQL statements (queries and DMLs) since the beginning of the in-flight transaction.



5.2.3 Support for Application Continuity

Application Continuity provides a general purpose, application-independent infrastructure that enables recovery of work from an application perspective, after the occurrence of a planned or unplanned outage. It provides the following benefits:

- Masking of outages from the end user
- · Recovery of user environments, in-flight transactions, and lost outcome
- A single, easy, and foolproof method for applications to recover
- A definite target response time for applications, regardless of outages

See Also:

Application Continuity for Java



6

Features Specific to JDBC OCI Driver

This chapter introduces the features specific to the Java Database Connectivity (JDBC) Oracle Call Interface (OCI) driver. It also describes the OCI Instant Client. This chapter contains the following sections:

- OCI Connection Pooling
- Transparent Application Failover
- OCI Native XA
- OCI Instant Client
- About Instant Client Light (English)

6.1 OCI Connection Pooling

The OCI connection pooling feature is an Oracle-designed extension. The connection pooling provided by the JDBC OCI driver enables applications to have multiple logical connections, all of which are using a small set of physical connections. Each call on a logical connection is routed on to the physical connection that is available at the given time.



6.2 Transparent Application Failover

The Transparent Application Failover feature of JDBC OCI driver enables you to automatically reconnect to a database if the database instance to which the connection is made goes down. The new database connection, though created by a different node, is identical to the original.

See Also: Transparent Application Failover

6.3 OCI Native XA

The JDBC OCI driver also provides a feature called Native XA. This feature enables to use native APIs to send XA commands. The native APIs provide high performance gains as compared to non-native APIs.



Related Topics

OCI Native XA

6.4 OCI Instant Client

The Instant Client facilitates the deployment OCI, Oracle C++ Call Interface (OCCI), Open Database Connectivity (ODBC), and JDBC-OCI based customer applications, by eliminating the need for an Oracle home.

The storage space requirement of a JDBC OCI application, using the Instant Client, is significantly reduced compared to the same application running on a full client-side installation as the Instant Client shared libraries occupy only about one-fourth of the disk space used by a full client installation.



6.5 About Instant Client Light (English)

The lightweight version of the Instant Client is called Instant Client Light (English). It is a significantly smaller version of the Instant Client, which reduces the disk space requirements of the client installation by about 63 MB.

See Also:

Installing Oracle Instant Client Basic Light



7 Server-Side Internal Driver

This chapter covers the following topics:

- Overview of the Server-Side Internal Driver
- Connecting to the Database
- About Session and Transaction Context
- Testing JDBC on the Server
- Loading an Application into the Server

7.1 Overview of the Server-Side Internal Driver

The server-side internal driver is intrinsically tied to Oracle Database and to the embedded Java Virtual Machine, also known as Oracle Java Virtual Machine (Oracle JVM). The driver runs as part of the same process as the Database. It also runs within the default session, the same session in which the Oracle JVM was started. Each Oracle JVM session has a single implicit native connection to the Database session in which it exists. This connection is conceptual and is not a Java object. It is an inherent aspect of the session and cannot be opened or closed from within the JVM.

The server-side internal driver is optimized to run within the database server and provide direct access to SQL data and PL/SQL subprograms on the local database. The entire JVM operates in the same address space as the database and the SQL engine. Access to the SQL engine is a function call. This enhances the performance of your Java Database Connectivity (JDBC) applications and is much faster than running a remote Oracle Net call to access the SQL engine.

The server-side internal driver supports the same features, application programming interfaces (APIs), and Oracle extensions as the client-side drivers. This makes application partitioning very straightforward. For example, if you have a Java application that is data-intensive, then you can easily move it into the database server for better performance, without having to modify the application-specific calls.

7.2 Connecting to the Database

As described in the preceding section, the server-side internal driver runs within a default session. Therefore, you are already connected. There are two methods to access the default connection:

- Use the OracleDataSource.getConnection method, with any of the following forms as the URL string:
 - jdbc:oracle:kprb
 - jdbc:default:connection
 - jdbc:oracle:kprb:
 - jdbc:default:connection:



• Use the Oracle-specific defaultConnection method of the OracleDriver class.

Using defaultConnection is generally recommended.

Note:

You are no longer required to register the OracleDriver class for connecting with the server-side internal driver.

Connecting with the OracleDriver Class defaultConnection Method

The defaultConnection method of the oracle.jdbc.OracleDriver class is an Oracle extension and always returns the same connection object. Even if you call this method multiple times, assigning the resulting connection object to different variable names, then only a single connection object is reused.

You need not include a connection string in the defaultConnection call. For example:

```
import java.sql.*;
import oracle.jdbc.*;
class JDBCConnection
{
    public static Connection connect() throws SQLException
    {
        Connection conn = null;
        try {
            // connect with the server-side internal driver
                conn = ora.defaultConnection();
        }
        } catch (SQLException e) {...}
        return conn;
    }
}
```

Note that there is no conn.close call in the example. When JDBC code is running inside the target server, the connection is an implicit data channel, not an explicit connection instance as from a client. It should *not* be closed.

OracleDriver has a static variable to store a default connection instance. The method OracleDriver.defaultConnection returns this default connection instance if the connection exists and is not closed. Otherwise, it creates a new, open instance and stores it in the static variable and returns it to the caller.

Typically, you should use the OracleDriver.defaultConnection method. This method is faster and uses less resources. Java stored procedures should be carefully written. For example, to close statements before the end of each call.

Typically, you should not close the default connection instance because it is a single instance that can be stored in multiple places, and if you close the instance, each would become unusable. If it is closed, a later call to the OracleDriver.defaultConnection method gets a new, open instance.



Connecting with the OracleDataSource.getConnection Method

To connect to the internal server connection from code that is running within the target server, you can use the <code>OracleDataSource.getConnection</code> method with either of the following URLs:

```
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:oracle:kprb");
Connection conn = ods.getConnection();
```

or:

```
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:default:connection");
Connection conn = ods.getConnection();
```

Any user name or password you include in the URL is ignored in connecting to the default server connection.

The OracleDataSource.getConnection method returns a new Java Connection object every time you call it. The fact that OracleDataSource.getConnection returns a new connection object every time you call it is significant if you are working with object maps or type maps. A type map is associated with a specific Connection object and with any state that is part of the object. If you want to use multiple type maps as part of your program, then you can call getConnection to create a new Connection object for each type map.

Note:

Although the OracleDataSource.getConnection method returns a new object every time you call it, it does not create a new database connection every time. They all utilize the same implicit native connection and share the same session state, in particular, the local transaction.

7.3 About Session and Transaction Context

The server-side driver operates within a default session and default transaction context. The default session is the session in which the JVM was started. In effect, you are already connected to the database on the server. This is different from the client-side where there is no default session. You must explicitly connect to the database.

Auto-commit mode is disabled in the server. You must manage transaction COMMIT and ROLLBACK operations explicitly by using the appropriate methods on the connection object:

```
conn.commit();
```

or:

conn.rollback();



Note:

As a best practice, it is recommended not to commit or rollback a transaction inside the server.

7.4 Testing JDBC on the Server

Almost any JDBC program that can run on a client can also run on the server. All the programs in the samples directory can be run on the server, with only minor modifications. Usually, these modifications concern only the connection statement.

Consider the following code fragment which obtains a connection to a database:

```
ods.setUrl(
"jdbc:oracle:oci:@(DESCRIPTION=
  (ADDRESS=(PROTOCOL=TCP)(HOST=cluster_alias)
      (PORT=5221))
      (CONNECT_DATA=(SERVICE_NAME=orcl)))");
ods.setUser("HR");
ods.setPassword("hr");
Connection conn = ods.getConnection();
```

We can modify this code fragment for use in the server-side internal driver. In the server-side internal driver, no user, password, or database information is necessary. For the connection statement, you use:

```
ods.setUrl(
"jdbc:oracle:kprb:@");
Connection conn = ods.getConnection();
```

However, the most convenient way to get a connection is to call the OracleDriver.defaultConnection method, as follows:

```
Connection conn = OracleDriver.defaultConnection();
```

7.5 Loading an Application into the Server

When loading an application into the server, you can load .class files that you have already compiled on the client or you can load .java source files and have them automatically compiled on the server.

7.5.1 Using the Loadjava Utility

You can use the loadjava utility to load your files. You can either specify source file names on the command line or put the files into a Java Archive (JAR) file and specify the JAR file name on the command line.

The loadjava script, which runs the actual utility, is in the bin directory in your Oracle home. This directory should already be in your path once Oracle has been installed.



Note: The loadjava utility supports compressed files.

Loading Class Files into the Server

Consider a case where you have the following three class files in your application: Fool.class, Foo2.class, and Foo3.class. Each class is written into its own class schema object in the server.

You can load the class files using the default JDBC Oracle Call Interface (OCI) driver in the following ways:

Specifying the individual class file names, as follows:

```
loadjava -user HR Fool.class Foo2.class Foo3.class Password: password
```

• Specifying the class file names using a wildcard, as follows:

```
loadjava -user HR Foo*.class
Password: password
```

Specifying a JAR file that contains the class files, as follows:

loadjava -user HR Foo.jar Password: *password*

You can load the files using the JDBC Thin driver, as follows:

```
loadjava -thin -user HR@localhost:5221:orcl Foo.jar
Password: password
```

Note:

Starting from Oracle Database 12c Release 1 (12.1), JDK 6, and JDK 7 are supported. However, only one of the JVMs will be active at a given time.

Ensure that your classes are not compiled using a newer version of JDK than the active runtime version on the server.

Loading Source Files into the Server

If you enable the <code>loadjava -resolve</code> option when loading a .java source file, then the server-side compiler will compile your application as it is loaded, resulting in both a source schema object for the original source code and one or more class schema objects for the compiled output.

If you do not specify -resolve, then the source is loaded into a source schema object without any compilation. In this case, however, the source is implicitly compiled the first time an attempt is made to use a class defined in the source.

For example, run loadjava as follows to load and compile Foo.java, using the default JDBC OCI driver:

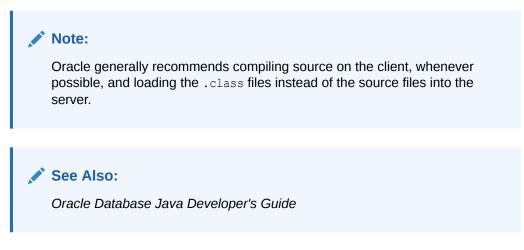


```
loadjava -user HR -resolve Foo.java
Password: password
```

Or, use the following command to load using the JDBC Thin driver:

```
loadjava -thin -user HR@localhost:5221:orcl -resolve Foo.java Password: password
```

Either of these will result in appropriate class schema objects being created in addition to the source schema object.



7.5.2 Using the JVM Command Line

You can also use the JVM command-line option to load your files. The command-line interface to Oracle JVM is analogous to using the JDK or JRE shell commands. You can:

- Use the standard -classpath syntax to indicate where to find the classes to load
- Set the system properties by using the standard -D syntax

The interface is a PL/SQL function that takes a string (VARCHAR2) argument, parses it as a command-line input and if it is properly formed, runs the indicated Java method in Oracle JVM. To do this, PL/SQL package DBMS JAVA provides the following functions:

• runjava

You can use the runjava function in the following way:

FUNCTION runjava(cmdline VARCHAR2) RETURN VARCHAR2;

runjava_in_current_session

You can use the runjava in current session function in the following way:

FUNCTION runjava_in_current_session(cmdline VARCHAR2) RETURN VARCHAR2;



Note:

Starting with Oracle Database 11*g* Release 1, there is a just-in-time (JIT) compiler for Oracle JVM environment. A JIT compiler for Oracle JVM enables much faster execution because the JIT compiler uses advanced techniques as compared to the old Native compiler and compiles dynamically generated code. Unlike the old Native compiler, the JIT compiler does not require a C compiler. It is enabled without the support of any plug-ins.



Part III

Connection and Security

This part consists of chapters that discuss the use of data sources and URLs to connect to the database. It also includes chapters that discuss the security features supported by the Oracle Java Database Connectivity (JDBC) Oracle Call Interface (OCI) and Thin drivers, TLS (Transport Layer Security) support in JDBC Thin driver, and middle-tier authentication through proxy connections.

Part III contains the following chapters:

- Data Sources and URLs
- JDBC Client-Side Security Features
- JDBC Service Provider Extensions
- Proxy Authentication



8 Data Sources and URLs

This chapter discusses connecting applications to databases using Java Database Connectivity (JDBC) data sources, as well as the URLs that describe databases. This chapter contains the following sections:

- About Data Sources
- Database URLs and Database Specifiers

8.1 About Data Sources

Data sources are standard, general-use objects for specifying databases or other resources to use. The JDBC 2.0 extension application programming interface (API) introduced the concept of data sources. For convenience and portability, data sources can be bound to Java Naming and Directory Interface (JNDI) entities, so that you can access databases by logical names.

The data source facility provides a complete replacement for the previous JDBC DriverManager facility. You can use both facilities in the same application, but it is recommended that you transition your application to data sources.

This section covers the following topics:

- Overview of Oracle Data Source Support for JNDI
- Features and Properties of Data Sources
- Creating a Data Source Instance and Connecting
- Creating a Data Source Instance_ Registering with JNDI_ and Connecting
- Supported Connection Properties
- About Using Roles for SYS Login
- Configuring Database Remote Login
- Using Bequeath Connection and SYS Logon
- Setting Properties for Oracle Performance Extensions
- Support for Network Data Compression

8.1.1 Overview of Oracle Data Source Support for JNDI

The JNDI standard provides a way for applications to find and access remote services and resources. These services can be any enterprise services. However, for a JDBC application, these services would include database connections and services.

JNDI enables an application to use logical names in accessing these services, removing vendor-specific syntax from application code. JNDI has the functionality to associate a logical name with a particular source for a desired service.



All Oracle JDBC data sources are JNDI-referenceable. The developer is not required to use this functionality, but accessing databases through JNDI logical names makes the code more portable.

Note:

Using JNDI functionality requires the jndi.jar file to be in the CLASSPATH environment variable. This file is included with the Java products on the installation CD. You must add it to the CLASSPATH environment variable separately.

8.1.2 Features and Properties of Data Sources

By using the data source functionality with JNDI, you do not need to register the vendor-specific JDBC driver class name and you can use logical names for URLs and other properties. This ensures that the code for opening database connections is portable to other environments.

The DataSource Interface and Oracle Implementation

A JDBC data source is an instance of a class that implements the standard javax.sql.DataSource interface:

```
public interface DataSource
{
   Connection getConnection() throws SQLException;
   Connection getConnection(String username, String password)
      throws SQLException;
   ...
}
```

Oracle implements this interface with the OracleDataSource class in the oracle.jdbc.pool package. The overloaded getConnection method returns a connection to the database.

To use other values, you can set properties using appropriate setter methods. For alternative user names and passwords, you can also use the getConnection method that takes these parameters as input. This would take priority over the property settings.

Note:

The OracleDataSource class and all subclasses implement the java.io.Serializable and javax.naming.Referenceable interfaces.

Properties of DataSource

The OracleDataSource class, as with any class that implements the DataSource interface, provides a set of properties that can be used to specify a database to connect to. These properties follow the JavaBeans design pattern.



The following tables list the OracleDataSource standard properties and Oracle extensions respectively.

Note: Oracle does not implement the standard roleName property.

Table 8-1 Standard Data Source Properties

Name	Туре	Description
databaseName	String	Name of the particular database on the server.
dataSourceName	String	Name of the underlying data source class. For connection pooling, this is an underlying pooled connection data source class. For distributed transactions, this is an underlying XA data source class.
description	String	Description of the data source.
networkProtocol	String	Network protocol for communicating with the server. For Oracle, this applies only to the JDBC Oracle Call Interface (OCI) drivers and defaults to tcp.
password	String	Password for the connecting user.
portNumber	int	Number of the port where the server listens for requests
serverName	String	Name of the database server
user	String	User name to log in

Note:

For security reasons, there is no getPassword() method.

 Table 8-2
 Oracle Extended Data Source Properties

Name	Туре	Description
connectionCacheName	String	Specifies the name of the cache. This cannot be changed after the cache has been created.
connectionCacheProperties	java.util.P roperties	Specifies properties for implicit connection cache.
connectionCachingEnabled	Boolean	Specifies whether implicit connection cache is in use.
connectionProperties	java.util.P roperties	Specifies the connection properties.
driverType	String	Specifies Oracle JDBC driver type. It can be one of oci, thin, or kprb.
fastConnectionFailoverEnable d	Boolean	Specifies whether Fast Connection Failover is in use.
implicitCachingEnabled	Boolean	Specifies whether the implicit statement connection cache is enabled.



Name	Туре	Description
loginTimeout	int	Specifies the maximum time in seconds that this data source will wait while attempting to connect to a database.
logWriter	java.io.Pri ntWriter	Specifies the log writer for this data source.
maxStatements	int	Specifies the maximum number of statements in the application cache.
serviceName	String	Specifies the database service name for this data source.
tnsEntry	String	Specifies the TNS entry name. The TNS entry name corresponds to the TNS entry specified in the tnsnames.ora configuration file.
		Enable this OracleXADataSource property when using the Native XA feature with the OCI driver, to access Oracle pre-8.1.6 databases and later. If the tnsEntry property is not set when using the Native XA feature, then a SQLException with error code ORA-17207 is thrown
url	String	Specifies the URL of the database connection string. Provided as a convenience, it can help you migrate from an older Oracle Database. You can use this property in place of the Oracle tnsEntry and driverType properties and the standard portNumber, networkProtocol, serverName, and databaseName properties.
nativeXA	Boolean	Allows an OracleXADataSource using the Native XA feature with the OCI driver, to access Oracle pre-8.1.6 databases and later. If the nativeXA property is enabled, be sure to set the tnsEntry property as well. This property is only for OracleXADatasource.
		This DataSource property defaults to false.
ONSConfiguration	String	Specifies the ONS configuration string that is used to remotely subscribe to FAN/ONS events.

Table 8-2 (Cont.) Oracle Extended Data Source Properties

Note:

- This table omits properties that supported the deprecated connection cache based on OracleConnectionCache.
- Because Native XA performs better than Java XA, use Native XA whenever possible.



Use the setConnectionProperties method to set the properties of the connection and the setConnectionCacheProperties method to set the properties of the connection cache.

If you are using the server-side internal driver, that is, the driverType property is set to kprb, then any other property settings are ignored.

If you are using the JDBC Thin or OCI driver, then note the following:

• A URL setting can include settings for user and password, as in the following example, in which case this takes precedence over individual user and password property settings:

jdbc:oracle:thin:HR/<password>@localhost:5221:orcl

- Settings for user and password are required, either directly through the URL setting or through the getConnection call. The user and password settings in a getConnection call take precedence over any property settings.
- If the url property is set, then any tnsEntry, driverType, portNumber, networkProtocol, serverName, and databaseName property settings are ignored.
- If the tnsEntry property is set, which presumes the url property is not set, then any databaseName, serverName, portNumber, and networkProtocol settings are ignored.
- If you are using an OCI driver, which presumes the driverType property is set to oci, and the networkProtocol is set to ipc, then any other property settings are ignored.

Also, note that getConnectionCacheName() will return the name of the cache only if the ConnectionCacheName property of the data source is set after caching is enabled on the data source.

8.1.3 Creating a Data Source Instance and Connecting

This section shows an example of the most basic use of a data source to connect to a database, without using JNDI functionality. Note that this requires vendor-specific, hard-coded property settings.

Create an OracleDataSource instance, initialize its connection properties as appropriate, and get a connection instance, as in the following example:

```
OracleDataSource ods = new OracleDataSource();
ods.setDriverType("oci");
ods.setServerName("localhost");
ods.setNetworkProtocol("tcp");
ods.setDatabaseName(<database_name>);
ods.setPortNumber(5221);
ods.setUser("HR");
ods.setPassword("hr");
Connection conn = ods.getConnection();
```

Or, optionally, override the user name and password, as follows:

```
Connection conn = ods.getConnection("OE", "oe");
```

8.1.4 Creating a Data Source Instance, Registering with JNDI, and Connecting

This section exhibits JNDI functionality in using data sources to connect to a database. Vendor-specific, hard-coded property settings are required only in the portion of code that



binds a data source instance to a JNDI logical name. From that point onward, you can create portable code by using the logical name in creating data sources from which you will get your connection instances.



Initialize Data Source Properties

Create an OracleDataSource instance, and then initialize its properties as appropriate, as in the following example:

```
OracleDataSource ods = new OracleDataSource();
ods.setDriverType("oci");
ods.setServerName("localhost");
ods.setNetworkProtocol("tcp");
ods.setDatabaseName("816");
ods.setPortNumber(5221);
ods.setUser("HR");
ods.setPassword("hr");
```

Register the Data Source

Once you have initialized the connection properties of the OracleDataSource instance ods, as shown in the preceding example, you can register this data source instance with JNDI, as in the following example:

```
Context ctx = new InitialContext();
ctx.bind("jdbc/sampledb", ods);
```

Calling the JNDI InitialContext() constructor creates a Java object that references the initial JNDI naming context. System properties, which are not shown, instruct JNDI which service provider to use.

The ctx.bind call binds the OracleDataSource instance to a logical JNDI name. This means that anytime after the ctx.bind call, you can use the logical name jdbc/ sampledb in opening a connection to the database described by the properties of the OracleDataSource instance ods. The logical name jdbc/sampledb is logically bound to this database.

The JNDI namespace has a hierarchy similar to that of a file system. In this example, the JNDI name specifies the subcontext jdbc under the root naming context and specifies the logical name sampledb within the jdbc subcontext.

The Context interface and InitialContext class are in the standard javax.naming package.



Note:

The JDBC 2.0 Specification requires that all JDBC data sources be registered in the jdbc naming subcontext of a JNDI namespace or in a child subcontext of the jdbc subcontext.

Open a Connection

To perform a lookup and open a connection to the database logically bound to the JNDI name, use the logical JNDI name. Doing this requires casting the lookup result, which is otherwise a Java Object, to OracleDataSource and then using its getConnection method to open the connection.

Here is an example:

```
OracleDataSource odsconn = (OracleDataSource)ctx.lookup("jdbc/sampledb");
Connection conn = odsconn.getConnection();
```

8.1.5 Supported Connection Properties

For a detailed list of connection properties that Oracle JDBC drivers support, see the Oracle Database JDBC Java API Reference.

Note:

The JDBC Thin Driver connection property oracle.jdbc.ReadTimeout and the JDBC method java.sql.Connection.setNetworkTimeout may not behave as expected, when Dead Connection Detection (DCD) is enabled in the old way in Oracle Database. See the following My Oracle Support Note for more information about the new implementation of DCD: https://support.oracle.com/rs? type=doc&id=1591874.1

8.1.6 About Using Roles for SYS Login

To specify the role for the SYS login, use the internal_logon connection property. To log on as SYS, set the internal_logon connection property to SYSDBA or SYSOPER.

Note:

The ability to specify a role is supported only for the $\ensuremath{\mathtt{sys}}$ user name.

For a bequeath connection, we can get a connection as SYS by setting the internal_logon property. For a remote connection, we need additional password file setting procedures.



8.1.7 Configuring Database Remote Login

Before the JDBC Thin driver can connect to the database as SYSDBA, you must configure the user, because Oracle Database security system requires a password file for remote connections as an administrator. Perform the following:

- 1. Set a password file on the server-side or on the remote database, using the orapwd password utility. You can add a password file for user SYS as follows:
 - In UNIX

```
orapwd file=$ORACLE_HOME/dbs/orapwORACLE_SID entries=200
Enter password: password
```

In Microsoft Windows

```
orapwd file=%ORACLE_HOME%\database\PWDORACLE_SID.ora entries=200
Enter password: password
```

In this case, file is the name of the password file, *password* is the password for user SYS. It can be altered using the ALTER USER statement in SQL Plus. You should set entries to a value higher than the number of entries you expect.

The syntax for the password file name is different on Microsoft Windows and UNIX.

See Also: Oracle Database Administrator's Guide

2. Enable remote login as SYSDBA. This step grants SYSDBA and SYSOPER system privileges to individual users and lets them connect as themselves.

Stop the database, and add the following line to initservice_name.ora, in UNIX, or init.ora, in Microsoft Windows:

remote_login_passwordfile=exclusive

The initservice_name.ora file is located at ORACLE_HOME/dbs/ and also at ORACLE_HOME/admin/db_name/pfile/. Ensure that you keep the two files synchronized.

The init.ora file is located at <code>%ORACLE_BASE%\ADMIN\db_name\pfile\.</code>

3. Change the password for the SYS user. This is an optional step.

```
PASSWORD sys
```

```
Changing password for sys
New password: password
Retype new password: password
```

4. Verify whether SYS has the SYSDBA privilege.

SQL> select * from v\$pwfile_user	s;	
USERNAME	SYSDB	SYSOP

5. Restart the remote database.



Example 8-1 Using SYS Login To Make a Remote Connection

```
//This example works regardless of language settings of the database.
 /** case of remote connection using sys **/
import java.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.*;
// create an OracleDataSource
OracleDataSource ods = new OracleDataSource();
// set connection properties
java.util.Properties prop = new java.util.Properties();
prop.put("user", "sys");
prop.put("password", "sys");
prop.put("internal logon", "sysoper");
ods.setConnectionProperties(prop);
//\ {\rm set} the url
\ensuremath{{\prime}}\xspace // the url can use oci driver as well as:
// url = "jdbc:oracle:oci8:@remotehost"; the remotehost is a remote database
String url = "jdbc:oracle:thin:@localhost:5221/orcl";
ods.setURL(url);
// get the connection
Connection conn = ods.getConnection();
. . .
```

8.1.8 Using Bequeath Connection and SYS Logon

The following example illustrates how to use the internal_logon and SYSDBA arguments to specify the SYS login. This example works regardless of the database's national-language settings of the database.

```
/** Example of bequeath connection **/
import java.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.*;
// create an OracleDataSource instance
OracleDataSource ods = new OracleDataSource();
// set neccessary properties
java.util.Properties prop = new java.util.Properties();
prop.put("user", "sys");
prop.put("password", "sys");
prop.put("internal logon", "sysdba");
ods.setConnectionProperties(prop);
// the url for bequeath connection
String url = "jdbc:oracle:oci8:@";
ods.setURL(url);
// retrieve the connection
Connection conn = ods.getConnection();
. . .
```

8.1.9 Setting Properties for Oracle Performance Extensions

Some of the connection properties are for use with Oracle performance extensions. Setting these properties is equivalent to using corresponding methods on the OracleConnection object, as follows:



- Setting the defaultRowPrefetch property is equivalent to calling setDefaultRowPrefetch.
- Setting the remarksReporting property is equivalent to calling setRemarksReporting.



Example

The following example shows how to use the put method of the java.util.Properties class, in this case, to set Oracle performance extension parameters.

```
//import packages and register the driver
import java.sql.*;
import java.math.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.OracleDataSource;
//specify the properties object
java.util.Properties info = new java.util.Properties();
info.put ("user", "HR");
info.put ("password", "hr");
info.put ("defaultRowPrefetch","20");
info.put ("defaultBatchValue", "5");
//specify the datasource object
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:oracle:thin:@localhost:5221/orcl");
ods.setUser("HR");
ods.setPassword("hr");
ods.setConnectionProperties(info);
```

...

8.1.10 Support for Network Data Compression

Starting from Oracle Database 12c Release 2 (12.2.0.1), the JDBC Thin driver supports network data compression. Network data compression reduces the size of the session data unit (SDU) transmitted over a data connection and reduces the time required to transmit a SQL query and the result across the network. The benefits are more significant in case of Wireless Area Network (WAN). For enabling network data compression, you must set the connection properties in the following way:



Note: Network compression does not work for streamed data. ... OracleDataSource ds = new OracleDataSource(); Properties prop = new Properties(); prop.setProperty("user", "user1"); prop.setProperty("password", <password>); // Enabling Network Compression prop.setProperty("oracle.net.networkCompression", "on"); //Optional configuration for setting the client compression threshold. prop.setProperty("oracle.net.networkCompressionThreshold", "1024"); ds.setConnectionProperties(prop); ds.setURL(url); Connection conn = ds.getConnection(); ...

8.2 Database URLs and Database Specifiers

Database URLs are strings. The complete URL syntax is:

```
jdbc:oracle:driver_type:[username/password]@database_specifier
```

Note:

- The brackets indicate that the username/password pair is optional.
- kprb, the internal server-side driver, uses an implicit connection. Database URLs for the server-side driver end after the *driver type*.

The first part of the URL specifies which JDBC driver is to be used. The supported *driver type* values are thin, oci, and kprb.

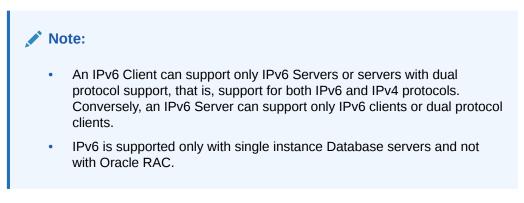
The remainder of the URL contains an optional user name and password separated by a slash, an @, and the database specifier, which uniquely identifies the database to which the application is connected. Some database specifiers are valid only for the JDBC Thin driver, some only for the JDBC OCI driver, and some for both.

8.2.1 Support for Internet Protocol Version 6

This release of Oracle JDBC drivers supports Internet Protocol Version 6 (IPv6) addresses in the JDBC URL and machine names that resolve to IPv6 addresses. IPv6 is a new Network layer protocol designed by the Internet Engineering Task Force (IETF) to replace the current version of Internet Protocol, Internet Protocol Version 4 (IPv4). The primary benefit of IPv6 is



a large address space, derived from the use of 128-bit addresses. IPv6 also improves upon IPv4 in areas such as routing, network auto configuration, security, quality of service, and so on.



If you want to use a literal IPv6 address in a URL, then you should enclose the literal address enclosed in a left bracket ([) and a right bracket (]). For example: [2001:0db8:7654:3210:FEDC:BA98:7654:3210]. So, a JDBC URL, using an IPv6 address will look like the following:

```
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)
  (HOST=[2001:0db8:7654:3210:FEDC:BA98:7654:3210])(PORT=5521))
  (CONNECT_DATA=(SERVICE_NAME=sales.example.com))
```

Note:

All the new System classes that are required for IPv6 support are loaded when Java is enabled during database initialization. So, if your application does not have any IPv6 addressing, then you do not need to change your code to use IPv6 functionality. However, if your application has either IPv6 only or both IPv6 and IPv4 addressing, then you should set the java.net.preferIPv6Addresses system property in the command line. This enables the Oracle JVM to load appropriate libraries. These libraries are loaded once and cannot be reloaded without restarting the Java process.

8.2.2 Support for HTTPS Proxy Configuration

Oracle Database Release 18c JDBC drivers support HTTPS Proxy Configuration. HTTPS Proxy enables tunnelling secure connections over forward HTTP proxy using the HTTP CONNECT method. This helps in accessing the public cloud database service as it eliminates the requirement to open an outbound port on a client side firewall. This parameter is applicable only to the connect descriptors where PROTOCOL=TCPS. This is similar to the web browser setting for intranet users who want to connect to internet hosts.

For configuring HTTPS Proxy, add details to the ADDRESS part of the Connection String as shown in the following code snippet:

(DESCRIPTION=



```
(ADDRESS=(HTTPS_PROXY=sales-proxy)(HTTPS_PROXY_PORT=8080)(PROTOCOL=TCPS)(HOST=sales2-svr)(PORT=443))
```

```
(CONNECT DATA=(SERVICE NAME=sales.us.example.com)))
```

8.2.3 Database Specifiers

Table 8-3, shows the possible database specifiers, listing which JDBC drivers support each specifier.

Specifier	Supported Drivers	Example
Oracle Net	Thin, OCI	Thin, using an address list:
connection descriptor		<pre>url="jdbc:oracle:thin:@(DESCRIPTION= (LOAD_BALANCE=on) (ADDRESS_LIST= (ADDRESS=(PROTOCOL=TCP)(HOST=host1) (PORT=5221)) (ADDRESS=(PROTOCOL=TCP)(HOST=host2)(PORT=5221))) (CONNECT_DATA=(SERVICE_NAME=orc1)))"</pre>
		OCI, using a cluster:
		"jdbc:oracle:oci:@(DESCRIPTION= (ADDRESS=(PROTOCOL=TCP)(HOST=cluster_alias) (PORT=5221)) (CONNECT_DATA=(SERVICE_NAME=orcl)))"
Thin-style service	Thin	Refer to "Thin-style Service Name Syntax" for details.
name		"jdbc:oracle:thin:HR/ <password>@localhost:5221/orcl"</password>
LDAP syntax	Thin	Refer to LDAP Syntax for details.
		"jdbc:oracle:thin:@ldap:ldap.example.com:7777/ sales,cn=OracleContext,dc=com"
Bequeath	OCI	Empty. That is, nothing after @
connection		"jdbc:oracle:oci:HR/ <password>/@"</password>
TNSNames alias	Thin, OCI	Refer to "TNSNames Alias Syntax" for details.
		<pre>OracleDataSource ods = new OracleDataSource(); ods.setTNSEntryName("MyTNSAlias");</pre>

Table 8-3 Supported Database Specifiers

8.2.4 Thin-style Service Name Syntax

Thin-style service names are supported only by the JDBC Thin driver. The syntax is:

@host_name:port_number/service_name

For example:

jdbc:oracle:thin:HR/<password>@localhost:5221/orcl

ORACLE

Note:

The JDBC Thin driver supports only the TCP/IP protocol.

8.2.5 Support for Easy Connect Plus

Oracle Database Release 19c JDBC driver introduced enhancements to the EasyConnect URL with many capabilities and this new enhanced Easy Connection URL is called Easy Connect Plus.

The EasyConnect URL supported simplified connection strings and the TCP transfer protocol. The goal of the enhancement to the EasyConnect URL is to simplify the client-side deployment and configuration, while connecting to the database. This enhancement can eliminate the need of using the tnsnames.ora file and you do not need to specify the TNS_ADMIN environment variable. The main features of Easy Connect Plus are the following:

- Support for TCPS Protocol
- Support for Multiple Host and Port
- Support to Pass Connection Properties in the Connection String

8.2.5.1 Support for TCPS Protocol

The EasyConnect URL supported only the TCP transport protocol. However, the Easy Connect Plus supports both TCP and TCPS protocols. This enhancement simplifies the client configurations to Oracle Database Cloud Services that mandate TCPS connections.

For example, if you have a EasyConnect URL in the following format:

```
jdbc:oracle:thin:@(DESCRIPTION=
 (ADDRESS=(PROTOCOL=tcps)(HOST=salesserver1)(PORT=1521))
 (CONNECT DATA=(SERVICE NAME=sales.us.example.com)))
```

Then, the Easy Connect Plus URL will be:

jdbc:oracle:thin:@tcps:salesserver1:1521/sales.us.example.com

8.2.5.2 Support for Multiple Host and Port

Using Easy Connect Plus, you can specify a comma-separated list of hosts.

The EasyConnect URL allowed only single host name and single port. But, Easy Connect Plus allows specifying multiple hosts and multiple ports in the connection string, which can be used to connect to a database with load balancing turned on.

For example, if you want to specify two hosts with a single port number in the connection using the following EasyConnect URL:

jdbc:oracle:thin:@(DESCRIPTION=
(ADDRESS LIST= (LOAD BALANCE=ON) (ADDRESS=(PROTOCOL=tcp))



```
(HOST=salesserver1) (PORT=1521)) (ADDRESS=(PROTOCOL=tcp) (HOST=salesserver2)
(PORT=1521)))
(CONNECT DATA=(SERVICE NAME=sales.us.example.com)))
```

Then, the Easy Connect Plus URL will be the following:

```
jdbc:oracle:thin:@tcp:salesserver1,salesserver2:1521/sales.us.example.com
```

Again, if you want to specify multiple hosts as a list, where each list follows its own port number, using the following EasyConnect URL:

```
jdbc:oracle:thin:@(DESCRIPTION= (ADDRESS_LIST= (LOAD_BALANCE=ON)
(ADDRESS=(PROTOCOL=tcp) (HOST=salesserver1) (PORT=1521))
(ADDRESS=(PROTOCOL=tcp) (HOST=salesserver2) (PORT=1522))
(ADDRESS=(PROTOCOL=tcp) (HOST=salesserver3) (PORT=1522)))
(CONNECT DATA=(SERVICE NAME=sales.us.example.com)))
```

Then, the Easy Connect Plus URL will be the following:

```
jdbc:oracle:thin:@tcp:salesserver1:1521,salesserver2,salesserver3:1522/
sales.us.example.com
```

8.2.5.3 Support to Pass Connection Properties in the Connection String

Easy Connect Plus supports specification of connection properties as name-value pairs in the connection URL. After the ? delimiter, Easy Connect Plus supports a list of parameters as name-value pairs, which can be optionally specified.

Remember the following points while specifying the name-value pairs:

- Use a question mark sign (?) to indicate the start of the name-value pairs and an ampersand sign (δ) as a delimiter between each name-value pair.
- Specify the entire connection string as a single string.
- Use the backslash (\) escape character if there are any special characters in the value.
- Place white spaces within double-quotes if they are required as part of the value because leading and trailing white spaces are ignored within the parameter values.

The following table lists a few of the supported parameters:



Parameter Name	Old URL Example	New URL Example
wallet_location	<pre>jdbc:oracle:thin:@(DESC RIPTION= (ADDRESS=(PROTOCOL=tcps)(HOST=salesserver1) (PORT=1521)) (CONNECT_DATA=(SERVICE_ NAME=sales.us.example.c om))</pre>	<pre>jdbc:oracle:thin:@tcps: salesserver1.com:1521/ sales.us.example.com? wallet_location=/Users/ jsmith/DBCloudService/ wallet_JDBCTEST&oracle. net.ssl_server_cert_dn= \"CN=salesserver2.com.c om,OU=Oracle BMCS US,O=Oracle Corporation,L=Redwood City,ST=California,C=US \"";</pre>
ssl_server_dn_match		
ssl_server_cert_dn	<pre>jdbc:oracle:thin:@(DESC RIPTION= (ADDRESS=(PROTOCOL=tcps)(HOST=salesserver1) (PORT=1521))(SECURITY= (SSL_SERVER_DN_MATCH=TR UE) (SSL_SERVER_CERT_DN=cn= sales,cn=OracleContext, dc=us,dc=example,dc=com)) (CONNECT_DATA=(SERVICE_ NAME=sales.us.example.c om)))</pre>	<pre>jdbc:oracle:thin:@tcps: salesserver1:1521/ sales.us.example.com? ssl_server_cert_dn="cn= sales,cn=OracleContext, dc=us,dc=example,dc=com "</pre>
https_proxy and https_proxy_port	<pre>jdbc:oracle:thin:@(DESC RIPTION= (ADDRESS=(PROTOCOL=tcps)(HOST=salesserver1) (PORT=1521) (https_proxy=www- proxy.mycompany.com) (https_proxy_port=80)) (CONNECT_DATA=(SERVICE_ NAME=sales.us.example.c om)))</pre>	<pre>jdbc:oracle:thin:@tcps: salesserver1:1521/ sales.us.example.com? https_proxy=www- proxy.mycompany.com&htt ps_proxy_port=80</pre>
connect_timeout	<pre>jdbc:oracle:thin:@(DESC RIPTION= (retry_count=3) (connect_timeout=60) (transport_connect_time out=30) (ADDRESS=(PROTOCOL=tcp) (HOST=salesserver1) (PORT=1521)) (CONNECT_DATA=(SERVICE_ NAME=sales.us.example.c om)))</pre>	<pre>jdbc:oracle:thin:@tcps: salesserver1:1521/ sales.us.example.com? connect_timeout=60& transport_connect_timeo ut=30&retry_count=3</pre>



The list of supported keywords under DESCRIPTION is as follows:

- ENABLE
- FAILOVER
- LOAD BALANCE
- RECV_BUF_SIZE
- SEND_BUF_SIZE
- SDU
- SOURCE_ROUTE
- RETRY_COUNT
- RETRY_DELAY
- CONNECT_TIMEOUT
- TRANSPORT_CONNECT_TIMEOUT

Note:

When your application uses a SOCKS proxy to connect to Oracle Database (for example, a Bastion in the Oracle Cloud), then the value of the TRANSPORT CONNECT TIMEOUT parameter is ignored.

8.2.6 Support for Delay in Connection Retries

Starting from Oracle Database 12c Release 1 (12.1.0.2), there is a new connection attribute RETRY_DELAY, which specifies the delay between connection retries in seconds. The following code snippet shows how to use this attribute:

```
(DESCRIPTION_LIST=
 (DESCRIPTION=
  (CONNECT_TIMEOUT=10) (RETRY_COUNT=3) (RETRY_DELAY=3)
  (ADDRESS_LIST=
   (ADDRESS=(PROTOCOL=tcp) (HOST=myhost1) (PORT=1521)))
  (ADDRESS=(PROTOCOL=tcp) (HOST=myhost2) (PORT=1521)))
  (CONNECT_DATA=(SERVICE_NAME=example1.com)))
  (DESCRIPTION=
   (CONNECT_TIMEOUT=60) (RETRY_COUNT=1) (RETRY_DELAY=5)
   (ADDRESS_LIST=
   (ADDRESS=(PROTOCOL=tcp) (HOST=myhost3) (PORT=1521))
   (ADDRESS=(PROTOCOL=tcp) (HOST=myhost4) (PORT=1521)))
   (CONNECT_DATA=(SERVICE_NAME=example2.com))))
```

8.2.7 TNSNames Alias Syntax

You can find the available TNSNAMES entries listed in the tnsnames.ora file on the client computer from which you are connecting. On Windows, this file is located in the ORACLE_HOME\NETWORK\ADMIN directory. On UNIX systems, you can find it in the ORACLE_HOME directory or the directory indicated in your TNS ADMIN environment variable.



For example, if you want to connect to the database on host myhost as user HR with password hr that has a TNSNAMES entry of MyHostString, then write the following:

```
OracleDataSource ods = new OracleDataSource();
ods.setTNSEntryName("MyTNSAlias");
ods.setUser("HR");
ods.setPassword("hr");
ods.setDriverType("oci");
Connection conn = ods.getConnection();
```

The oracle.net.tns_admin system property must be set to the location of the tnsnames.ora file so that the JDBC Thin driver can locate the tnsnames.ora file. For example:

```
System.setProperty("oracle.net.tns_admin", "c:\\Temp");
String url = "jdbc:oracle:thin:@tns entry";
```

Note:

When using TNSNames with the JDBC Thin driver, you must set the oracle.net.tns_admin property to the directory that contains your tnsnames.ora file.

java -Doracle.net.tns admin=\$ORACLE HOME/network/admin

8.2.8 LDAP Syntax

This section describes the database specifiers that you can use with LDAP servers.

An example of database specifier using the Lightweight Directory Access Protocol (LDAP) syntax is as follows:

"jdbc:oracle:thin:@ldap:ldap.example.com:7777/sales,cn=OracleContext,dc=com"

When using SSL, change this syntax to the following:

"jdbc:oracle:thin:@ldaps:ldap.example.com:7777/sales,cn=OracleContext,dc=com"

Note:

The JDBC Thin driver can use LDAP over SSL to communicate with Oracle Internet Directory if you substitute ldap: with ldaps: in the database specifier. The LDAP server must be configured to use SSL. If it is not, then the connection attempt will hang.

The JDBC Thin driver supports failover of a list of LDAP servers during the service name resolution process, without the need for a hardware load balancer. Also, client-side load balancing is supported for connecting to LDAP servers. A list of space separated LDAP URLs syntax is used to support failover and load balancing.

When a list of LDAP URLs is specified, both failover and load balancing are enabled by default. You can use the <code>oracle.net.ldap loadbalance</code> connection property to



disable load balancing, and the oracle.net.ldap_failover connection property to disable failover.

The following example shows failover with client-side load balancing disabled:

```
Properties prop = new Properties();
String url = "jdbc:oracle:thin:@ldap:ldap1.example.com:3500/
cn=salesdept,cn=OracleContext,dc=com/salesdb" +
"ldap:ldap2.example.com:3500/cn=salesdept,cn=OracleContext,dc=com/salesdb" +
"ldap:ldap3.example.com:3500/cn=salesdept,cn=OracleContext,dc=com/salesdb";
prop.put("oracle.net.ldap_loadbalance", "OFF" );
OracleDataSource ods = new OracleDataSource();
ods.setURL(url);
ods.setConnectionProperties(prop);
```

The JDBC Thin driver supports LDAP nonanonymous bind. A set of JNDI environment properties, which contains authentication information, can be specified for a data source. If an LDAP server is configured as not to allow anonymous bind, then you must provide the authentication information to connect to the LDAP server. The following example shows a simple clear-text password authentication:

```
String url = "jdbc:oracle:thin:@ldap:ldap.example.com:7777/
sales,cn=salesdept,cn=OracleContext,dc=com";
```

```
Properties prop = new Properties();
prop.put("oracle.net.ldap.security.authentication", "simple");
prop.put("oracle.net.ldap.security.principal", "cn=salesdept,cn=OracleContext,dc=com");
prop.put("oracle.net.ldap.security.credentials", "mysecret");
```

```
OracleDataSource ods = new OracleDataSource();
ods.setURL(url);
ods.setConnectionProperties(prop);
```

In the preceding example, as JDBC passes down the three properties to JNDI, the authentication mechanism chosen by client is consistent with how these properties are interpreted by JNDI. For example, if the client specifies authentication information without explicitly specifying the java.naming.security.authentication property, then the default authentication mechanism is "simple".

Starting from Oracle Database Release 21c, you can specify the location of the wallet that the driver uses for SSL negotiation with the LDAP server. You can add the user name and the password used for authentication to the wallet secret store.

See Also: JDBC Java API Reference



9 JDBC Client-Side Security Features

This chapter discusses support for IAM authentication for Autonomous Database, login authentication, network encryption and integrity with respect to features of the Oracle Advanced Security options in the JDBC OCI and the JDBC Thin drivers.

Note:

- This discussion is not relevant to the server-side internal driver because all communication through server-side internal driver is completely internal to the server.
- Using SHA-1 (Secure Hash Algorithm 1) with the parameters SQLNET.CRYPTO_CHECKSUM_TYPES_CLIENT and SQLNET.CRYPTO_CHECKSUM_TYPES_SERVER is deprecated in this release, and can be desupported in a future release. Using SHA-1 ciphers with DBMS_CRYPTO is also deprecated (HASH_SH1, HMAC_SH1). Instead of using SHA1, Oracle recommends that you start using a stronger SHA-2 cipher in place of the SHA-1 cipher.

Oracle Advanced Security, previously known as the Advanced Networking Option (ANO) or Advanced Security Option (ASO), provides industry standards-based network encryption, network integrity, third-party authentication, single sign-on, and access authorization. Both the JDBC OCI and JDBC Thin drivers support all the Oracle Advanced Security features.



Note:

If you want to use the security policy file for JDBC <code>ojdbc.policy</code>, then you can download the file from the following link:

http://www.oracle.com/technetwork/index.html

The ojdbc.policy file contains the granted permissions that you need to run your application in control environment of the Java Security Manager. You can either use this file itself as your Java policy file, or get contents from this file and add the content in your Java policy file. This file contains permissions like:

- A few mandatory permissions that are always required, for example, permission java.util.PropertyPermission "user.name", "read";
- A few driver-specific permissions, for example, JDBC OCI driver needs permission java.lang.RuntimePermission "loadLibrary.ocijdbc12";
- A few feature-based permissions, for example, permissions related to XA, XDB, FCF and so on

You can set the system properties mentioned in the file or direct values for permissions as per your requirement.

This chapter contains the following sections:

- Support for Token-Based Authentication for IAM
- Support for Token-Based Authentication for Azure AD
- Support for Oracle Advanced Security
- Support for Login Authentication
- Support for Strong Authentication
- Support for Network Encryption and Integrity
- Support for TLS
- Support for Kerberos
- Support for RADIUS
- About Secure External Password Store

9.1 Support for Token-Based Authentication for IAM

In Oracle Database release 23c, the JDBC Thin drivers provide enhanced support for Oracle Cloud Infrastructure (OCI) Identity Access Management (IAM).

While connecting to the database, the JDBC application provides a token to the database. The database verifies the token with a public key that it requests from the authentication service, and retrieves the corresponding user group membership information to find the database schema and role mappings to complete the user authorization to the database.

Additionally, the application sends a signed header, which proves that it possesses a private key that is paired to a public key embedded in the token. If both the token and



the signature are valid, and there exists a mapping between the IAM user and a database user, then access to the database is granted to the JDBC application.

The token-based authentication is supported in the following ways:

9.1.1 Using the File System

When a database token is available on the file system, for example, if you use the Oracle Cloud Infrastructure Command-Line Interface (OCI CLI), then you can configure the JDBC driver to use this token for connecting to the Database.

You can use the CONNECTION_PROPERTY_TOKEN_AUTHENTICATION for this purpose, which can be specified in the following ways:

- As an ojdbc.properties file
- As a JVM system property
- As a parameter in the query section of a connection string
- With a Properties object passed to OracleDataSource.setConnectionProperties (Properties)
- As a parameter in the SECURITY section of an Oracle Net Descriptor

JDBC supports the following two types of connection strings, where you can specify this parameter:

• As an Oracle Net descriptor, where you can specify the parameter as:

```
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(HOST=dbhost)(PORT=1522)
(PROTOCOL=tcps))(SECURITY=(SSL_SERVER_DN_MATCH=ON)
(TOKEN AUTH=OCI TOKEN))(CONNECT DATA=(SERVICE NAME=my.example.com)))
```

• As a connection property, where you can specify the parameter as:

```
jdbc:oracle:thin:@tcps:dbhost:1522/my.example.com?
oracle.jdbc.tokenAuthentication=OCI_TOKEN&oracle.jdbc.tokenLocation
="/path/to/my/token"
```



Note:

- If an Oracle Net Descriptor style URL includes the TOKEN_LOCATION parameter, then the value of that parameter takes precedence over a value defined by CONNECTION PROPERTY TOKEN LOCATION.
- The token file must contain a JSON Web Token (JWT) on a single line of UTF-8 encoded text. The JWT format is specified by RFC 7519.
- The token location must also contain a private key file named oci_db_key.pem. The private key file must use the PEM format and contain the base64 encoding of an RSA private key in the PKCS#8 encoding

You can also set the connection property

CONNECTION_PROPERTY_TOKEN_AUTHENTICATION (oracle.jdbc.tokenAuthentication) to OAUTH and the connection property CONNECTION_PROPERTY_TOKEN_LOCATION (oracle.jdbc.tokenLocation) to point to the bearer token on the file system. There is no default location set in this case, so you must set the location to either of the following:

- A directory, in which case, the driver loads a file named token
- A fully qualified file name

You can perform this in the following ways:

Configuring the ojdbc.properties file

Enable the OAUTH authentication mode oracle.jdbc.tokenAuthentication=OAUTH # Specify the location of the Bearer token location oracle.jdbc.tokenLocation=/home/user1/mytokens/jwtbearertoken

Using the JDBC URL

jdbc:oracle:thin:@tcps:adb.mydomain.oraclecloud.com:1522/ xyz.adb.oraclecloud.com? oracle.jdbc.tokenAuthentication=OAUTH&oracle.jdbc.tokenLocation=/ home/user/token

Using the TNS format for a JDBC URL:

```
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCPS)(PORT=1521)
(HOST=adb.mydomain.oraclecloud.com))(CONNECT_DATA=
(SERVICE_NAME=xyz.adb.oraclecloud.com))(SECURITY=(TOKEN_AUTH=OAUTH)
(TOKEN LOCATION=/home/user1/mytokens/jwtbearertoken)))"
```



9.1.2 Using the oracle.jdbc.accessToken Connection Property

Set the CONNECTION_PROPERTY_ACCESS_TOKEN (oracle.jdbc.accessToken) to the access token value.

You can perform this in the following ways:

Note:

You must enclose the token value with double quotation marks ("") to escape the equal signs (=) that may appear in the token value.

Using the JDBC URL

```
jdbc:oracle:thin:@tcps:adb.mydomain.oraclecloud.com:1522/
xyz.adb.oraclecloud.com?oracle.jdbc.accessToken="ey...5c"
```

Using the Descriptor URL

Configuring the ojdbc.properties File

```
# Enable the OAUTH authentication mode
oracle.jdbc.accessToken="ey...5c"
```

Or,

Enable the OAUTH authentication mode oracle.jdbc.accessToken=\${DATABASE ACCESS TOKEN}

Where, the access token is the value of the DATABASE_ACCESS_TOKEN environment variable.

Note:

You do not need to set <code>oracle.jdbc.tokenAuthentication=OAUTH</code> as the driver automatically sets the <code>OAUTH</code> mode, when the access token is provided.



9.1.3 Using the OracleConnectionBuilder Interface

Call the OracleConnectionBuilder.accessToken method for authentication with a database access token. This method accepts a token value that the application obtains from the authentication service.

If you pass a token using this method, then it overrides the connection string setting of TOKEN_AUTH=OCI_TOKEN, which means that JDBC does not read the token from the file system as it typically does. Instead, JDBC uses the AccessToken object provided to the accessToken method.

In this case, JDBC also generates a signature using the private key and sends it to the Database along with the IAM database access token. First, the Database verifies the token with the public signing key from IAM. Then, it verifies the JDBC-generated signature by decrypting it with a public key that is embedded in the token. If the decrypted signature is valid, then it proves that JDBC possesses the private key.

A single instance of the OracleDataSource class, configured with a single URL, creates instances of the OracleConnectionBuilder interface. These instances support traditional authentication with O5Logon, while also supporting token-based authentication. The application then calls the methods to configure a user and a password, or calls methods to configure a token. However, it is invalid to configure this builder with both a token and with a user name or a password. If both the accessToken method and the password or user methods are invoked with non-null values, then a SQLException, indicating an invalid configuration, is thrown when creating a connection with this builder.

9.1.4 Using the OracleDataSource Class

Call the OracleCommonDataSource.setTokenSupplier(AccessToken accessToken) method for authentication with a database access token.

This method sets a supplier function that generates an access token when creating a connection with this DataSource. The supplier function is invoked each time this DataSource creates a connection. Instances of access tokens, which are generated by the supplier, must represent a token type that is supported by Oracle Database for client authentication. The supplier must be thread safe.

Note:

Use the AccessToken.createJsonWebTokenCache (Supplier) method to create a thread safe Supplier that caches tokens from a user defined Supplier.

It is invalid to configure this DataSource with both a token supplier and with a user name or password. If you invoke the setUser(String), setPassword(String), setConnectionProperties(java.util.Properties), Or setConnectionProperty(String, String) methods to configure this DataSource with a user name or a password, and also invoke the setTokenSuppliersetTokenSupplier(AccessToken accessToken) method to



configure a token supplier, then a SQLException indicating an invalid configuration is thrown, when creating a connection with this DataSource.

The access tokens are ephemeral in nature and expire within an hour or less. So, use the Supplier type that enables an instance of the OracleDataSource class to obtain a newly generated token, each time it creates a connection. The Supplier can generate the same token multiple times, until the expiration time of that token passes. After the expiration time of a token is over, the Supplier must no longer generate that token, and instead begin to generate a new token with a later expiration time.

See Also:

- Authenticating and Authorizing IAM Users for Oracle Autonomous Databases
- About TCP/IP with TLS Protocol

9.2 Support for Token-Based Authentication for Azure AD

In this release of Oracle Database, the JDBC Thin drivers provide support for Azure Active Directory (Azure AD) OAuth2 access tokens.

While connecting to the database, the JDBC application provides a token to the database. The database verifies the token with a public key that it requests from the authentication service, and retrieves the corresponding user group membership information to find the database schema and role mappings to complete the user authorization to the database.

The token-based authentication is supported in the following ways:

9.2.1 Using the File System

When a database token is available on the file system, then you can configure the JDBC driver to use this token for connecting to the Database.

You can use the CONNECTION_PROPERTY_TOKEN_AUTHENTICATION for this purpose, which can be specified in the following ways:

- As an ojdbc.properties file
- As a JVM system property
- As a parameter in the query section of a connection string
- With a Properties object passed to OracleDataSource.setConnectionProperties (Properties)
- As a parameter in the SECURITY section of an Oracle Net Descriptor

JDBC supports the following two types of connection strings, where you can specify this parameter:

As an Oracle Net descriptor, where you can specify the parameter as:

```
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(HOST=dbhost)(PORT=1522)
(PROTOCOL=tcps))(SECURITY=(SSL_SERVER_DN_MATCH=ON)
(TOKEN AUTH=OAUTH)(TOKEN LOCATION=/path/to/my/token)))
```



• As a connection property, where you can specify the parameter as:

```
jdbc:oracle:thin:@tcps:dbhost:1522/my.example.com?
oracle.jdbc.tokenAuthentication=OAUTH&oracle.jdbc.tokenLocation
="/path/to/my/token"
```

When CONNECTION_PROPERTY_TOKEN_AUTHENTICATION is set to OAUTH, then the CONNECTION_PROPERTY_TOKEN_LOCATION specifies the file system path, from where you can obtain the access tokens. There is no default location in this case. You must set the token location, which can be a directory containing the token in a filed named token. For instance, if the directory mytokendirectory contains the file named token, then you set the token location in the following way:

/path/to/mytokendirectory

Note:

- If an Oracle Net Descriptor style URL includes the TOKEN_LOCATION parameter, then the value of that parameter takes precedence over a value defined by CONNECTION_PROPERTY_TOKEN_LOCATION.
- The token file must contain a JSON Web Token (JWT) on a single line of UTF-8 encoded text. The JWT format is specified by RFC 7519.

You can also set the connection property

CONNECTION_PROPERTY_TOKEN_AUTHENTICATION (oracle.jdbc.tokenAuthentication) to OAUTH and the connection property CONNECTION_PROPERTY_TOKEN_LOCATION (oracle.jdbc.tokenLocation) to point to the bearer token on the file system. There is no default location set in this case, so you must set the location to either of the following:

- A directory, in which case, the driver loads a file named token
- A fully qualified file name

You can perform this in the following ways:

Configuring the ojdbc.properties file

```
# Enable the OAUTH authentication mode
oracle.jdbc.tokenAuthentication=OAUTH
# Specify the location of the Bearer token location
oracle.jdbc.tokenLocation=/home/user1/mytokens/jwtbearertoken
```

Using the JDBC URL

```
jdbc:oracle:thin:@tcps:adb.mydomain.oraclecloud.com:1522/
xyz.adb.oraclecloud.com?
oracle.jdbc.tokenAuthentication=OAUTH&oracle.jdbc.tokenLocation=/
home/user/token
```



Using the TNS format for a JDBC URL:

```
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCPS)(PORT=1521)
(HOST=adb.mydomain.oraclecloud.com))(CONNECT_DATA=
(SERVICE_NAME=xyz.adb.oraclecloud.com))(SECURITY=(TOKEN_AUTH=OAUTH)
(TOKEN LOCATION=/home/user1/mytokens/jwtbearertoken)))"
```

9.2.2 Using the oracle.jdbc.accessToken Connection Property

Set the CONNECTION_PROPERTY_ACCESS_TOKEN (oracle.jdbc.accessToken) to the access token value.

You can perform this in the following ways:

Note:

You must enclose the token value with double quotation marks ("") to escape the equal signs (=) that may appear in the token value.

Using the JDBC URL

```
jdbc:oracle:thin:@tcps:adb.mydomain.oraclecloud.com:1522/
xyz.adb.oraclecloud.com?oracle.jdbc.accessToken="ey...5c"
```

Using the Descriptor URL

Configuring the ojdbc.properties File

Enable the OAUTH authentication mode oracle.jdbc.accessToken="ey...5c"

Or,

```
# Enable the OAUTH authentication mode
oracle.jdbc.accessToken=${DATABASE ACCESS TOKEN}
```

Where, the access token is the value of the DATABASE_ACCESS_TOKEN environment variable.

Note:

You do not need to set oracle.jdbc.tokenAuthentication=OAUTH as the driver automatically sets the OAUTH mode, when the access token is provided.



9.2.3 Using the OracleConnectionBuilder Interface

Call the OracleConnectionBuilder.accessToken method for authentication with a database access token.

This method accepts a token value that the application obtains from the authentication service. If you pass a token using this method, then it overrides the connection string setting of TOKEN_AUTH=OCI_TOKENthen TOKEN_AUTH=OAUTH, which means that JDBC does not read the token from the file system as it typically does. Instead, JDBC uses the AccessToken object provided to the accessToken method.

A single instance of the OracleDataSource class, configured with a single URL, creates instances of the OracleConnectionBuilder interface. These instances support traditional authentication with O5Logon, while also supporting token-based authentication. The application then calls the methods to configure a user and a password, or calls methods to configure a token. However, it is invalid to configure this builder with both a token and with a user name or a password. If both the accessToken method and the password or user methods are invoked with non-null values, then a SQLException, indicating an invalid configuration, is thrown when creating a connection with this builder.

9.2.4 Using the OracleDataSource Class

Call the OracleCommonDataSource.setTokenSupplier(AccessToken accessToken) method for authentication with a database access token.

This method sets a supplier function that generates an access token when creating a connection with this DataSource. The supplier function is invoked each time this DataSource creates a connection. Instances of access tokens, which are generated by the supplier, must represent a token type that is supported by Oracle Database for client authentication. The supplier must be thread safe.

Note:

Use the AccessToken.createJsonWebTokenCache (Supplier) method to create a thread safe Supplier that caches tokens from a user defined Supplier.

It is invalid to configure this DataSource with both a token supplier and with a user name or password. If you invoke the setUser(String), setPassword(String), setConnectionProperties(java.util.Properties), Or setConnectionProperty(String, String) methods to configure this DataSource with a user name or a password, and also invoke the setTokenSuppliersetTokenSupplier(AccessToken accessToken) method to configure a token supplier, then a SQLException indicating an invalid configuration is thrown, when creating a connection with this DataSource.

The access tokens are ephemeral in nature and expire within an hour or less. So, use the Supplier type that enables an instance of the OracleDataSource class to obtain a newly generated token, each time it creates a connection. The Supplier can generate the same token multiple times, until the expiration time of that token passes. After the



expiration time of a token is over, the Supplier must no longer generate that token, and instead begin to generate a new token with a later expiration time.

See Also:

- Use Azure Active Directory (Azure AD) with Autonomous Database
- About TCP/IP with TLS Protocol

9.3 Support for Oracle Advanced Security

This section describes the following concepts:

- Overview of Oracle Advanced Security
- JDBC OCI Driver Support for Oracle Advanced Security
- JDBC Thin Driver Support for Oracle Advanced Security

9.3.1 Overview of Oracle Advanced Security

Oracle Advanced Security provides the following security features:

Network Encryption

Sensitive information communicated over enterprise networks and the Internet can be protected by using encryption algorithms, which transform information into a form that can be deciphered only with a decryption key. For example, AES.

To ensure network integrity during transmission, Oracle Advanced Security generates a cryptographically secure message digest. Starting from Oracle Database 12c Release 1 (12.1), the SHA-2 list of hashing algorithms are also supported and Oracle Advanced Security uses the following hashing algorithms to generate the secure message digest and includes it with each message sent across a network.

This protects the communicated data from attacks, such as data modification, deleted packets, and replay attacks.

The following code snippet shows how to calculate the checksum using any of the algorithms mentioned previously:

```
prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM_TYPES,
"( SHA1)");
prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM_LEVEL,
"REQUIRED");
```

Strong Authentication

To ensure network security in distributed environments, it is necessary to authenticate the user and check his credentials. Password authentication is the most common means of authentication. Oracle Database enables strong authentication with Oracle authentication adapters, which support various third-party authentication services, including TLS with digital certificates. Oracle Database supports the following industry-standard authentication methods:

- Kerberos
- Remote Authentication Dial-In User Service (RADIUS)



Transport Layer Security (TLS)



9.3.2 JDBC OCI Driver Support for Oracle Advanced Security

If you are using the JDBC OCI driver, which presumes that you are running from a computer with an Oracle client installation, then support for Oracle Advanced Security and incorporated third-party features is fairly similar to the support provided by in any Oracle client situation. Your use of Advanced Security features is determined by related settings in the sqlnet.ora file on the client computer.

Note:

Starting from Oracle Database 12c Release 1 (12.1), Oracle recommends you to use the configuration parameters present in the new XML configuration file oraaccess.xml instead of the OCI-specific configuration parameters present in the sqlnet.ora file. However, the configuration parameters present in the sqlnet.ora file are still supported.

The JDBC OCI driver attempts to use external authentication if you try connecting to a database without providing a password. The following are some examples using the JDBC OCI driver to connect to a database without providing a password:

TLS Authentication

The following code snippet shows how to use TLS authentication to connect to the database:

Example 9-1 Using TLS Authentication to Connect to the Database

```
import java.sql.*;
import java.util.Properties;
public class test
{
    public static void main( String [] args ) throws Exception
    {
        String url = "jdbc:oracle:oci:@"
        +"(DESCRIPTION=(ADDRESS=(PROTOCOL=tcps)(HOST=localhost)(PORT=5221))"
        +"(CONNECT_DATA=(SERVICE_NAME=orcl)))";
        Driver driver = new oracle.jdbc.OracleDriver();
        Properties props = new Properties();
        Connection conn = driver.connect( url, props );
        conn.close();
    }
}
```



Using a Data Source

The following code snippet shows how to use a data source to connect to the database:

Example 9-2 Using a Data Source to Connect to the Database

```
import java.sql.*;
import java.sql.*;
import java.util.Properties;
import oracle.jdbc.pool.*;
public class testpool {
    public static void main( String args ) throws Exception
    { String url = "jdbc:oracle:oci:@" +"(DESCRIPTION=(ADDRESS=(PROTOCOL=tcps)
(HOST=localhost)(PORT=5221))"
    +"(CONNECT_DATA=(SERVICE_NAME=orcl)))";
    OracleConnectionPoolDataSource ocpds = new OracleConnectionPoolDataSource();
    ocpds.setURL(url);
    PooledConnection pc = ocpds.getPooledConnection();
    Connection conn = pc.getConnection();
    }
  }
```

Note:

The key exception to the preceding, with respect to Java, is that the Transport Layer Security (TLS) protocol is supported by the Oracle JDBC OCI drivers only if you use native threads in your application. This requires special attention, because green threads are generally the default.

9.3.3 JDBC Thin Driver Support for Oracle Advanced Security

The JDBC Thin driver cannot assume the existence of an Oracle client installation or the presence of the sqlnet.ora file. Therefore, it uses a Java approach to support Oracle Advanced Security. Java classes that implement Oracle Advanced Security are included in the ojdbc8.jar and ojdbc11.jar files. Security parameters for encryption and integrity, usually set in the sqlnet.ora file, are set using a Java Properties object or through system properties.

9.4 Support for Login Authentication

Basic login authentication through JDBC consists of user names and passwords, as with any other means of logging in to an Oracle server. Specify the user name and password through a Java properties object or directly through the getConnection method call. This applies regardless of which client-side Oracle JDBC driver you are using, but is irrelevant if you are using the server-side internal driver, which uses a special direct connection and does not require a user name or password.

Starting with Oracle Database 12*c* Release 1 (12.1.0.2), the Oracle JDBC Thin driver supports the O7L_MR client ability when you are running your application with a JDK such as JDK 8, which supports the PBKDF2-SHA2 algorithm. If you are running an application with JDK 7, then you must add a third-party security provider that supports the PBKDF2-SHA2 algorithm,



otherwise the driver will not support the new 12a password verifier that requires the O7L MR client ability.

If you are using Oracle Database 12c Release 1 (12.1.0.2) with the SQLNET.ALLOWED_LOGON_VERSION_SERVER parameter set to 12a, then keep the following points in mind:

- You must also use the 12.1.0.2 Oracle JDBC Thin driver and JDK 8 or JDK 7 with a third-party security provider that supports the PBKDF2-SHA2 algorithm
- If you use an earlier version of Oracle JDBC Thin driver, then you will get the following error:

```
ORA-28040: No matching authentication protocol
```

 If you use the 12.1.0.2 Oracle JDBC Thin driver with JDK 7, then also you will get the same error, if you do not add a third-party security provider that supports the PBKDF2-SHA2 algorithm.

9.5 Support for Strong Authentication

Oracle Advanced Security enables Oracle Database users to authenticate externally. External authentication can be with RADIUS, Kerberos, Certificate-Based Authentication, Token Cards, and Smart Cards. This is called strong authentication. Oracle JDBC drivers provide support for the following strong authentication methods:

- Kerberos
- RADIUS
- TLS

See Also:

Oracle Database Net Services Reference



9.6 Support for Network Encryption and Integrity

The section describes the support for network encryption and integrity.

Note:

 Starting with Oracle Database 21c, older encryption and hashing algorithms are deprecated.

The deprecated algorithms for DBMS_CRYPTO and native network encryption include MD4, MD5, DES, 3DES, and RC4-related algorithms as well as 3DES for Transparent Data Encryption (TDE). Removing older, less secure cryptography algorithms prevents accidental use of these algorithms. To meet your security requirements, Oracle recommends that you use more modern cryptography algorithms, such as the Advanced Encryption Standard (AES).

• Oracle provides a patch that you can download to address necessary security enhancements that affect native network encryption environments in Oracle Database release 11.2 and later. This patch is available in My Oracle Support note 2118136.2.

See Also:

Oracle Database Security Guide

This section describes the following concepts:

- Overview of JDBC Support for Data Encryption and Integrity
- JDBC OCI Driver Support for Encryption and Integrity
- JDBC Thin Driver Support for Encryption and Integrity
- Setting Encryption and Integrity Parameters in Java

9.6.1 Overview of JDBC Support for Network Encryption and Integrity

You can use Oracle Database and Oracle Advanced Security network encryption and integrity features in your Java database applications, depending on related settings in the server. When using the JDBC OCI driver, set parameters as you would in any Oracle client situation. When using the Thin driver, set parameters through a Java properties object.

Encryption is enabled or disabled based on a combination of the client-side encryption-level setting and the server-side encryption-level setting. Similarly, integrity is enabled or disabled based on a combination of the client-side integrity-level setting and the server-side integrity-level setting.

Encryption and integrity support the same setting levels, REJECTED, ACCEPTED, REQUESTED, and REQUIRED. Table 9-1 shows how these possible settings on the client-side and server-side combine to either enable or disable the feature. By default, remote OS authentication (through TCP) is disabled in the database for security reasons.



Client/Server Settings Matrix	Client Rejected	Client Accepted (default)	Client Requested	Client Required
Server Rejected	OFF	OFF	OFF	connection fails
Server Accepted (default)	OFF	OFF	ON	ON
Server Requested	OFF	ON	ON	ON
Server Required	connection fails	ON	ON	ON

Table 9-1 Client/Server Negotiations for Encryption or Integrity

Table 9-1 shows, for example, that if encryption is requested by the client, but rejected by the server, it is disabled. The same is true for integrity. As another example, if encryption is accepted by the client and requested by the server, it is enabled. The same is also true for integrity.

See Also:

Oracle Database Security Guide for more information about network encryption and integrity features

Note:

The term checksum still appears in integrity parameter names, but is no longer used otherwise. For all intents and purposes, checksum and integrity are synonymous.

9.6.2 JDBC OCI Driver Support for Encryption and Integrity

If you are using the JDBC OCI driver, which presumes an Oracle-client setting with an Oracle client installation, then you can enable or disable network encryption or integrity and set related parameters as you would in any Oracle client situation, through settings in the sqlnet.ora file on the client.

Note:

Starting from Oracle Database 12c Release 1 (12.1), Oracle recommends you to use the configuration parameters present in the new XML configuration file oraaccess.xml instead of the OCI-specific configuration parameters present in the sqlnet.ora file. However, the configuration parameters present in the sqlnet.ora file are still supported.

To summarize, the client parameters are shown in Table 9-2:



Parameter Description	Parameter Name	Possible Settings
Client encryption level	SQLNET.ENCRYPTION_CLIENT	REJECTED ACCEPTED REQUESTED REQUIRED
Client encryption selected list	SQLNET.ENCRYPTION_TYPES_CLIENT	AES128, AES192, AES256
Client integrity level	SQLNET.CRYPTO_CHECKSUM_CLIENT	REJECTED ACCEPTED REQUESTED REQUIRED
Client integrity selected list	SQLNET.CRYPTO_CHECKSUM_TYPES_CLIENT	SHA-1

Table 9-2 OCI Driver Client Parameters for Encryption and Integrity

9.6.3 JDBC Thin Driver Support for Encryption and Integrity

The JDBC Thin driver support for network encryption and integrity parameter settings parallels the JDBC OCI driver support discussed in the preceding section. You can set the corresponding parameters through a Java properties object that you can use while opening a database connection.

The default value for the encryption and integrity level is ACCEPTED for both the server side and the client side. This enables you to achieve the desired security level for a connection pair by configuring only one side of a connection, either the server side or the client side. This increases the efficiency of your program because if there are multiple Oracle clients connecting to an Oracle Server, then you need to change the encryption and integrity level to REQUESTED in the sqlnet.ora file only on the server side to turn on encryption or integrity for all connections. This saves time and effort because you do not have to change the settings for each client separately.

Table 9–3 lists the parameter information for the JDBC Thin driver. These parameters are defined in the oracle.jdbc.OracleConnection interface.

Parameter Name	Parameter Type	Possible Settings
CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_LEVEL	String	REJECTED ACCEPTED REQUESTED REQUIRED
CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_TYPES	String	AES256, AES192, AES128
CONNECTION_PROPERTY_THIN_NET_CHECKSUM_LEVEL	String	REJECTED ACCEPTED REQUESTED REQUIRED
CONNECTION_PROPERTY_THIN_NET_CHECKSUM_TYPES	String	SHA1

Table 9-3 Thin Driver Client Parameters for Encryption and Integrity



Note:

 Oracle Advanced Security support for the Thin driver is incorporated directly into the JDBC classes JAR file. So, there is no separate version for domestic and export editions. Only parameter settings that are suitable for an export edition are possible.

9.6.4 Setting Encryption and Integrity Parameters in Java

Use a Java properties object, that is, an instance of java.util.Properties, to set the network encryption and integrity parameters supported by the JDBC Thin driver.

The following example instantiates a Java properties object, uses it to set each of the parameters in Table 9-3, and then uses the properties object in opening a connection to the database:

```
...
Properties prop = new Properties();
prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_LEVEL,
"REQUIRED");
prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM_LEVEL,
"REQUESTED");
prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM_TYPES,
"( SHA1 )");
OracleDataSource ods = new OracleDataSource();
ods.setConnectionProperties(prop);
ods.setURL("jdbc:oracle:thin:@localhost:5221:main");
Connection conn = ods.getConnection();
...
```

The parentheses around the values encryption type and checksum type allow for lists of values. When multiple values are supplied, the server and the client negotiate to determine which value is to be actually used.

Example

Example 9-3 is a complete class that sets network encryption and integrity parameters before connecting to a database to perform a query.

Note:

In the example, the string REQUIRED is retrieved dynamically through the functionality of the AnoServices and Service classes. You have the option of retrieving the strings in this manner or including them in the software code as shown in the previous examples.

Before running this example, you must turn on encryption in the sqlnet.ora file. For example, the following lines will turn on AES256, AES192, and AES128 for the encryption and SHA1 for the checksum:



```
SQLNET.ENCRYPTION_SERVER = ACCEPTED
SQLNET.CRYPTO_CHECKSUM_SERVER = ACCEPTED
SQLNET.CRYPTO_CHECKSUM_TYPES_SERVER = (SHA1)
SQLNET.ENCRYPTION_TYPES_SERVER = (AES256, AES192, AES128)
```

Example 9-3 Setting Network Encryption and Integrity Parameters

```
import java.sql.*;
import java.util.Properties;
import oracle.net.ano.AnoServices;
import oracle.jdbc.*;
import oracle.jdbc.pool.*;
public class DemoAESAndSHA1
{
  static final String USERNAME= "HR";
  static final String PASSWORD= "hr";
 static final String URL =
"jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=localhost)
(PORT=5221))"
      +"(CONNECT DATA=(SERVICE NAME=orcl)))";
  public static final void main(String[] argv)
    DemoAESAndSHA1 demo = new DemoAESAndSHA1();
    try
    {
     demo.run();
    }catch(SQLException ex)
    {
      ex.printStackTrace();
    }
  }
  void run() throws SQLException
    OracleDataSource ods = new OracleDataSource();
    Properties prop = new Properties();
    // We require the connection to be encrypted with either AES256 or
AES192.
    // If the database does not accept such a security level, then the
connection attempt will fail.
prop.setProperty(OracleConnection.CONNECTION PROPERTY THIN NET ENCRYPTION LEV
EL, AnoServices.ANO REQUIRED);
prop.setProperty(OracleConnection.CONNECTION PROPERTY THIN NET ENCRYPTION TYP
ES, "( " + AnoServices.ENCRYPTION AES256 + "," +
AnoServices.ENCRYPTION AES192 + ")");
```

 $//\ensuremath{\,\mathrm{We}}$ also require the use of the SHA1 algorithm for network integrity checking.



```
prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM
_LEVEL, AnoServices.ANO_REQUIRED);
prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM
_TYPES, "( " + AnoServices.CHECKSUM_SHA1 + " )");
    prop.setProperty("user", DemoAESAndSHA1.USERNAME);
    prop.setProperty("password", DemoAESAndSHA1.PASSWORD);
    ods.setConnectionProperties(prop);
    ods.setURL(DemoAESAndSHA1.URL);
    OracleConnection oraConn = (OracleConnection) ods.getConnection();
    System.out.println("Connection created! Encryption algorithm is: " + oraConn.getEncryptionAlgorithmName() + ", network integrity
    algorithm is: " + oraConn.getDataIntegrityAlgorithmName());
    oraConn.close();
  }
}
```

9.7 Support for TLS

This section describes the following topics:

- Overview of JDBC Support for TLS
- About Managing Certificates and Wallets
- About Keys and certificates containers
- Database Connectivity Over TLS Version 1.2 Using JDBC Thin and JKS
- Automatic TLS Connection Configuration
- Support for Key Store Service

9.7.1 Overview of JDBC Support for TLS

Oracle Database 19c provides support for the Transport Layer Security (TLS) protocol. TLS is a widely used industry standard protocol that provides secure communication over a network. TLS provides authentication, data encryption, and data integrity. It provides a secure enhancement to the standard TCP/IP protocol, which is used for Internet communication.

TLS uses digital certificates that comply with the X.509v3 standard for authentication and a public and private key pair for encryption. TLS also uses secret key cryptography and digital signatures to ensure privacy and integrity of data. When a network connection over TLS is initiated, the client and server perform a TLS handshake that includes the following steps:

- Client and server negotiate about the cipher suites to use. This includes deciding on the encryption algorithms to be used for data transfer.
- Server sends its certificate to the client, and the client verifies that the certificate
 was signed by a trusted certification authority (CA). This step verifies the identity of
 the server.



- If client authentication is required, the client sends its own certificate to the server, and the server verifies that the certificate was signed by a trusted CA.
- Client and server exchange key information using public key cryptography. Based on this
 information, each generates a session key. All subsequent communications between the
 client and the server is encrypted and decrypted by using this set of session keys and the
 negotiated cipher suite.

TLS Terminology

The following terms are commonly used in the TLS context:

- **Certificate**: A certificate is a digitally signed document that binds a public key with an entity. The certificate can be used to verify that the public key belongs to that individual.
- **Certification authority**: A certification authority (CA), also known as certificate authority, is an entity which issues digitally signed certificates for use by other parties.
- **Cipher suite**: A cipher suite is a set of cryptographic algorithms and key sizes used to encrypt data sent over a TLS-enabled network.
- **Private key**: A private key is a secret key, which is never transmitted over a network. The private key is used to decrypt a message that has been encrypted using the corresponding public key. It is also used to sign certificates. The certificate is verified using the corresponding public key.
- **Public key**: A public key is an encryption key that can be made public or sent by ordinary means such as an e-mail message. The public key is used for encrypting the message sent over TLS. It is also used to verify a certificate signed by the corresponding private key.
- Key Store or Wallet: A wallet is a password-protected container that is used to store authentication and signing credentials, including private keys, certificates, and trusted certificates required by TLS.
- **Security Provider**: A Java implementation that provides some functionality related to security. A provider is responsible for decoding a key store file.
- Key Store Service (KSS): A component of Oracle Platform Security services. KSS enables a key store to be referenced as a URI with kss:// scheme (rather than a file name).

Java Version of TLS

The Java Secure Socket Extension (JSSE) provides a framework and an implementation for a Java version of the TLS protocol. JSSE provides support for data encryption, server and client authentication, and message integrity. It abstracts the complex security algorithms and handshaking mechanisms and simplifies application development by providing a building block for application developers, which they can directly integrate into their applications. JSSE is integrated into Java Development Kit (JDK) 1.4 and later, and supports TLS version 2.0 and 3.0.

Oracle strongly recommends that you have a clear understanding of the JavaTM Secure Socket Extension (JSSE) framework before using TLS in the Oracle JDBC drivers.

The JSSE standard application programming interface (API) is available in the javax.net, javax.net.ssl, and javax.security.cert packages. These packages provide classes for creating and configuring sockets, server sockets, TLS sockets, and TLS server sockets. The packages also provide a class for secure HTTP connections, a public key certificate API compatible with JDK1.1-based platforms, and interfaces for key and trust managers.



TLS works the same way, as in any networking environment, in Oracle Database 18c.

Note:

In order to use JSSE in your program, you must have clear understanding of JavaTM Secure Socket Extension (JSSE) framework.

9.7.2 About Managing Certificates and Wallets

To establish a TLS connection with a JDBC client, either Thin or OCI, the Oracle Database server sends its certificate, which is stored in its wallet. The client may or may not need a certificate or wallet, depending on the server configuration.

The Oracle JDBC Thin driver uses the JSSE framework to create a TLS connection. It uses the default provider (*SunJSSE*) to create a TLS context, but you can provide your own provider.

You do not need a certificate for the client, unless the SSL_CLIENT_AUTHENTICATION parameter is set on the server.

9.7.3 About Keys and certificates containers

Java clients can use multiple types of containers such as Oracle wallets, JKS, PKCS12, and so on, as long as a provider is available. For Oracle wallets, *OraclePKI* provider must be used because the PKCS12 support provided by *SunJSSE* provider does not support all the features of PKCS12. In order to use *OraclePKI* provider, the following JARs are required:

- oraclepki.jar
- osdt_cert.jar
- osdt_core.jar

All these JAR files should be under <code>\$ORACLE_HOME/jlib</code> directory.

9.7.4 Database Connectivity Over TLS Version 1.2 Using JDBC Thin and JKS

This section describes the steps to configure the Oracle JDBC thin driver to connect to the Database using SSL v1.2.

• Always use the latest update of the JDK

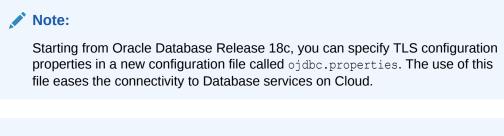
Use the latest update of either JDK 11 or JDK 8 because the updated versions include bug fixes that are required for using SSL version 1.2.

• Install the JCE files

Install the Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files because irrespective of the JDK version that you use, the strong cipher suites (for example TLS_RSA_WITH_AES_256_CBC_SHA256) are not enabled without them.

• Use JKS files or wallets







Oracle Database JDBC Java API Reference

After performing all the preceding steps, if you run into more issues, then you can turn on tracing to diagnose the problems using -Djavax.net.debug=all option.

9.7.5 Automatic TLS Connection Configuration

Starting from Oracle Database Release 18c, you can use default values or programmatic logic for resolving the connection configuration values without manually adding or updating the security provider. You can resolve the configuration values in the following two ways:

- Provider Resolution
- Automatic Key Store Type (KSS) Resolution

9.7.5.1 Provider Resolution

For certain key store types, the JDBC driver can resolve the provider implementation that is used to load the key store. For these types, it is not necessary to register the provider with Java security. As long as the provider implementation is on the CLASSPATH, the driver can instantiate the security provider.

The following key store types map to a known provider:

- **SSO:** oracle.security.pki.OraclePKIProvider
- KSS: oracle.security.jps.internal.keystore.provider.FarmKeyStoreProvider

The driver attempts to resolve the provider only if there is no provider registered for the specified type.

If the oraclepki.jar file is on the CLASSPATH, then the driver can automatically load the Oracle PKI Provider in the following way:

```
java -cp oraclepki.jar:ojdbc8.jar -D javax.net.ssl.keyStore=/path/to/wallet/
cwallet.sso MyApp
```

Similarly, for a specified value of the oracle.net.wallet_location connection property, the driver can automatically load the Oracle PKI Provider in the following way:

```
java -cp .:oraclepki.jar:ojdbc8.jar -D oracle.net.wallet_location=file:/
path/to/wallet/cwallet.sso MyApp
```



Note:

For PKCS12 types created by the orapki tool (The ewallet.p12 file), you may still need to register the OraclePKIProvider with Java security because the PKCS12 file created by the orapki tool includes the ASN1 *Key Bag* element (Type Code: 1.2.840.113549.1.12.10.1.1). The Sun PKCS12 implementation does not support the *Key Bag* type and throws an error when attempting to read the ewallet.p12 file. For HotSpot and Open JDK users, the Sun Provider comes bundled as the PKCS12 provider. This means that the PKCS12 provider will already have a registered provider, and the driver will make no attempt to override this.

9.7.5.2 Automatic Key Store Type (KSS) Resolution

The JDBC driver can resolve common key store types based on the value of the javax.net.ssl.keyStore and javax.net.ssl.trustStore properties, eliminating the need to specify the type using these properties.

Key Store or Trust Store with a Recognized File Extension

A key store or trust store with a recognized file extension maps to the following types:

• File extension .jks resolves to javax.net.ssl.keyStoreType as JKS:

java -cp ojdbc8.jar -D javax.net.ssl.keyStore=/path/to/keystore/ keystore.jks MyApp

• File extension .sso resolves to javax.net.ssl.keyStoreType as SSO:

java -cp ojdbc8.jar -D javax.net.ssl.keyStore=/path/to/keystore/keystore.sso MyApp

• File extension .p12 resolves to javax.net.ssl.keyStoreType as PKCS12:

java -cp ojdbc8.jar -D javax.net.ssl.keyStore=/path/to/keystore/keystore.p12 MyApp

• File extension .pfx resolves to javax.net.ssl.keyStoreType as PKCS12:

```
java -cp ojdbc8.jar -D
javax.net.ssl.keyStore=/path/to/keystore/keystore.pfx MyApp
```

Key Store or Trust Store with a URI

If the key store or the trust store is a URI with a $\tt kss://$ scheme, this maps to type KSS:

```
java -cp ojdbc8.jar -D
javax.net.ssl.keyStore=kss://MyStripe/MyKeyStore MyApp
```



Note:

You can set the javax.net.ssl.trustStoreType and javax.net.ssl.keyStoreType properties for overriding the default type resolution.

9.7.6 Support for Default TLS Context

For applications that require finer control over the TLS configuration, you can configure the JDBC driver to use the SSLContext returned by the SSLContext.getDefault method. Use one of the following methods for the driver to use the default SSLContext:

- javax.net.ssl.keyStore=NONE
- javax.net.ssl.trustStore=NONE

You can use the default SSLContext to support key store types that are not file-based. Common examples of such key store types include hardware-based smart cards. Key store types that require programmatic call to the load (KeyStore.LoadStoreParameter) method also belong to this category.

See Also:

- https://docs.oracle.com/javase/8/docs/api/javax/net/ssl/ SSLContext.html#getDefault--
- https://docs.oracle.com/javase/8/docs/api/java/security/KeyStore.html#loadjava.security.KeyStore.LoadStoreParameter-

9.7.7 Support for Key Store Service

This release of Oracle Database introduces support for Key Store Service (KSS) in the JDBC driver. So, if you have configured a Key Store Service in a WebLogic server, then JDBC applications can now integrate with the existing Key Store Service configuration.

The driver can load the key stores that are managed by the Key Store Service. If the value of the javax.net.ssl.keyStore property or the javax.net.ssl.trustStore property is a URI with kss:// scheme, then the driver loads the key store from Key Store Service.

For permission-based protection, the following permission must be granted to the ojdbc JAR file:

```
permission KeyStoreAccessPermission
"stripeName=*,keystoreName=*,alias=*", "read";
```

This permission grants access to every key store. For limiting the scope of access, you can replace the asterisk wild cards (*) with a specific application stripe and a key store name. The driver does not load the key store as a privileged action, which means that the KeyStoreAccessPermission must also be granted to the application code base.



9.8 Support for Kerberos

This section discusses the following topics:

- Overview of JDBC Support for Kerberos
- Configuring Windows to Use Kerberos
- Configuring Oracle Database to Use Kerberos
- Code Example for Using Kerberos
- Support for Kerberos Constrained Delegation
- Kerberos Authentication Enhancements

9.8.1 Overview of JDBC Support for Kerberos

Kerberos is a network authentication protocol that provides the tools of authentication and strong cryptography over the network. Kerberos helps you secure your information systems across your entire enterprise by using secret-key cryptography. The Kerberos protocol uses strong cryptography so that a client or a server can prove its identity to its server or client across an insecure network connection. After a client and server have used Kerberos to prove their identity, they can also encrypt all of their communications to assure privacy and data integrity as they go about their business.

The Kerberos architecture is centered around a trusted authentication service called the key distribution center, or KDC. Users and services in a Kerberos environment are referred to as principals; each principal shares a secret, such as a password, with the KDC. A principal can be a user such as HR or a database server instance.

Starting from 12c Release 1, Oracle Database supports cross-realm authentication for Kerberos. If you add the referred realm appropriately in the domain_realms section of the kerberos configuration file, then being in one particular realm, you can access the services of another realm.

Starting from Release 19c, Oracle Database supports Kerberos Constrained Delegation. This feature added a new method OracleConnectionBuilder gssCredential (GSSCredential credential) in the oracle.jdbc.OracleConnectionBuilder interface. This method accepts the GSSCredential of the user and then delegates it during the Kerberos authentication in the driver.

See Also:

Oracle Database JDBC Java API Reference

9.8.2 Configuring Windows to Use Kerberos

A good Kerberos client providing klist, kinit, and other tools, can be found at the following link:

http://web.mit.edu/kerberos/dist/index.html



This client also provides a nice GUI.

You need to make the following changes to configure Kerberos on your Windows machine:

- 1. Right-click the My Computer icon on your desktop.
- 2. Select Properties. The System Properties dialog box is displayed.
- 3. Select the Advanced tab.
- 4. Click Environment Variables. The Environment Variables dialog box is displayed.
- 5. Click New to add a new user variable. The New User Variable dialog box is displayed.
- 6. Enter KRB5CCNAME in the Variable name field.
- 7. Enter FILE:C:\Documents and Settings\<user_name>\krb5cc in the Variable value field.
- 8. Click **OK** to close the New User Variable dialog box.
- 9. Click OK to close the Environment Variables dialog box.
- 10. Click **OK** to close the System Properties dialog box.

Note:

C:\WINDOWS\krb5.ini file has the same content as krb5.conf file.

9.8.3 Configuring Oracle Database to Use Kerberos

Perform the following steps to configure Oracle Database to use Kerberos:

1. Use the following command to connect to the database:

```
SQL> connect system
Enter password: password
```

2. Use the following commands to create a user CLIENT@MYORACLE.COM that is identified externally:

SQL> create user "CLIENT@MYORACLE.COM" identified externally; SQL> grant create session to "CLIENT@MYORACLE.COM";

3. Use the following commands to connect to the database as sysdba and dismount it:

SQL> connect / as sysdba SQL> shutdown immediate;

4. Add the following line to \$T_WORK/t_init1.ora file:

OS AUTHENT PREFIX=""

5. Use the following command to restart the database:

SQL> startup pfile=t_init1.ora

6. Modify the sqlnet.ora file to include the following lines:

```
names.directory_path = (tnsnames)
#Kerberos
sqlnet.authentication_services = (beq,kerberos5)
```



```
sqlnet.authentication_kerberos5_service = dbji
sqlnet.kerberos5_conf = /home/Jdbc/Security/kerberos/krb5.conf
sqlnet.kerberos5_keytab = /home/Jdbc/Security/kerberos/dbji.oracleserver
sqlnet.kerberos5_conf_mit = true
sqlnet.kerberos_cc_name = /tmp/krb5cc_5088
# logging (optional):
trace_level_server=16
trace_directory_server=/scratch/sqlnet/
```

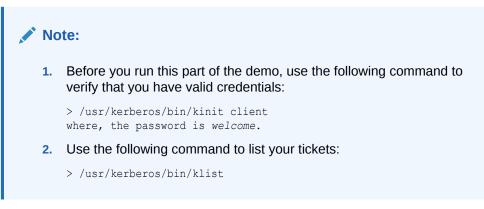
7. Use the following commands to verify that you can connect through SQL*Plus:

```
> kinit client
> klist
    Ticket cache: FILE:/tmp/krb5cc_5088
    Default principal: client@MYORACLE.COM
    Valid starting Expires Service principal
    06/22/06 07:13:29 06/22/06 17:13:29 krbtgt/MYORACLE.COM@MYORACLE.COM
    Kerberos 4 ticket cache: /tmp/tkt5088
    klist: You have no tickets cached
> sqlplus '/@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)
(HOST=oracleserver.mydomain.com)(PORT=5221))
(CONNECT DATA=(SERVICE NAME=orcl)))'
```

9.8.4 Code Example for Using Kerberos

This following example demonstrates the Kerberos authentication feature that is part of Oracle Database 23c JDBC thin driver. This demo covers two scenarios:

• In the first scenario, the OS maintains the user name and credentials. The credentials are stored in the cache and the driver retrieves the credentials before trying to authenticate to the server. This scenario is in the module connectWithDefaultUser().



 The second scenario covers the case where the application wants to control the user credentials. This is the case of the application server where multiple web users have their own credentials. This scenario is in the module connectWithSpecificUser().



Note:

To run this demo, you need to have a working setup, that is, a Kerberos server up and running, and an Oracle database server that is configured to use Kerberos authentication. You then need to change the URLs used in the example to compile and run it.

Example 9-4 Using Kerberos Authentication to Connect to the Database

```
import com.sun.security.auth.module.Krb5LoginModule;
import java.io.IOException;
import java.security.PrivilegedExceptionAction;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.HashMap;
import java.util.Properties;
import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.callback.UnsupportedCallbackException;
import oracle.jdbc.OracleConnection;
import oracle.jdbc.OracleDriver;
import oracle.net.ano.AnoServices;
public class KerberosJdbcDemo
  String url ="jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)"+
    "(HOST=oracleserver.mydomain.com)(PORT=5221))(CONNECT DATA=" +
    "(SERVICE NAME=orcl)))";
  public static void main(String[] arv)
    /* If you see the following error message [Mechanism level: Could not load
     * configuration file c:\winnt\krb5.ini (The system cannot find the path
     * specified] it's because the JVM cannot locate your kerberos config file.
     * You have to provide the location of the file. For example, on Windows,
     * the MIT Kerberos client uses the config file: C\WINDOWS\krb5.ini:
     */
    // System.setProperty("java.security.krb5.conf","C:\\WINDOWS\\krb5.ini");
    System.setProperty("java.security.krb5.conf","/home/Jdbc/Security/kerberos/
krb5.conf");
    KerberosJdbcDemo kerberosDemo = new KerberosJdbcDemo();
    try
    {
      System.out.println("Attempt to connect with the default user:");
      kerberosDemo.connectWithDefaultUser();
    }
    catch (Exception e)
      e.printStackTrace();
    }
    trv
```



{

```
System.out.println("Attempt to connect with a specific user:");
     kerberosDemo.connectWithSpecificUser();
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }
  void connectWithDefaultUser() throws SQLException
  {
    OracleDriver driver = new OracleDriver();
    Properties prop = new Properties();
prop.setProperty(OracleConnection.CONNECTION PROPERTY_THIN_NET_AUTHENTICATION_SER
VICES,
      "("+AnoServices.AUTHENTICATION KERBEROS5+")");
prop.setProperty(OracleConnection.CONNECTION PROPERTY THIN NET AUTHENTICATION KRB
5 MUTUAL,
     "true");
    /* If you get the following error [Unable to obtain Principal Name for
     * authentication] although you know that you have the right TGT in your
     * credential cache, then it's probably because the JVM can't locate your
     * cache.
     * Note that the default location on windows is "C:\Documents and
Settings\krb5cc username".
     */
    11
prop.setProperty (OracleConnection.CONNECTION PROPERTY THIN NET AUTHENTICATION KRB
5 CC NAME,
    /*
      On Linux:
        > which kinit
         /usr/kerberos/bin/kinit
        > ls -l /etc/krb5.conf
        lrwxrwxrwx 1 root root 47 Jun 22 06:56 /etc/krb5.conf -> /home/
Jdbc/Security/kerberos/krb5.conf
         > kinit client
         Password for client@MYORACLE.COM:
         > klist
         Ticket cache: FILE:/tmp/krb5cc 5088
        Default principal: client@MYORACLE.COM
                                               Service principal
         Valid starting
                            Expires
         11/02/06 09:25:11 11/02/06 19:25:11 krbtgt/MYORACLE.COM@MYORACLE.COM
         Kerberos 4 ticket cache: /tmp/tkt5088
         klist: You have no tickets cached
    */
```

prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_AUTHENTICATION_KRB



```
5 CC NAME,
                     "/tmp/krb5cc 5088");
    Connection conn = driver.connect(url,prop);
    String auth = ((OracleConnection)conn).getAuthenticationAdaptorName();
    System.out.println("Authentication adaptor="+auth);
   printUserName(conn);
    conn.close();
  }
  void connectWithSpecificUser() throws Exception
    Subject specificSubject = new Subject();
    // This first part isn't really meaningful to the sake of this demo. In
    // a real world scenario, you have a valid "specificSubject" Subject that
    // represents a web user that has valid Kerberos credentials.
   Krb5LoginModule krb5Module = new Krb5LoginModule();
   HashMap sharedState = new HashMap();
   HashMap options = new HashMap();
   options.put("doNotPrompt", "false");
    options.put("useTicketCache", "false");
    options.put("principal","client@MYORACLE.COM");
   krb5Module.initialize(specificSubject,newKrbCallbackHandler(),sharedState,options);
   boolean retLogin = krb5Module.login();
    krb5Module.commit();
    if(!retLogin)
     throw new Exception ("Kerberos5 adaptor couldn't retrieve credentials (TGT) from
the cache");
    // to use the TGT from the cache:
    // options.put("useTicketCache","true");
    // options.put("doNotPrompt","true");
    // options.put("ticketCache","C:\\Documents and Settings\\user\\krb5cc");
    // krb5Module.initialize(specificSubject,null,sharedState,options);
    // Now we have a valid Subject with Kerberos credentials. The second scenario
    // really starts here:
    // execute driver.connect(...) on behalf of the Subject 'specificSubject':
    Connection conn =
      (Connection) Subject.doAs (specificSubject, new PrivilegedExceptionAction()
        {
          public Object run()
          {
            Connection con = null;
            Properties prop = new Properties();
            prop.setProperty(AnoServices.AUTHENTICATION PROPERTY SERVICES,
                             "(" + AnoServices.AUTHENTICATION KERBEROS5 + ")");
            try
              OracleDriver driver = new OracleDriver();
              con = driver.connect(url, prop);
            } catch (Exception except)
              except.printStackTrace();
            return con;
          }
```

```
});
    String auth = ((OracleConnection)conn).getAuthenticationAdaptorName();
    System.out.println("Authentication adaptor="+auth);
   printUserName(conn);
    conn.close();
  }
  void printUserName (Connection conn) throws SQLException
    Statement stmt = null;
    try
    {
     stmt = conn.createStatement();
     ResultSet rs = stmt.executeQuery("select user from dual");
     while(rs.next())
       System.out.println("User is:"+rs.getString(1));
     rs.close();
    }
    finally
    {
     if(stmt != null)
        stmt.close();
    }
 }
}
class KrbCallbackHandler implements CallbackHandler
{
public void handle(Callback[] callbacks) throws IOException,
                                                  UnsupportedCallbackException
 {
   for (int i = 0; i < callbacks.length; i++)</pre>
   {
     if (callbacks[i] instanceof PasswordCallback)
     {
       PasswordCallback pc = (PasswordCallback)callbacks[i];
       System.out.println("set password to 'welcome'");
       pc.setPassword((new String("welcome")).toCharArray());
     } else
     {
       throw new UnsupportedCallbackException(callbacks[i],
                                               "Unrecognized Callback");
   }
 }
}
```

9.8.5 Support for Kerberos Constrained Delegation

Starting from Oracle Database Release 19c, the Thin driver supports the Kerberos constrained delegation feature.

To make it a secure practice, the implementation relies on the use of the GSSCredential utility. For this purpose, the OracleConnectionBuilder interface has been enhanced, so that it can accept a GSSCredential object.



Example 9-5 Using Constrained Delegation

The following example shows how to implement constrained delegation in your code:

```
OracleDataSource ods = new OracleDataSource();
ods.setURL(url);
ods.setConnectionProperties(prop);
con = ods.createConnectionBuilder()
                     .gssCredential(impersonatedGssUserCreds)
                    .build();
```

🖍 See Also:

- Java Platform, Standard Edition Security Developer's Guide
- The OracleConnectionBuilder Interface

9.8.6 Kerberos Authentication Enhancements

Starting with Oracle Database Release 23c, Kerberos authentication does not need instantiating the KerberosLoginModule or the availability of Ticket Granting Ticket (TGT) in the CredentialCache.

In this release, you can connect to Oracle Database using the following enhanced Kerberos Authentication methods:

- Kerberos Authentication Using the User and the Password Properties
- Kerberos Authentication Using the JAAS Configuration

9.8.6.1 Kerberos Authentication Using the User and the Password Properties

Now you can configure the Kerberos Principal and Password properties in the same way as you configure the user and password properties for a simple authentication (O5Logon). Using these values, the JDBC Thin driver initializes the KerberosLoginModule for your application, simplifying the Kerberos Authentication configuration.

For using the configured user name and password, you must set the <code>PASSWORD_AUTH</code> parameter to <code>KERBEROS5</code> in the connection string. You can also set <code>PASSWORD_AUTH</code> to <code>KERBEROS5</code> using the <code>oracle.jdbc.passwordAuthentication</code> connection property. However, the value specified in the connection string has a higher priority.

Example 9-6 Kerberos Authentication: Using the User and the Password Properties

The following example shows how you can configure the Kerberos Principal and the Password:

```
import oracle.jdbc.OracleConnection;
import oracle.jdbc.OracleDriver;
import java.sql.Connection;
import java.sql.SQLException;
import java.util.Properties;
```



```
public class KerberosExample {
  private static String KERBEROS PRINCIPAL = "client@EXAMPLE.COM";
  private static String PASSWORD = <password>;
  private static String URL
="jdbc:oracle:thin:@(DESCRIPTION=(SECURITY=(PASSWORD AUTH=KERBEROS5))
(ADDRESS=(PROTOCOL=TCP)"+
      "(HOST=myserver.example.com)(PORT=5221))
(CONNECT DATA=(SERVICE NAME=orcl)))";
  public static void main(String a[]) {
    try{
      Properties prop = new Properties();
      prop.setProperty("oracle.net.authentication services",
"(KERBEROS5)");
      prop.setProperty("user", KERBEROS_PRINCIPAL); // Kerberos
Principal
      prop.setProperty("password", PASSWORD); // Kerberos Password
      OracleDriver driver = new OracleDriver();
      Connection conn = driver.connect(URL, prop);
      String auth =
((OracleConnection)conn).getAuthenticationAdaptorName();
      System.out.println ("Authentication adaptor=" + auth);
      conn.close();
    }
    catch(SQLException e) {
      e.printStackTrace();
    }
  }
}
```

9.8.6.2 Kerberos Authentication Using the JAAS Configuration

In this release, you can specify the JAAS configuration file through the JDBC connection properties. By default, the Thin driver uses the default kerberos login module that is bundled with Oracle JDK

(com.sun.security.auth.module.Krb5LoginModule). If you want to override this behaviour in your application, then use the JAAS configurations.

The following code example shows how you can configure the Kerberos Authentication using the JAAS configurations:

```
import java.sql.Connection;
import java.util.Properties;
import oracle.jdbc.OracleDriver;
import oracle.jdbc.OracleConnection;
public class JAASKerberosAuthentication {
    // Set the following properties
    private final static String DB_URL =
```





```
"(ADDRESS=(PROTOCOL=TCP)(HOST=myserver.example.com)(PORT=5221))" +
      "(CONNECT DATA=(SERVICE NAME=orcl)))";
 private final static String KERBEROS CONFIG FILE = "/etc/krb5.conf";
 private final static String JAAS CONFIG FILE PATH = "/myworkdir/jaas.conf";
 private final static String KERBEROS LOGIN MODULE NAME = "kprb5module";
 public static void main(String a[]) throws Exception {
    // Set JAAS configuration
System.setProperty("java.security.auth.login.config", JAAS CONFIG FILE PATH);
    // Set Kerberos Configuration
    System.setProperty("java.security.krb5.conf", KERBEROS CONFIG FILE);
   OracleDriver driver = new OracleDriver();
    Properties prop = new Properties();
    prop.setProperty("oracle.net.authentication services", "(KERBEROS5)");
    prop.setProperty("oracle.net.KerberosJaasLoginModule",
KERBEROS LOGIN MODULE NAME);
    Connection conn = driver.connect(DB URL, prop);
    String auth = ((OracleConnection) conn).getAuthenticationAdaptorName();
    System.out.println("Got Connection. Authentication adaptor=" + auth);
    conn.close();
  }
}
```

The preceding example uses the following JAAS configuration file in the /myworkdir/ jaas.conf directory:

```
kprb5module {
    com.sun.security.auth.module.Krb5LoginModule required
        doNotPrompt=true useTicketCache=true ticketCache="file:/tmp/
krb5cc_5088";
};
```

9.9 Support for RADIUS

This section describes the following concepts:

- Overview of JDBC Support for RADIUS
- Configuring Oracle Database to Use RADIUS
- Code Example for Using RADIUS
- Support for Challenge-Response Authentication

9.9.1 Overview of JDBC Support for RADIUS

Oracle Database 11g Release 1 introduced support for Remote Authentication Dial-In User Service (RADIUS). RADIUS is a client/server security protocol that is most widely known for enabling remote authentication and access. Oracle Advanced Security uses this standard in a client/server network environment to enable use of any authentication method that supports



the RADIUS protocol. RADIUS can be used with a variety of authentication mechanisms, including token cards and smart cards.

9.9.2 Configuring Oracle Database to Use RADIUS

Perform the following steps to configure Oracle Database to use RADIUS:

1. Use the following command to connect to the database:

```
SQL> connect system
Enter password: password
```

2. Use the following commands to create a new user aso from within a database:

```
SQL> create user aso identified externally;
SQL> grant create session to aso;
```

3. Use the following commands to connect to the database as sysdba and dismount it:

```
SQL> connect / as sysdba
SQL> shutdown immediate;
```

Add the following lines to the t init1.ora file:

```
os_authent_prefix = ""
```

Note:

Once the test is over, you need to revert the preceding changes made to the t_init1.ora file.

5. Use the following command to restart the database:

```
SQL> startup pfile=?/work/t_init1.ora
```

6. Modify the sqlnet.ora file so that it contains only these lines:

```
sqlnet.authentication_services = ( beq, radius)
sqlnet.radius_authentication = <RADUIUS_SERVER_HOST_NAME>
sqlnet.radius_authentication_port = 1812
sqlnet.radius_authentication_timeout = 120
sqlnet.radius_secret=/home/Jdbc/Security/radius/radius_key
# logging (optional):
trace_level_server=16
trace_directory_server=/scratch/sqlnet/
```

7. Use the following command to verify that you can connect through SQL*Plus:

```
>sqlplus 'aso/1234@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)
(HOST=oracleserver.mydomain.com)(PORT=5221))
(CONNECT DATA=(SERVICE NAME=orcl)))'
```

9.9.3 Code Example for Using RADIUS

This example demonstrates the new RADIUS authentication feature that is a part of Oracle Database 12*c* Release 1 (12.1) JDBC thin driver. You need to have a working setup, that is, a RADIUS server up and running, and an Oracle database server that is configured to use RADIUS authentication. You then need to change the URLs given in the example to compile and run it.



```
Example 9-7 Using RADIUS Authentication to Connect to the Database
```

```
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;
import oracle.jdbc.OracleConnection;
import oracle.jdbc.OracleDriver;
import oracle.net.ano.AnoServices;
public class RadiusJdbcDemo
{
  String url ="jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)"+
    "(HOST=oracleserver.mydomain.com)(PORT=5221))(CONNECT_DATA=" +
    "(SERVICE NAME=orcl)))";
  public static void main(String[] arv)
   RadiusJdbcDemo radiusDemo = new RadiusJdbcDemo();
   try
    {
     radiusDemo.connect();
   }
   catch (Exception e)
    {
     e.printStackTrace();
    }
  }
  /*
  * This method attempts to logon to the database using the RADIUS
  * authentication protocol.
  * It should print the following output to stdout:
   * _____
   * Authentication adaptor=RADIUS
   * User is:ASO
   * _____
  */
  void connect() throws SQLException
  {
   OracleDriver driver = new OracleDriver();
   Properties prop = new Properties();
prop.setProperty(OracleConnection.CONNECTION PROPERTY THIN NET AUTHENTICATION SERVICES,
      "("+AnoServices.AUTHENTICATION RADIUS+")");
    // The user "aso" needs to be properly setup on the radius server with
   // password "1234".
   prop.setProperty("user", "aso");
   prop.setProperty("password","1234");
   Connection conn = driver.connect(url,prop);
   String auth = ((OracleConnection)conn).getAuthenticationAdaptorName();
    System.out.println("Authentication adaptor="+auth);
   printUserName(conn);
   conn.close();
  }
```

void printUserName (Connection conn) throws SQLException



```
{
   Statement stmt = null;
   try
   {
    stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery("select user from dual");
    while(rs.next())
      System.out.println("User is:"+rs.getString(1));
      rs.close();
   }
   finally
   {
    if(stmt != null)
      stmt.close();
   }
}
```

9.9.4 Support for Challenge-Response Authentication

The RADIUS challenge-response authentication is an interactive authentication, where the RADIUS server asks for a valid response to a displayed challenge. Starting from Oracle Database Release 23c, the JDBC thin drivers support this authentication.

In challenge-response authentication, the first level of authentication is performed using the user name and the password. The RADIUS server then sends a challenge to the application, to which the application must respond. For handling the challenge, you must configure a handler in your application, which is responsible for producing the response, using a given hint. The hint is provided to the handler as a byte array and the value of the byte array is dependent on the configuration of the challengeresponse authentication in the RADIUS server.



You can configure the handler in the following two ways:

- Using the oracle.net.radius_challenge_response_handler connection property
- Using the ConnectionBuilder.radiusChallengeResponseHandler method

Using the oracle.net.radius_challenge_response_handler Connection Property

Use the oracle.net.radius_challenge_response_handler connection property to configure the fully qualified name of the class that handles the challenge. This handler class must implement the java.util.function.Function
Syte[], byte[]> interface. The following example shows how to configure RADIUS challenge-response authentication using the oracle.net.radius_challenge_response_handler connection property.

```
import oracle.jdbc.OracleConnection;
import oracle.jdbc.pool.OracleDataSource;
```

```
import java.sql.SQLException;
```



```
import java.util.Properties;
import java.util.function.Function;
public class RadiusExample {
  private static String USER = "myRadiusUser";
  private static String PASSWORD = "myRadiusPwd";
 private static String URL
="jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)"+
      "(HOST=myserver.example.com)(PORT=5221))
(CONNECT DATA=(SERVICE NAME=orcl)))";
  public static void main(String a[]) {
    try {
      OracleDataSource ods = new OracleDataSource();
      Properties prop = new Properties();
      prop.setProperty("oracle.net.authentication services", "RADIUS");
      prop.setProperty("user", USER);
      prop.setProperty("password", PASSWORD);
      prop.setProperty("oracle.net.radius challenge response handler",
"MyChallengeResponseHandler");
      ods.setConnectionProperties(prop);
      ods.setURL(URL);
      OracleConnection conn = (OracleConnection) ods.getConnection();
      System.out.println("Got Connection. Authentication adaptor="+
conn.getAuthenticationAdaptorName());
      conn.close();
    }
    catch(SQLException e) {
      e.printStackTrace();
    }
  }
  class MyChallengeResponseHandler implements Function<byte[], byte[]> {
    @Override
    public byte[] apply(byte[] hint) {
      // TODO: use the hint to produce the challenge response
      byte[] response = null;
      return response;
    }
  }
}
```

Using the ConnectionBuilder.radiusChallengeResponseHandler Method

You can configure the handler using the new

ConnectionBuilder.radiusChallengeResponseHandler method. Use this method to specify the handler lambda / instance. The following example shows how to configure RADIUS challenge-response authentication using the radiusChallengeResponseHandler method:

```
import oracle.jdbc.OracleConnection;
import oracle.jdbc.pool.OracleDataSource;
```



```
import java.sql.SQLException;
import java.util.Properties;
public class RadiusExample {
  private static String USER = "myRadiusUser";
  private static String PASSWORD = "myRadiusPwd";
  private static String URL
="jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)"+
      "(HOST=myserver.example.com)(PORT=5221))
(CONNECT DATA=(SERVICE NAME=orcl)))";
  public static void main(String a[]) {
    try {
      OracleDataSource ods = new OracleDataSource();
      Properties prop = new Properties();
      prop.setProperty("oracle.net.authentication services", "RADIUS");
      prop.setProperty("user", USER);
      prop.setProperty("password", PASSWORD);
      ods.setConnectionProperties(prop);
      ods.setURL(URL);
      OracleConnection conn = ods.createConnectionBuilder()
          .radiusChallengeResponseHandler(RadiusExample::getResponse).b
uild();
      System.out.println("Got Connection. Authentication adaptor="+
conn.getAuthenticationAdaptorName());
      conn.close();
    }
    catch(SQLException e) {
      e.printStackTrace();
    }
  }
  private static byte[] getResponse(byte[] hint) {
    // TODO: use the hint to produce the challenge response
   byte[] response = null;
    return response;
  }
}
```

9.10 Secure External Password Store

For large-scale deployments where applications use password credentials to connect to databases, you can use a client-side Oracle wallet to store such credentials. An Oracle wallet is a secure software container that is used to store authentication and sign-in credentials.

Storing database password credentials in a client-side Oracle wallet eliminates the need to embed user names and passwords in application code, batch jobs, or scripts. This reduces the risk of exposing passwords in the scripts and application code, and simplifies maintenance because you do not need to change your code each time user names and passwords change. In addition, if you do not have to change the



application code, then it also becomes easier to enforce password management policies for these user accounts.

You can set the <code>oracle.net.wallet_location</code> connection property to specify the wallet location. The JDBC driver can then retrieve the user name and password pair from this wallet.

Note:

When an Oracle wallet is opened in a JDBC application, for security reasons, the wallet file permissions are aligned to make it accessible only to the wallet owner (creator).

See Also:

- Oracle Database Administrator's Guide for more information about configuring your client to use secure external password store and for information about managing credentials in it
- Oracle Database Security Guide for more information about managing the secure external password store for password credentials



10 JDBC Service Provider Extensions

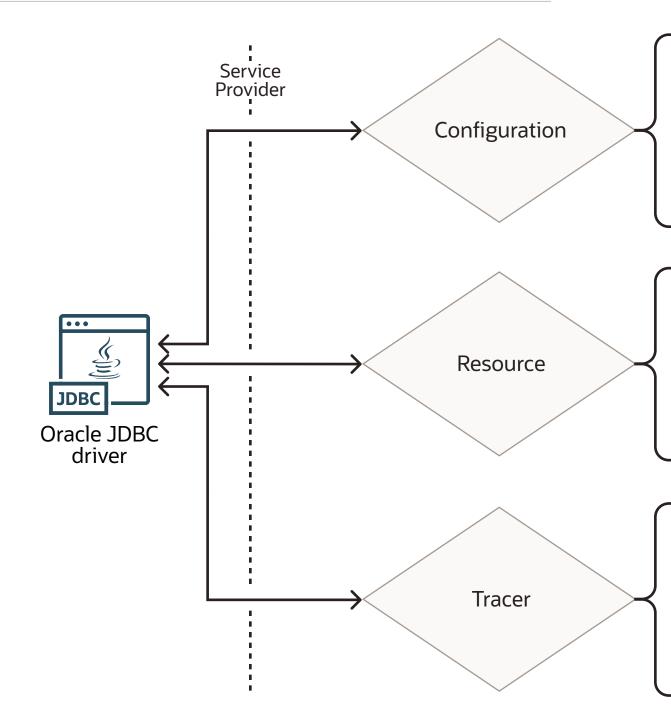
Starting with Oracle Database Release 23c, you can extend the capabilities of Oracle JDBC drivers through service providers. You can use the standard Service Provider Interface to define custom providers and load them at run time.

You can load the following three types of providers:

- Centralized Configuration Providers
- Resource Providers
- Trace Event Listener Providers
- JDBC Extensions for Cloud Vendors

The following diagram illustrates the use cases for each provider:





Oracle provides Open Source Providers for the following:

- Azure App Configuration
- OCI Object Storage
- Azure Vault
- Azure Active Directory Token
- OCI IAM Token
- OCI Vault



- Open Telemetry
- OCI APM
- HashiCorp Vault (through partnership)

10.1 Centralized Configuration Providers

Use these providers to support distributed configurations of database clients.

Note:

You can find the specification for the payloads and formats across .NET and C++ clients.

A Centralized Configuration Provider furnishes the Oracle JDBC drivers with the necessary information for creating a database connection including a connection string (JDBC URL), and optionally, JDBC connection properties such as user ID, database password, or wallet location. Sensitive information such as the Database password or wallet location is typically stored separately in a vault.

The Service Provider contract is defined in such a way that if the JDBC URL is of the following format:

jdbc:oracle:<driver type>:@config-<provider>

then, the driver attempts to load the *<provider>* from the list of registered service providers. For example:

jdbc:oracle:thin:@config-azure:myappconfig?key=sales app1&label=dev

To make the external provider discoverable to the applications, you must take care of the following:

- Implement the oracle.jdbc.spi.OracleConfigurationProvider interface available in the JDBC Driver
- Provide the URL string of the <provider> by overriding the getType method
- Provide an implementation of the getConnectionProperties method to return the properties to be used by the JDBC Driver

Oracle distributes individual JAR files as Configuration Providers for the following services in the Maven Central repository:

- Azure App Configuration (with optional reference to Azure Key Vault for secrets)
- OCI Object Storage (with optional reference to OCI Vault for secrets)
- OCI Database Tools Connections (with optional reference to OCI Vault for secrets)

If you want to build your own providers, then you must extend the oracle.jdbc.spi.OracleConfigurationProvider interface and follow the Java SPI specification.



10.1.1 Azure App Configuration

The Oracle data source uses a new prefix jdbc:oracle:thin:@config-azure: to be able to identify the configuration parameters that should be loaded to use the Azure App Configuration.

For enabling the existing applications to use this feature transparently, you only need to specify the name of the Azure App Configuration, and optionally, a prefix for the keynames and the Label, with the following syntax:

```
jdbc:oracle:thin:@config-azure:{appconfig-name}[?
key=prefix&label=value&option1
=value1&option2=value2]
```

The only requirements are the following:

- Include the provider JAR file in the classpath or the provider reference in the POM file
- Replace the existing URL values with the required values

This feature attaches the values of a data source to a key, which is the prefix of multiple keys in the Azure App Configuration, and a label. Both the key and the label are optional. If those are not specified, then it attaches all the values with no labels and no prefixes in the configuration.

Following are the four fixed values that are looked at with this key and label pair:

- connect descriptor (required)
- user (optional)
- password (optional)
- wallet location (optional)

The rest of the values are dependent on the JDBC Driver, which have the jdbc/ prefix. Multiple key-values pairs are retrieved for a specific label and key, which are applied to a data source. The key values are the properties (constant keys) defined in the OracleConnection interface.

For example, a data source URL with the value jdbc:oracle:thin:@configazure:myappconfig?key=/sales_app1/&label=dev in an App Configuration, say myappconfig, generates an OracleDatasource with values similar to the following values:

Note:

- Keep in mind that the prefix is /sales app1/ and the label is dev.
- The JDBC Driver internally concatenates the connection descriptor with jdbc:oracle:thin:@ to set the URL. This enables sharing the property with the other driver implementations.

ORACLE

Кеу	Value	Label	JDBC Driver Connection Property
/sales_app1/user	scott	dev	user=scott
/sales_app1/password	<pre>{"uri":"https:// mykeyvault.vault.azure. net/secrets/ <password>"}</password></pre>	dev	password= <i><password></password></i> (Value of the secret in the URI)
/sales_app1/ wallet_location	<pre>{"uri":"https:// mykeyvault.vault.azure. net/secrets/ <wallet_location>"}</wallet_location></pre>	dev	<pre>oracle.net.wallet_locat ion (value: "data:;base64,<value of<br="">the secret in the URI>")</value></pre>
/sales_app1/ connect_descriptor	<pre>(description=(retry_cou nt=20) (retry_delay=3) (address=(protocol=tcps) (port=1521) (host=myserver.oraclecl oud.com)) (connect_data=(service_ name=myservice.oraclecl oud.com)) (security=(ssl_server_d n_match=yes) (ssl_server_cert_dn="CN =DN1.oraclecloud.com, OU=Oracle US, O=Oracle Corporation, L=Redwood City, ST=California, C=US")))</pre>	dev	<pre>URL=jdbc:oracle:thin:@(description= (retry_count=20) (retry_delay=3) (address=(protocol=tcps)(port=1521) (host=myserver.oraclecl oud.com)) (connect_data=(service_ name= myservice.oraclecloud.c om))(security= (ssl_server_dn_match=ye s)(ssl_server_cert_dn= "CN=DN1.oraclecloud.com , OU=Oracle US, O=Oracle Corporation, L=Redwood City, ST=California, C=US")))</pre>
/sales_app1/jdbc/ autoCommit	false	dev	autoCommit=false
/sales_app1/jdbc/ oracle.jdbc.loginTimeou t	20	dev	oracle.jdbc.loginTimeou t=20
/sales_app1/jdbc/ oracle.jdbc.fanEnabled	false	dev	oracle.jdbc.fanEnabled= false

The following code snippet shows the relevant details in a sample Java client:

```
OracleDataSource ds = new OracleDataSource();
ds.setURL("jdbc:oracle:thin:@config-azure:myappconfig?
key=sales_app1&label=dev");
Connection cn = ds.getConnection();
Statement st = cn.createStatement();
ResultSet rs = st.executeQuery("select sysdate from dual");
```



10.1.2 OCI Object Storage

For using this Configuration Provider, you must make optional references to OCI Vault for secrets.

In this case, the configuration is stored in JSON format, which is common to all the clients. The provider is identified by ociobject provider in the URL, as shown in the following example:

```
jdbc:oracle:thin:@config-ociobject:
https://objectstorage.oraclecloud.com/myjava/bucket1/
payload_ojdbc_objectstorage.json
```

The only required parameter is the URL Path (URI) of the object. You can retrieve this value from the OCI Web Console, using the following navigation: Object Storage / Buckets / Object \rightarrow Object Details, which retrieves a value similar to the following:

```
https://objectstorage.oraclecloud.com/myjava/bucket1/
payload ojdbc objectstorage.json
```

There are four fixed values that are looked with this prefix and label pair:

- connect descriptor (required)
- user (optional)
- password (optional)
- wallet location (optional)

The rest of the values are dependent on the JDBC Driver, which have the jdbc/ prefix. Multiple key-values pairs are retrieved for a specific label and key, which are applied to a data source. The key values are the properties (constant keys) defined in the OracleConnection interface.

The password and wallet_location are references to a Vault provider, as shown in the following code snippet:

Note:

The Vault provider can also be an Azure Key Vault, which uses vault-azure as its type.

```
{"connect_descriptor": "(description=(retry_count=20)(retry_delay=3)
(address=(protocol=tcps)(port=1521)(host=myhost.oraclecloud.com))
(connect_data=(service_name=myservice.oraclecloud.com))
(security=(ssl_server_dn_match=yes)))",
    "user": "scott",
    "password": {
        "type": "vault-oci",
        "value": "myvalue",
```



```
"authentication": {
      "method": "OCI INSTANCE PRINCIPAL"
    }
  },
  "wallet location": {
    "type": "vault-oci",
    "value": "myvalue",
    "authentication": {
      "method": "OCI INSTANCE PRINCIPAL"
    }
  },
  "jdbc": {
    "oracle.jdbc.ReadTimeout": 1000,
    "defaultRowPrefetch": 20,
    "autoCommit": "false"
} }
```

You can set multiple keys in one payload, adding the optional name in the URL. For example:

```
jdbc:oracle:thin:@config-ociobject:{object-url}
[?key=name&option1=value1&option2=value2]
```

The following example uses two keys, namely, sales_app1 and hr_internal_app1:

```
{
 "sales app1":
 {"connect descriptor": "(description=(address=(protocol=tcps)
(port=1521) (host=myhost.oraclecloud.com))
    (connect data=(service name=myservice.oraclecloud.com))
(security=(ssl server dn match=yes)))",
   "user": "scott",
    "password": {
     "type": "vault-oci",
     "value": "myvalue",
     "authentication": {
        "method": "OCI INSTANCE PRINCIPAL"
      }
   },
   "jdbc": {
     "oracle.jdbc.ReadTimeout": 1000,
     "defaultRowPrefetch": 20,
     "autoCommit": "false"
   }
 },
 "hr internal app1":
 {"connect descriptor": "(description=(address=(protocol=tcps))
(port=1521) (host=myhost.oraclecloud.com))
    (connect data=(service name=myservice.oraclecloud.com))
(security=(ssl server dn match=yes)))",
   "user": "scott",
    "password": {
      "type": "vault-oci",
```

```
"value": "myvalue",
    "authentication": {
        "method": "OCI_INSTANCE_PRINCIPAL"
    }
},
    "jdbc": {
        "oracle.jdbc.ReadTimeout": 0,
        "defaultRowPrefetch": 100,
        "autoCommit": "true"
    }
}
```

10.1.3 OCI Database Tools

For using this Configuration Provider, you must make optional reference to OCI Vault for secrets.

The OCI Database Tools service is a managed service that can be used to configure connections to a database, either Oracle Autonomous Database or MySQL. You can then use the connection objects in the SQL worksheet in the Web console. You can also use it as a directory of database connection configurations, and also as a provider that allows access to these configurations. Each configuration has an Oracle Cloud Identifier (OCID) that is used to identify which connection is used. It contains a connectionString, a userName, a userPassword, keyStores, and advancedProperties. The advancedProperties are currently limited to JDBC properties.

The example shows a JDBC URL that uses the OCI Database Tools provider:

jdbc:oracle:thin:@config-ocidbtools:ocid1.databasetoolsconnection

10.1.4 Built-in Configuration Providers

The JDBC Driver includes two built-in Configuration Providers, one HTTPS Provider and one File provider.

You do not need any extra JAR files for using these providers as they are based on the same JSON Schema as used for OCI Object Storage, that is, the schema is the same for all the JSON based providers, namely, HTTPS, File, and OCI Object Storage.



HTTPS Configuration Provider

You can provide the JSON configuration document through an HTTPS endpoint like Oracle REST Data Services (ORDS). HTTP is not supported because the JSON configuration document may contain multiple aliases, so the URL must end with the alias name that needs to be loaded, for example, https://<URL>/aliasname.



You can protect the access to this configuration document using IP ACL, TCPS, and Basic HTTP Authentication.

The JDBC URL follows the following format:

```
jdbc:oracle:thin:@config-https://<URL>[?key=name&option1=value1]
```

As the https prefix is followed by a host, the double slash (//) is required after the provider name, whether it is http or https.

You can configure client authentication with basic HTTP authentication over HTTPS, using wallet through properties, similar to the following example:

```
jdbc:oracle:thin:@config-https://confighost.mydomain.com/oracleconfig?
key=name&authentication=BASIC_AUTH&wallet_location=/path/to/wallet
```

You can use the authentication option to make the HTTP Configuration Provider use the basic HTTP authentication to retrieve the JSON configuration document.

Note:

The HTTPS Configuration Provider that Oracle distributes, supports Basic HTTP Authentication.

File Configuration Provider

In this case, the JSON configuration document is provided through the file system and access to this configuration is protected by file system protections. File Configuration Provider, The JDBC URL follows the following format, in case of this Configuration Provider:

jdbc:oracle:thin:@config-file:{path-to-file}[?option list]

Here, the {path-to-file} parameter accepts the same rules as the java.io.File class. The option list includes an attribute to indicate the connection key name, for example, sales_app1 in the following example:

jdbc:oracle:thin:@config-file:path/to/file.json?key=sales app1

10.2 Resource Providers

A Resource Provider provides Oracle JDBC with a single resource, such as a password or a connection string.

Resource providers are configured by the connection properties. For example, you can use the following connection properties to configure a password provider:

```
oracle.jdbc.provider.password=example-provider
oracle.jdbc.provider.password.vaultId=9999-8888-7777
```



In the preceding example, the oracle.jdbc.provider.password property configures the name of a password provider. The oracle.jdbc.provider.password.vaultId property configures a parameter that is recognized by the password provider.

A resource provider may provide any of the following resources:

- Database connection string
- Database user name
- Database password
- Database access token
- TLS/SSL configuration
- Trace event listener

Corresponding to the list above, the following connection properties identify the name of Resource Provider:

- oracle.jdbc.provider.connectionString
- oracle.jdbc.provider.username
- oracle.jdbc.provider.password
- oracle.jdbc.provider.accessToken
- oracle.jdbc.provider.tlsConfiguration
- oracle.jdbc.provider.traceEventListener

You can also configure additional parameters as connection properties.

See Also:

10.3 Trace Event Listener Providers

Starting with Oracle Database Release 23c, the JDBC driver can generate events that can be used to monitor a JDBC application.

The JDBC driver defines an oracle.jdbc.spi.TraceEventListenerProvider interface that can be used to register a custom TraceEventListener through the SPI mechanism.

For example, you can use this feature to register a listener that publishes the following events to OpenTelemetry:

- Database round trips during query execution
- Virtual IP address retries while establishing a connection
- Starting of a recovery from a database outage when Application Continuity is used
- Successful recoveries from a database outage when Application Continuity is used



See Also:

The JDBC Javadoc for more information

10.4 JDBC Extensions for Cloud Vendors

The Oracle JDBC Driver Extensions include providers for centralized configuration or token providers for authentication with the Database.

These extensions help you in implementing the Service Provider Interfaces (SPIs) for integration with widely used services, such as Cloud computing platforms. The extensions are open-source on GitHub at the following link, and the artifacts are available on Maven Central:

https://github.com/oracle-samples/ojdbc-extensions/tree/main

For Centralized Configuration Providers, the extensions include providers for the following:

- OCI DBTools connection
- Azure App Configuration

For Resource Providers, these extensions include providers for the following:

- Authentication tokens issued by either OCI IAM (Identity and Access Management) or Azure AD (Active Directory), while using Oracle Autonomous Database Serverless
- The database password or user name stored in an OCI Vault secret or Azure Vault secret
- JDBC URL and client wallet from an Oracle Autonomous Database Serverless for mTLS connections
- JDBC events that are stored in OpenTelemetry

See Also:

Oracle JDBC Driver Extensions in GitHub fore information



11 Proxy Authentication

Oracle Java Database Connectivity (JDBC) provides proxy authentication, also called N-tier authentication. This feature is supported through both the JDBC Oracle Call Interface (OCI) driver and the JDBC Thin driver. This chapter contains the following sections:

- About Proxy Authentication
- Types of Proxy Connections
- Creating Proxy Connections
- Closing a Proxy Session
- Caching Proxy Connections
- Limitations of Proxy Connections

Note:

Oracle Database supports proxy authentication functionality in three tiers *only*. It does not support it across multiple middle tiers.

11.1 About Proxy Authentication

Proxy authentication is the process of using a middle tier for user authentication. You can design a middle tier server to proxy clients in a secure fashion by using the following three forms of proxy authentication:

- The middle tier server authenticates itself with the database server and a client. In this case, an application user or another application, authenticates itself with the middle tier server. Client identities can be maintained all the way through to the database.
- The client, that is, a database user, is not authenticated by the middle tier server. The client's identity and database password are passed through the middle tier server to the database server for authentication.
- The client, that is, a global user, is authenticated by the middle tier server, and passes either a Distinguished name (DN) or a Certificate through the middle tier for retrieving the client's user name.

Note:

Operations done on behalf of a client by a middle tier server can be audited.

In all cases, an administrator must authorize the middle tier server to proxy a client, that is, to act on behalf of the client. Suppose, the middle tier server initially connects to the database



as user HR and activates a proxy connection as user jeff, and then issues the following statement to authorize the middle tier server to proxy a client:

ALTER USER jeff GRANT CONNECT THROUGH HR;

You can also:

 Specify roles that the middle tier is permitted to activate when connecting as the client. For example,

```
CREATE ROLE role1;
GRANT SELECT ON employees TO role1;
ALTER USER jeff GRANT CONNECT THROUGH HR ROLE role1;
```

The role clause limits the access only to those database objects that are mentioned in the list of the roles. The list of roles can be empty.

- Find the users who are currently authorized to connect through a middle tier by querying the PROXY USERS data dictionary view.
- Disallow a proxy connection by using the REVOKE CONNECT THROUGH clause of ALTER USER statement.

Note:

In case of proxy authentication, a JDBC connection to the database creates a database session during authentication, and then other sessions can be created during the life time of the connection.

You need to use the different fields and methods present in the oracle.jdbc.OracleConnection interface to set up the different types of proxy connections.

11.2 Types of Proxy Connections

You can create proxy connections using any one of the following options:

• USER NAME

This is done by supplying the user name or the password or both. The SQL statement for specifying authentication using password is:

ALTER USER jeff GRANT CONNECT THROUGH HR AUTHENTICATED USING PASSWORD;

In this case, jeff is the user name and HR is the proxy for jeff.

The password option exists for additional security. Having no authenticated clause implies default authentication, which is using only the user name without the password. The SQL statement for specifying default authentication is:

ALTER USER jeff GRANT CONNECT THROUGH HR

• DISTINGUISHED NAME

This is a global name in lieu of the password of the user being proxied for. An example of the corresponding SQL statement using a distinguished name is:



```
CREATE USER jeff IDENTIFIED GLOBALLY AS 'CN=jeff,OU=americas,O=oracle,L=redwoodshores,ST=ca,C=us';
```

The string that follows the identified globally as clause is the distinguished name. It is then necessary to authenticate using this distinguished name. The corresponding SQL statement to specify authentication using distinguished name is:

ALTER USER jeff GRANT CONNECT THROUGH HR AUTHENTICATED USING DISTINGUISHED NAME;

• CERTIFICATE

This is a more encrypted way of passing the credentials of the user, who is to be proxied, to the database. The certificate contains the distinguished name encoded in it. One way of generating the certificate is by creating a wallet and then decoding the wallet to get the certificate. The wallet can be created using runutl mkwallet. It is then necessary to authenticate using the generated certificate. The SQL statement for specifying authentication using certificate is:

ALTER USER jeff GRANT CONNECT THROUGH HR AUTHENTICATED USING CERTIFICATE;

Note:

The use of certificates for proxy authentication will be desupported in future Oracle Database releases.

Note:

- All the options can be associated with roles.
- When opening a new proxied connection, a new session is started on the Database server. If you start a global transaction and then call the openProxySession method, then, at this point, you are no longer a part of the global transaction and instead it is like you are in a freshly created JDBC connection. Typically, this never happens because the openProxySession method is called prior to creating or resuming a global transaction. In such a case, you are still a part of the global transaction.

11.3 Creating Proxy Connections

A user, say jeff, has to connect to the database through another user, say ${\tt HR}$. The proxy user, ${\tt HR}$, should have an active authenticated connection. A proxy session is then created on this active connection, with the driver issuing a command to the server to create a session for the user, jeff. The server returns the new session ID, and the driver sends a session switch command to switch to this new session.

The JDBC OCI and Thin driver switch sessions in the same manner. The drivers permanently switch to the new session, jeff. As a result, the proxy session, HR, is not available until the new session, jeff, is closed.



Note:

You can use the isProxySession method from the oracle.jdbc.OracleConnection interface to check if the current session associated with your connection is a proxy session. This method returns true if the current session associated with the connection is a proxy session.

A new proxy session is opened by using the following method from the oracle.jdbc.OracleConnection interface:

void openProxySession(int type, java.util.Properties prop) throws
SQLExceptionOpens

Where,

type is the type of the proxy session and can have the following values:

OracleConnection.PROXYTYPE USER NAME

This type is used for specifying the user name.

• OracleConnection.PROXYTYPE_DISTINGUISHED_NAME

This type is used for specifying the distinguished name of the user.

• OracleConnection.PROXYTYPE_CERTIFICATE

This type is used for specifying the proxy certificate.

prop is the property value of the proxy session and can have the following values:

PROXY USER NAME

This property value should be used with the type OracleConnection.PROXYTYPE_USER_NAME. The value should be a java.lang.String.

PROXY DISTINGUISHED NAME

This property value should be used with the type OracleConnection.PROXYTYPE_DISTINGUISHED_NAME. The value should be a java.lang.String.

PROXY CERTIFICATE

This property value should be used with the type OracleConnection.PROXYTYPE_CERTIFICATE. The value is a bytep[] array that contains the certificate.

PROXY ROLES

This property value can be used with the following types:

- OracleConnection.PROXYTYPE_USER_NAME
- OracleConnection.PROXYTYPE_DISTINGUISHED_NAME
- OracleConnection.PROXYTYPE CERTIFICATE

The value should be a java.lang.String.

PROXY SESSION



This property value is used with the close method to close the proxy session.

PROXY USER PASSWORD

This property value should be used with the type OracleConnection.PROXYTYPE_USER_NAME. The value should be a java.lang.String.

The following code snippet shows the use of the openProxySession method:

```
java.util.Properties prop = new java.util.Properties();
prop.put(OracleConnection.PROXY_USER_NAME, "jeff");
String[] roles = {"role1", "role2"};
prop.put(OracleConnection.PROXY_ROLES, roles);
conn.openProxySession(OracleConnection.PROXYTYPE_USER_NAME, prop);
```

11.4 Closing a Proxy Session

You can close the proxy session opened with the OracleConnection.openProxySession method by passing the OracleConnection.PROXY_SESSION parameter to the OracleConnection.close method in the following way:

OracleConnection.close(OracleConnection.PROXY SESSION);

This is similar to closing a proxy session on a non-cached connection. The standard close method must be called explicitly to close the connection itself. If the close method is called directly, without closing the proxy session, then both the proxy session and the connection are closed. This can be achieved in the following way:

OracleConnection.close(OracleConnection.INVALID CONNECTION);

11.5 Caching Proxy Connections

Proxy connections, like standard connections, can be cached. Caching proxy connections enhances the performance. To cache a proxy connection, you need to create a connection using one of the getConnection methods on a cache enabled OracleDataSource object.

A proxy connection may be cached in the connection cache using the connection attributes feature of the connection cache. Connection attributes are name/value pairs that are userdefined and help tag a connection before returning it to the connection cache for reuse. When the tagged connection is retrieved, it can be directly used without having to do a round-trip to create or close a proxy session. Universal Connection Pool supports caching of any user/ password authenticated connection. Therefore, any user authenticated proxy connection can be cached and retrieved.

It is recommended that proxy connections should not be closed without applying the connection attributes. If a proxy connection is closed without applying the connection attributes, the connection is returned to the connection cache for reuse, but cannot be retrieved. The connection caching mechanism does not remember or reset session state.

A proxy connection can be removed from the connection cache by closing the connection directly.



11.6 Limitations of Proxy Connections

Closing a proxy connection automatically closes every SQL Statement created by the proxy connection, during the proxy session or prior to the proxy session. This may cause unexpected consequences on application pooling or statement caching. The following code samples explain this limitation of proxy connections:

Example 1

. . . .

```
public void displayName(String N) // Any function using the Proxy feature
{
     Properties props = new Properties();
     props.put("PROXY USER NAME", proxyUser);
     c.openProxySession(OracleConnection.PROXYTYPE USER NAME, props);
     . . . . . . .
     c.close(OracleConnection.PROXY SESSION);
}
public static void main (String args[]) throws SQLException
{
    . . . . . . . . . . . .
    PreparedStatement pstmt = conn.prepareStatement("SELECT first name FROM
EMPLOYEES WHERE employee id = ?");
    pstmt.setInt(1, 205);
    ResultSet rs = pstmt.executeQuery();
    while (rs.next())
    {
        displayName(rs.getString(1));
         if (rs.isClosed() // The ResultSet is already closed while closing the
connection!
         {
             throw new Exception ("Your ResultSet has been prematurely closed!
Your Statement object is also dead now.");
         }
    }
}
```

In the preceding example, when you close the proxy connection in the displayName method, then the PreparedStatement object and the ResultSet object also get closed. So, if you do not check the status of the ResultSet object inside loop, then the loop will fail when the next method is called for the second time.

Example 2

. . . .

```
PreparedStatement pstmt = conn.prepareStatement("SELECT first_name FROM
EMPLOYEES WHERE employee_id = ?");
    pstmt.setString(1, "205");
    ResultSet rs = pstmt.executeQuery();
    while (rs.next())
    {
        ....
    }
    Properties props = new Properties();
    props.put("PROXY USER NAME", proxyUser);
```



```
conn.openProxySession(OracleConnection.PROXYTYPE_USER_NAME, props);
......
conn.close(OracleConnection.PROXY_SESSION);
// Try to use the PreparedStatement again
pstmt.setString(1, "28960");
// This line of code will fail because the Statement is already closed while closing
the connection!
rs = pstmt.executeQuery();
```

In the preceding example, the PreparedStatement object and the ResultSet object work fine before opening the proxy connection. But, if you try to execute the same PreparedStatement object after closing the proxy connection, then the statement fails.



Part IV Data Access and Manipulation

This part provides a chapter that discusses about accessing and manipulating Oracle data. It also includes chapters that provide information about Java Database Connectivity (JDBC) support for user-defined object types, large object (LOB) and binary file (BFILE) locators and data, object references, and Oracle collections, such as nested tables. This part also provides chapters that discuss the result set functionality in JDBC, JDBC row sets, and globalization support provided by Oracle JDBC drivers.

Part IV contains the following chapters:

- Accessing and Manipulating Oracle Data
- Java Streams in JDBC
- Working with Oracle Object Types
- Working with LOBs and BFILEs
- Using Oracle Object References
- Working with Oracle Collections
- Result Set
- JDBC RowSets
- Globalization Support



12

Accessing and Manipulating Oracle Data

This chapter describes Oracle extensions (oracle.sql.* formats) and compares them to standard Java formats (java.sql.*). Using Oracle extensions involves casting your result sets and statements to OracleResultSet, OracleStatement, OraclePreparedStatement, and OracleCallableStatement, as appropriate, and using the getOracleObject, setOracleObject, getXXX, and setXXX methods of these classes, where XXX corresponds to the types in the oracle.sql package.

This chapter covers the following topics:

- Data Type Mappings
- Data Conversion Considerations
- Result Set and Statement Extensions
- Comparison of Oracle get and set Methods to Standard JDBC
- Using Result Set Metadata Extensions
- About Using SQL CALL and CALL INTO Statements

12.1 Data Type Mappings

The Oracle JDBC drivers support standard JDBC types as well as Oracle-specific data types. This section documents standard and Oracle-specific SQL-Java default type mappings. This section contains the following topics:

- Table of Mappings
- Notes Regarding Mappings

12.1.1 Table of Mappings

This section describes the default mappings between SQL data types, JDBC type codes, standard Java types, and Oracle extended types.

The SQL Data Types column lists the SQL types that exist in Oracle Database Release 23c. The JDBC Type Codes column lists data type codes supported by the JDBC standard and defined in the java.sql.Types class or by Oracle in the oracle.jdbc.OracleTypes class. For standard type codes, the codes are identical in these two classes.

The Standard Java Types column lists standard types defined in the Java language. The Oracle Extension Java Types column lists the <code>oracle.sql.*</code> Java types that correspond to each SQL data type in the database. These are Oracle extensions that let you retrieve all SQL data in the form of an <code>oracle.sql.*</code> Java type.



Note:

In general, the Oracle JDBC drivers are optimized to manipulate SQL data using the standard JDBC types. In a few specialized cases, it may be advantageous to use the Oracle extension classes that are available in the oracle.sql package. But, Oracle strongly recommends to use the standard JDBC types instead of Oracle extensions, whenever possible.

SQL Data Types	JDBC Type Codes	Standard Java Types	Oracle Extension Java Types
BOOLEAN	java.sql.Types.BOOLEAN	java.lang.Boolean or java.lang.boolean	oracle.sql.BOOLEAN
CHAR	java.sql.Types.CHAR	java.lang.String	oracle.sql.CHAR
VARCHAR2	java.sql.Types.VARCHAR	java.lang.String	oracle.sql.CHAR
LONG	java.sql.Types.LONGVARCHAR	java.lang.String	oracle.sql.CHAR
NUMBER	java.sql.Types.NUMERIC	java.math.BigDecimal	oracle.sql.NUMBER
NUMBER	java.sql.Types.DECIMAL	java.math.BigDecimal	oracle.sql.NUMBER
NUMBER	java.sql.Types.BIT	boolean	oracle.sql.NUMBER
NUMBER	java.sql.Types.TINYINT	byte	oracle.sql.NUMBER
NUMBER	java.sql.Types.SMALLINT	short	oracle.sql.NUMBER
NUMBER	java.sql.Types.INTEGER	int	oracle.sql.NUMBER
NUMBER	java.sql.Types.BIGINT	long	oracle.sql.NUMBER
NUMBER	java.sql.Types.REAL	float	oracle.sql.NUMBER
NUMBER	java.sql.Types.FLOAT	double	oracle.sql.NUMBER
NUMBER	java.sql.Types.DOUBLE	double	oracle.sql.NUMBER
RAW	java.sql.Types.BINARY	byte[]	oracle.sql.RAW
RAW	java.sql.Types.VARBINARY	byte[]	oracle.sql.RAW
LONGRAW	java.sql.Types.LONGVARBINARY	byte[]	oracle.sql.RAW
DATE	java.sql.Types.DATE	java.sql.Date	oracle.sql.DATE
DATE	java.sql.Types.TIME	java.sql.Time	oracle.sql.DATE
TIMESTAMP	java.sql.Types.TIMESTAMP	javal.sql.Timestamp	oracle.sql.TIMESTAMP
BLOB	java.sql.Types.BLOB	java.sql.Blob	oracle.jdbc.OracleBlo b ¹
CLOB	java.sql.Types.CLOB	java.sql.Clob	oracle.jdbc.OracleClo b ²
user-defined object	java.sql.Types.STRUCT	java.sql.Struct	oracle.jdbc.OracleStr uct ³
user-defined reference	java.sql.Types.REF	java.sql.Ref	oracle.jdbc.OracleRef ⁴
user-defined collection	java.sql.Types.ARRAY	java.sql.Array	oracle.jdbc.OracleArr ay ⁵

Table 12-1 Default Mappings Between SQL Types and Java Types



SQL Data Types	JDBC Type Codes	Standard Java Types	Oracle Extension Java Types
ROWID	java.sql.Types.ROWID	java.sql.RowId	oracle.sql.ROWID
NCLOB	java.sql.Types.NCLOB	java.sql.NClob	oracle.sql.NCLOB
NCHAR	java.sql.Types.NCHAR	java.lang.String	oracle.sql.CHAR
BFILE	<pre>oracle.jdbc.OracleTypes.BFILE (ORACLE EXTENSION)</pre>	NA	oracle.sql.BFILE
REF CURSOR	oracle.jdbc.OracleTypes.CURSO R	java.sql.ResultSet	oracle.jdbc.OracleRes ultSet
	(ORACLE EXTENSION)		
TIMESTAMP	oracle.jdbc.OracleTypes.TIMES TAMP	java.sql.Timestamp	oracle.sql.TIMESTAMP
	(ORACLE EXTENSION)		
TIMESTAMP WITH TIME ZONE	oracle.jdbc.OracleTypes.TIMES TAMPTZ	java.sql.Timestamp	oracle.sql.TIMESTAMPT Z
	(ORACLE EXTENSION)		
TIMESTAMP WITH LOCAL TIME	oracle.jdbc.OracleTypes.TIMES TAMPLTZ	java.sql.Timestamp	oracle.sql.TIMESTAMPL TZ
ZONE	(ORACLE EXTENSION)		

Table 12-1 (Cont.) Default Mappings Between SQL Types and Java Types

¹ Starting from Oracle Database 12c Release 1 (12.1), the oracle.sql.BLOB class is deprecated and replaced with the oracle.jdbc.OracleBlob interface.

² Starting from Oracle Database 12c Release 1, the oracle.sql.CLOB class is deprecated and is replaced with the oracle.jdbc.OracleClob interface.

³ Starting from Oracle Database 12c Release 1 (12.1), the oracle.sql.STRUCT class is deprecated and replaced with the oracle.jdbc.OracleStruct interface.

⁴ Starting from Oracle Database 12c Release 1 (12.1), the oracle.sql.REF class is deprecated and replaced with the oracle.jdbc.OracleRef interface.

⁵ Starting from Oracle Database 12c Release 1 (12.1), the oracle.sql.ARRAY class is deprecated and replaced with the oracle.jdbc.OracleArray interface.

Note:

For database versions, such as 8.1.7, which do not support the TIMESTAMP data type, TIMESTAMP is mapped to DATE.

Related Topics

- Standard Types Versus Oracle Types
- Supported SQL-JDBC Data Type Mappings
 This section lists all the possible Java types to which a given SQL data type can have a
 valid mapping.



12.1.2 Notes Regarding Mappings

This section provides further details regarding mappings for NUMBER and user-defined types.

NUMBER Types

For the different type codes that an Oracle NUMBER value can correspond to, call the getter routine that is appropriate for the size of the data for mapping to work properly. For example, call getByte to get a Java tinyint value for an item x, where -128 < x < 128.

User-Defined Types

User-defined types, such as objects, object references, and collections, map by default to weak Java types, such as java.sql.Struct, but alternatively can map to strongly typed custom Java classes. Custom Java classes can implement one of two interfaces:

- The standard java.sql.SQLData
- The Oracle-specific oracle.jdbc.OracleData

Related Topics

- About Mapping Oracle Objects
- About Creating and Using Custom Object Classes for Oracle Objects

12.2 Data Conversion Considerations

When JDBC programs retrieve SQL data into Java, you can use standard Java types, or you can use types of the <code>oracle.sql</code> package. This section covers the following topics:

- Standard Types Versus Oracle Types
- About Converting SQL NULL Data
- About Testing for NULLs

12.2.1 Standard Types Versus Oracle Types

The Oracle data types in oracle.sql store data in the same bit format as used by the database. In versions of the Oracle JDBC drivers prior to Oracle Database 10*g*, the Oracle data types were generally more efficient. Starting from Oracle Database 10*g*, the JDBC drivers were substantially updated. As a result, in most cases the standard Java types are preferred to the data types in oracle.sql.*. In particular, java.lang.String is much more efficient than oracle.sql.CHAR.

In general, Oracle recommends that you use the Java standard types. The exceptions to this are:

• Use the oracle.jdbc.OracleData rather than the java.sql.SqlData if the OracleData functionality better suits your needs.



• Use oracle.sql.NUMBER rather than java.lang.Double if you need to retain the exact values of floating point numbers. Oracle NUMBER is a decimal representation and Java Double and Float are binary representations. Conversion from one format to the other can result in slight variations in the actual value represented. Additionally, the range of values that can be represented using the two formats is different.

Use oracle.sql.NUMBER rather than java.math.BigDecimal when performance is critical and you are not manipulating the values, just reading and writing them.

• Use oracle.sql.DATE or oracle.sql.TIMESTAMP if you are using a JDK version earlier than JDK 6. Use java.sql.Date or java.sql.Timestamp if you are using JDK 6 or a later version.

Note:

Due to a bug in all versions of Java prior to JDK 6, construction of java.lang.Date and java.lang.Timestamp objects is slow, especially in multithreaded environments. This bug is fixed in JDK 6.

- Use oracle.sql.CHAR only when you have data from some external source, which has been represented in an Oracle character set encoding. In all other cases, you should use java.lang.String.
- STRUCT, ARRAY, BLOB, CLOB, REF, and ROWID are all the implementation classes of the corresponding JDBC standard interface types. So, there is no benefit of using the Oracle extension types as they are identical to the JDBC standard types.
- BFILE, TIMESTAMPTZ, and TIMESTAMPLTZ have no representation in the JDBC standard. You must use these Oracle extensions.
- In all other cases, you should use the standard JDBC type rather than the Oracle extensions.

Note:

If you convert an oracle.sql data type to a Java standard data type, then the benefits of using the oracle.sql data type are lost.

12.2.2 About Converting SQL NULL Data

Java represents a SQL NULL datum by the Java value null. Java data types fall into two categories: primitive types, such as byte, int, and float, and object types, such as class instances. The primitive types cannot represent null. Instead, they store null as the value zero, as defined by the JDBC specification. This can lead to ambiguity when you try to interpret your results.

In contrast, Java object types can represent null. The Java language defines an object container type corresponding to every primitive type that can represent null. The object container types must be used as the targets for SQL data to detect SQL NULL without ambiguity.



12.2.3 About Testing for NULLs

You cannot use a relational operator to compare NULL values with each other or with other values. For example, the following SELECT statement does not return any row even if the COMMISSION PCT column contains one or more NULL values.

```
PreparedStatement pstmt = conn.prepareStatement(
   "SELECT * FROM EMPLOYEES WHERE COMMISSION_PCT = ?");
pstmt.setNull(1, java.sql.Types.VARCHAR);
```

The next example shows how to compare values for equality when some return values might be <code>NULL</code>. The following code returns all the <code>FIRST_NAME</code> from the <code>EMPLOYEES</code> table that are <code>NULL</code>, if there is no value of 205 for <code>COMM</code>.

```
PreparedStatement pstmt = conn.prepareStatement("SELECT FIRST_NAME FROM
EMPLOYEES
WHERE COMMISSION_PCT =? OR ((COMM IS NULL) AND (? IS NULL))");
pstmt.setBigDecimal(1, new BigDecimal(205));
pstmt.setNull(2, java.sql.Types.VARCHAR);
```

12.3 Result Set and Statement Extensions

The Statement object returns a java.sql.ResultSet. If you want to apply only standard JDBC methods to the object, then keep it as a ResultSet type. However, if you want to use the Oracle extensions on the object, then you must cast it to OracleResultSet. All of the Oracle Result Set extensions are in the oracle.jdbc.OracleResultSet interface and all the Statement extensions are in the oracle.jdbc.OracleStatement interface.

For example, assuming you have a standard <code>Statement</code> object <code>stmt</code>, do the following if you want to use only standard JDBC <code>ResultSet</code> methods:

ResultSet rs = stmt.executeQuery("SELECT * FROM employees");

If you need the extended functionality provided by the Oracle extensions to JDBC, you can select the results into a standard ResultSet variable and then cast that variable to OracleResultSet later.

Key extensions to the result set and statement classes include the getOracleObject and setOracleObject methods, used to access and manipulate data in oracle.sql.* formats.

12.4 Comparison of Oracle get and set Methods to Standard JDBC

This section describes get and set methods, particularly the JDBC standard getObject and setObject methods and the Oracle-specific getOracleObject and setOracleObject methods, and how to access data in oracle.sql.* format compared with Java format.

You can use the standard getXXX methods for all Oracle SQL types.

This section covers the following topics:



- Standard getObject Method
- Oracle getOracleObject Method
- Summary of getObject and getOracleObject Return Types
- Other getXXX Methods
- Data Types For Returned Objects from getObject and getXXX
- The setObject and setOracleObject Methods
- Other setXXX Methods

Note:

You cannot qualify a column name with a table name and pass it as a parameter to the get*XXX* method. For example:

```
ResultSet rset = stmt.executeQuery("SELECT employees.department_id,
department.department_id FROM employees, department");
rset.getInt("employees.department_id");
```

The getInt method in the preceding code will throw an exception. To uniquely identify the columns in the getXXX method, you can either use column index or specify column aliases in the query and use these aliases in the getXXX method.

12.4.1 Standard getObject Method

The standard getObject method of a result set or callable statement has a return type of java.lang.Object. The class of the object returned is based on its SQL type, as follows:

- For SQL data types that are not Oracle-specific, the getObject method returns the default Java type corresponding to the SQL type of the column, following the mapping in the JDBC specification.
- For Oracle-specific data types, getObject returns an object of the appropriate oracle.sql.* class, such as oracle.sql.ROWID.
- For Oracle database objects, getObject returns a Java object of the class specified in your type map. Type maps specify a mapping from database named types to Java classes. The getObject(parameter_index) method uses the default type map of the connection. The getObject(parameter_index, map) enables you to pass in a type map. If the type map does not provide a mapping for a particular Oracle object, then getObject returns an oracle.sql.OracleStruct object.

12.4.2 Oracle getOracleObject Method

If you want to retrieve data from a result set or callable statement as an oracle.sql.* object, then you must follow a special process. For an OracleResultSet object, you must cast the Result Set to oracle.jdbc.OracleResultSet and then call getOracleObject instead of getObject. The same applies to CallableStatement and oracle.jdbc.OracleCallableStatement.

The return type of getOracleObject is oracle.sql.Datum. The actual returned object is an instance of the appropriate oracle.sql.* class. The method signature is:



public oracle.sql.Datum getOracleObject(int parameter_index)

When you retrieve data into a Datum variable, you can use the standard Java instanceof operator to determine which oracle.sql.* type it really is.

Example: Using getOracleObject with a Result Set

The following example creates a table that contains a column of CHAR data and a column containing a BFILE locator. A SELECT statement retrieves the contents of the table as a result set. The getOracleObject then retrieves the CHAR data into the char_datum variable and the BFILE locator into the bfile_datum variable. Note that because getOracleObject returns a Datum object, the return values must be cast to CHAR and BFILE, respectively.

```
stmt.execute ("CREATE TABLE bfile_table (x VARCHAR2 (30), b BFILE)");
stmt.execute
    ("INSERT INTO bfile_table VALUES ('one', BFILENAME ('TEST_DIR', 'file1'))");
ResultSet rset = stmt.executeQuery ("SELECT * FROM bfile_table");
while (rset.next ())
{
    CHAR char_datum = (CHAR) ((OracleResultSet)rset).getOracleObject (1);
    BFILE bfile_datum = (BFILE) ((OracleResultSet)rset).getOracleObject (2);
    ...
}
```

Example: Using getOracleObject in a Callable Statement

The following example prepares a call to the procedure <code>myGetDate</code>, which associates a character string with a date. The program passes "HR" to the prepared call and registers the DATE type as an output parameter. After the call is run, <code>getOracleObject</code> retrieves the date associated with "HR". Note that because <code>getOracleObject</code> returns a Datum object, the results are cast to DATE.

12.4.3 Summary of getObject and getOracleObject Return Types

The following table lists the underlying return types for the getObject and getOracleObject methods for each Oracle SQL type.

Keep in mind the following when you use these methods:

- getObject always returns data into a java.lang.Object instance
- getOracleObject always returns data into an oracle.sql.Datum instance

You must cast the returned object to use any special functionality.



Oracle SQL Type	getObject Underlying Return Type	getOracleObject Underlying Return Type	
CHAR	String	oracle.sql.CHAR	
VARCHAR2	String	oracle.sql.CHAR	
NCHAR	String	oracle.sql.CHAR	
LONG	String	oracle.sql.CHAR	
NUMBER	java.math.BigDecimal	oracle.sql.NUMBER	
RAW	byte[]	oracle.sql.RAW	
LONGRAW	byte[]	oracle.sql.RAW	
DATE	java.sql.Date	oracle.sql.DATE	
TIMESTAMP	java.sql.Timestamp ¹	oracle.sql.TIMESTAMP	
TIMESTAMP WITH TIME ZONE	oracle.sql.TIMESTAMPTZ	oracle.sql.TIMESTAMPTZ	
TIMESTAMP WITH LOCAL TIME ZONE	oracle.sql.TIMESTAMPLTZ	oracle.sql.TIMESTAMPLTZ	
BINARY_FLOAT	java.lang.Float	oracle.sql.BINARY_FLOAT	
BINARY_DOUBLE	java.lang.Double	oracle.sql.BINARY_DOUBLE	
INTERVAL DAY TO SECOND	oracle.sql.INTERVALDS	oracle.sql.INTERVALDS	
INTERVAL YEAR TO MONTH	oracle.sql.INTERVALYM	oracle.sql.INTERVALYM	
ROWID	oracle.sql.ROWID	oracle.sql.ROWID	
REF CURSOR	java.sql.ResultSet	(not supported)	
BLOB	oracle.jdbc.OracleBlob ²	oracle.jdbc.OracleBlob	
CLOB	oracle.jdbc.OracleClob ³	oracle.jdbc.OracleClob	
NCLOB	java.sql.NClob	oracle.sql.NCLOB	
BFILE	oracle.sql.BFILE	oracle.sql.BFILE	
Oracle object	class specified in type map	oracle.jdbc.OracleStruct	
	<pre>or oracle.sql.OracleStruct⁴ (if no type map entry)</pre>		
Oracle object reference	oracle.jdbc.OracleRef ⁵	oracle.jdbc.OracleRef	
collection (varray or nested table)	oracle.jdbc.OracleArray ⁶	oracle.sql.ARRAY	

 Table 12-2
 getObject and getOracleObject Return Types

ResultSet.getObject returns java.sql.Timestamp only if the oracle.jdbc.J2EE13Compliant connection property is set to TRUE, else the method returns oracle.sql.TIMESTAMP.

² Starting from Oracle Database 12c Release 1 (12.1), the oracle.sql.BLOB class is deprecated and replaced with the oracle.jdbc.OracleBlob interface.

- ³ Starting from Oracle Database 12c Release 1 (12.1), the oracle.sql.CLOB class is deprecated and replaced with the oracle.jdbc.OracleClob interface.
- ⁴ Starting from Oracle Database 12c Release 1 (12.1), the oracle.sql.STRUCT class is deprecated and replaced with the oracle.jdbc.OracleStruct interface.

- ⁵ Starting from Oracle Database 12c Release 1, the oracle.sql.REF class is deprecated and is replaced with the oracle.jdbc.OracleRef interface.
- ⁶ Starting from Oracle Database 12c Release 1, the oracle.sql.ARRAY class is deprecated and replaced with the oracle.jdbc.OracleArray interface.

Note:

The ResultSet.getObject method returns java.sql.Timestamp for the TIMESTAMP SQL type, only when the connection property oracle.jdbc.J2EE13Compliant is set to TRUE. This property has to be set when the connection is obtained. If this connection property is not set or if it is set after the connection is obtained, then the ResultSet.getObject method returns oracle.sql.TIMESTAMP for the TIMESTAMP SQL type.

The oracle.jdbc.J2EE13Compliant connection property can also be set without changing the code in the following ways:

- Including the ojdbc6dms.jar or ojdbc7dms.jar files in the CLASSPATH. These files set oracle.jdbc.J2EE13Compliant to TRUE by default. These are specific to the Oracle Application Server release and are not available as part of the general JDBC release. They are located in \$ORACLE HOME/jdbc/lib.
- Setting the system property by calling the java command with the flag Doracle.jdbc.J2EE13Compliant=true. For example,

java -Doracle.jdbc.J2EE13Compliant=true ...

When the J2EE13Compliant is set to TRUE the action is as in Table B-3 of the JDBC specification.

Related Topics

• Supported SQL-JDBC Data Type Mappings This section lists all the possible Java types to which a given SQL data type can have a valid mapping.

12.4.4 Other getXXX Methods

Standard JDBC provides a getXXX for each standard Java type, such as getByte, getInt, getFloat, and so on. Each of these returns exactly what the method name implies.

In addition, the OracleResultSet and OracleCallableStatement interfaces provide a full complement of getXXX methods corresponding to all the oracle.sql.* types. Each getXXX method returns an oracle.sql.XXX object. For example, getROWID returns an oracle.sql.ROWID object.

There is no performance advantage in using the specific getXXX methods. However, they do save you the trouble of casting, because the return type is specific to the object being returned.

This section covers the following topics:

Return Types of getXXX Methods



Special Notes about getXXX Methods

12.4.4.1 Return Types of getXXX Methods

Refer to the JDBC Javadoc to know the return types for each getXXX method and also which are Oracle extensions under Java Development Kit (JDK) 6. You must cast the returned object to OracleResultSet or OracleCallableStatement to use methods that are Oracle extensions.

12.4.4.2 Special Notes about getXXX Methods

This section provides additional details about some getXXX methods.

getBigDecimal

JDBC 2.0 simplified method signatures for the getBigDecimal method. The previous input signatures were:

(int columnIndex, int scale) or (String columnName, int scale)

The simplified input signature is:

(int columnIndex) or (String columnName)

The scale parameter, used to specify the number of digits to the right of the decimal, is no longer necessary. The Oracle JDBC drivers retrieve numeric values with full precision.

getBoolean

If the column from which you are trying to fetch the data is not of BOOLEAN type, which is introduced in Oracle Database 23c Release, then the use of the getBoolean method results in a data type conversion. Apart from the BOOLEAN type, the getBoolean method also supports the following columns:

- INT
- CHAR
- NCHAR
- NUMBER
- VARCHAR
- NVARCHAR

When you apply this method to the supported data types, then the getBoolean method interprets any zero value as false and any other value as true. When applied to any other column, apart from the supported columns, the getBoolean method raises the exception java.lang.NumberFormatException.

See Also:

Class oracle.jdbc.OracleTypes for more information about the BOOLEAN data type



12.4.5 Data Types For Returned Objects from getObject and getXXX

The return type of getObject is java.lang.Object. The returned value is an instance of a subclass of java.lang.Object. Similarly, the return type of getOracleObject is oracle.sql.Datum, and the class of the returned value is a subclass of oracle.sql.Datum. You typically cast the returned object to the appropriate class to use particular methods and functionality of that class.

In addition, you have the option of using a specific getXXX method instead of the generic getObject or getOracleObject methods. The getXXX methods enable you to avoid casting, because the return type of getXXX corresponds to the type of object returned. For example, the return type of getCLOB is oracle.sql.CLOB, as opposed to java.lang.Object.

Example of Casting Return Values

This example assumes that you have fetched data of the NUMBER type as the first column of a result set. Because you want to manipulate the NUMBER data without losing precision, cast your result set to OracleResultSet and use getOracleObject to return the NUMBER data in oracle.sql.* format. If you do not cast your result set, then you have to use getObject, which returns your numeric data into a Java Float and loses some of the precision of your SQL data.

The getOracleObject method returns an oracle.sql.NUMBER object into an oracle.sql.Datum return variable unless you cast the output. Cast the getOracleObject output to oracle.sql.NUMBER if you want to use a NUMBER return variable and any of the special functionality of that class.

NUMBER x = (NUMBER) ors.getOracleObject(1);

12.4.6 The setObject and setOracleObject Methods

Just as there is a standard getObject and Oracle-specific getOracleObject in result sets and callable statements, there are also standard setObject and Oracle-specific setOracleObject methods in OraclePreparedStatement and OracleCallableStatement. The setOracleObject methods take oracle.sql.* input parameters.

To bind standard Java types to a prepared statement or callable statement, use the setObject method, which takes a java.lang.Object as input. The setObject method does support a few of the oracle.sql.* types. However, the method has been implemented so that you can enter instances of the oracle.sql.* classes that correspond to the following JDBC standard types: Blob, Clob, Struct, Ref, and Array.

To bind oracle.sql.* types to a prepared statement or callable statement, use the setOracleObject method, which takes a subclass of oracle.sql.Datum as input. To use setOracleObject, you must cast your prepared statement or callable statement to OraclePreparedStatement Or OracleCallableStatement.

Example of Using setObject and setOracleObject

For a prepared statement, the setOracleObject method binds the oracle.sql.CHAR
data represented by the charVal variable to the prepared statement. To bind the
oracle.sql.* data, the prepared statement must be cast to



OraclePreparedStatement. Similarly, the setObject method binds the Java String data represented by the variable strVal.

```
PreparedStatement ps= conn.prepareStatement("text_of_prepared_statement");
((OraclePreparedStatement)ps).setOracleObject(1,charVal);
ps.setObject(2,strVal);
```

12.4.7 Other setXXX Methods

As with the getXXX methods, there are several specific setXXX methods. Standard setXXX methods are provided for binding standard Java types, and Oracle-specific setXXX methods are provided for binding Oracle-specific types.

Similarly, there are two forms of the setNull method:

void setNull(int parameterIndex, int sqlType)

This is specified in the standard java.sql.PreparedStatement interface. This signature takes a parameter index and a SQL type code defined by the java.sql.Types or oracle.jdbc.OracleTypes class. Use this signature to set an object other than a REF, ARRAY, or STRUCT to NULL.

• void setNull(int parameterIndex, int sqlType, String sql_type_name)

With JDBC 2.0, this signature is also specified in the standard java.sql.PreparedStatement interface. This method takes a SQL type name in addition to a parameter index and a SQL type code. Use this method when the SQL type code is java.sql.Types.REF, ARRAY, or STRUCT. If the type code is other than REF, ARRAY, or STRUCT, then the given SQL type name is ignored.

Similarly, the registerOutParameter method has a signature for use with REF, ARRAY, or STRUCT data:

Binding Oracle-specific types using the appropriate setXXX methods, instead of the methods used for binding standard Java types, may offer some performance advantage.

This section covers the following topics:

- Input Data Binding
- Method setFixedCHAR for Binding CHAR Data into WHERE Clauses

12.4.7.1 Input Data Binding

There are three way to bind data for input:

- Direct binding where the data itself is placed in a bind buffer
- Stream binding where the data is streamed
- LOB binding where a temporary lob is created, the data placed in the LOB using the LOB APIs, and the bytes of the LOB locator are placed in the bind buffer

The three kinds of binding have some differences in performance and have an impact on batching. Direct binding is fast and batching is fine. Stream binding is slower, may require multiple round trips, and turns batching off. LOB binding is very slow and requires many



round trips. Batching works, but might be a bad idea. They also have different size limits, depending on the type of the SQL statement.

For SQL parameters, the length of standard parameter types, such as RAW and VARCHAR2, is fixed by the size of the target column. For PL/SQL parameters, the size is limited to a fixed number of bytes, which is 32766.

In Oracle Database 10g release 2, certain changes were made to the setString, setCharacterStream, setAsciiStream, setBytes, and setBinaryStream methods of PreparedStatement. The original behavior of these APIs were:

- setString: Direct bind of characters
- setCharacterStream: Stream bind of characters
- setAsciiStream: Stream bind of bytes
- setBytes: Direct bind of bytes
- setBinaryStream: Stream bind of bytes

Starting from Oracle Database 10*g* Release 2, automatic switching between binding modes, based on the data size and on the type of the SQL statement is provided.

setBytes and setBinaryStream

For SQL, direct bind is used for size up to 2000 and stream bind for larger.

For PL/SQL direct bind is used for size up to 32766 and LOB bind is used for larger.

setString, setCharacterStream, and setAsciiStream

For SQL, direct bind is used up to 32766 Java characters and stream bind is used for larger. This is independent of character set.

For PL/SQL, you must be careful about the byte size of the character data in the database character set or the national character set depending on the setting of the form of use parameter. Direct bind is used for data where the byte length is less than 32766 and LOB bind is used for larger.

For fixed length character sets, multiply the length of the Java character data by the fixed character size in bytes and compare that to the restrictive values. For variable length character sets, there are three cases based on the Java character length, as follows:

- If character length is less than 32766 divided by the maximum character size, then direct bind is used.
- If character length is greater than 32766 divided by the minimum character size, then LOB bind is used.
- If character length is in between and if the actual length of the converted bytes is less than 32766, then direct bind is used, else LOB bind is used.

Note:

When a PL/SQL procedure is embedded in a SQL statement, the binding action is different.



The server-side internal driver has the following additional limitations:

- setString, setCharacterStream, and setASCIIStream APIs are not supported for SQL CLOB columns when the data size in characters is over 32767 bytes
- setBytes and setBinaryStream APIs are not supported for SQL BLOB columns when the data size is over 32767 bytes

Note:

Do not use these APIs with the server-side internal driver, without careful checking of the data size in client code.

Related Topics

Data Interface for LOBs

The data interface for LOBs includes a set of Java and OCI APIs that are extended to work with LOB data types.

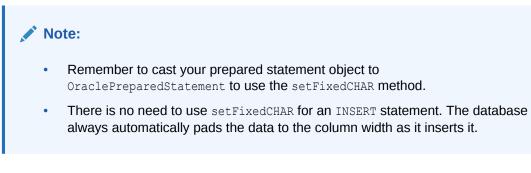
See Also:

JDBC Release Notes for further discussion and possible workarounds

12.4.7.2 Method setFixedCHAR for Binding CHAR Data into WHERE Clauses

CHAR data in the database is padded to the column width. This leads to a limitation in using the setCHAR method to bind character data into the WHERE clause of a SELECT statement. The character data in the WHERE clause must also be padded to the column width to produce a match in the SELECT statement. This is especially troublesome if you do not know the column width.

To remedy this, Oracle has added the setFixedCHAR method to the OraclePreparedStatement class. This method runs a non-padded comparison.



Example

The following example demonstrates the difference between the setCHAR and setFixedCHAR methods.



```
/* Schema is :
create table my table (coll char(10));
insert into my table values ('JDBC');
*/
PreparedStatement pstmt = conn.prepareStatement
                    ("select count(*) from my table where col1 = ?");
pstmt.setString (1, "JDBC"); // Set the Bind Value
                              // This will print " No of rows are 0"
runQuery (pstmt);
                             ", null);
CHAR ch = new CHAR("JDBC
((OraclePreparedStatement)pstmt).setCHAR(1, ch); // Pad it to 10 bytes
runQuery (pstmt);
                              // This will print "No of rows are 1"
 ((OraclePreparedStatement)pstmt).setFixedCHAR(1, "JDBC");
                              // This will print "No of rows are 1"
 runQuery (pstmt);
void runQuery (PreparedStatement ps)
{
  // Run the Query
  ResultSet rs = pstmt.executeQuery ();
  while (rs.next())
    System.out.println("No of rows are " + rs.getInt(1));
  rs.close();
  rs = null;
1
```

12.5 Using Result Set Metadata Extensions

The oracle.jdbc.OracleResultSetMetaData interface is JDBC 2.0-compliant but does not implement the getSchemaName and getTableName methods because Oracle Database does not make this feasible.

The following code snippet uses several of the methods in the OracleResultSetMetadata interface to retrieve the number of columns from the EMPLOYEES table and the numerical type and SQL type name of each column:

```
DatabaseMetaData dbmd = conn.getMetaData();
ResultSet rset = dbmd.getTables("", "HR", "EMPLOYEES", null);
while (rset.next())
{
    OracleResultSetMetaData orsmd = ((OracleResultSet)rset).getMetaData();
    int numColumns = orsmd.getColumnCount();
    System.out.println("Num of columns = " + numColumns);
    for (int i=0; i<numColumns; i++)
    {
        System.out.print ("Column Name=" + orsmd.getColumnName (i+1));
        System.out.print (" Type=" + orsmd.getColumnType (i + 1) );
        System.out.println (" Type Name=" + orsmd.getColumnTypeName (i + 1));
    }
}
```

The program returns the following output:

```
Num of columns = 5
Column Name=TABLE_CAT Type=12 Type Name=VARCHAR2
```



```
Column Name=TABLE_SCHEM Type=12 Type Name=VARCHAR2
Column Name=TABLE_NAME Type=12 Type Name=VARCHAR2
Column Name=TABLE_TYPE Type=12 Type Name=VARCHAR2
Column Name=TABLE_REMARKS Type=12 Type Name=VARCHAR2
```

12.6 About Using SQL CALL and CALL INTO Statements

You can use the CALL statement to execute a routine from within SQL in the following two ways:

Note:

A routine is a procedure or a function that is standalone or is defined within a type or package. You must have EXECUTE privilege on the standalone routine or on the type or package in which the routine is defined.

- By issuing a call to the routine itself by name or by using the routine clause
- By using an object access expression inside the type of an expression

You can specify one or more arguments to the routine, if the routine takes arguments. You can use positional, named, or mixed notation for argument.

CALL INTO Statement

The INTO clause applies only to calls to functions. You can use the following types of variables with this clause:

- Host variable
- Indicator variable

See Also:

Oracle Database SQL Language Reference for more information

PL/SQL Blocks

The basic unit in PL/SQL is a block. All PL/SQL programs are made up of blocks, which can be nested within each other. A PL/SQL block has three parts: a declarative part, an executable part, and an exception-handling part. You get the following advantages by using PL/SQL blocks in your application:

- Better performance
- Higher productivity
- Full portability
- Tight integration with Oracle
- Tight security



13 Java Streams in JDBC

This chapter describes how the Oracle Java Database Connectivity (JDBC) drivers handle Java streams for several data types. Data streams enable you to read LONG column data of up to 2 gigabytes (GB).

This chapter covers the following topics:

- Overview of Java Streams
- About Streaming LONG or LONG RAW Columns
- About Streaming CHAR_VARCHAR_ or RAW Columns
- About Streaming LOBs and External Files
- Relation Between Data Streaming and Multiple Columns
- Relation Between Streaming and Row Prefetching
- Closing a Stream
- Notes and Precautions on Streams

13.1 Overview of Java Streams

Oracle JDBC drivers support the manipulation of data streams in either direction between server and client. The drivers support all stream conversions: binary, ASCII, and Unicode. Following is a brief description of each type of stream:

Binary

Used for RAW bytes of data, and corresponds to the getBinaryStream method

ASCII

Used for ASCII bytes in ISO-Latin-1 encoding, and corresponds to the getAsciiStream method

Unicode

Used for Unicode bytes with the UTF-16 encoding, and corresponds to the getUnicodeStream method

The getBinaryStream, getAsciiStream, and getUnicodeStream methods return the bytes of data in an InputStream object.

Note:

Starting from Oracle Database 12c Release 1 (12.1), the CONNECTION_PROPERTY_STREAM_CHUNK_SIZE is deprecated and the driver does not use it internally for setting the stream chunk size.



See Also: Working with LOBs and BFILEs

13.2 About Streaming LONG or LONG RAW Columns

This section covers the following topics:

- Overview of Streaming LONG or LONG RAW Columns
- LONG RAW Data Conversions
- LONG Data Conversions
- Examples:Streaming LONG RAW Data
- About Avoiding Streaming for LONG or LONG RAW

13.2.1 Overview of Streaming LONG or LONG RAW Columns

When a query selects one or more LONG or LONG RAW columns, the JDBC driver transfers these columns to the client in streaming mode. In streaming mode, the JDBC driver does not read the column data from the network for LONG or LONG RAW columns, until required. The column data remains in the network communications channel until your code calls a getXXX method to read the column data. Even after the call, the column data is read only as needed to populate return value from the getXXX call. Because the column data remains in the communications channel, the streaming mode interferes with all other use of the connection. Any use of the connection, other than reading the column data, will discard the column data from the channel. While the streaming mode makes efficient use of memory and minimizes network round trips, it interferes with many other database operations.

Note:

Oracle recommends avoiding LONG and LONG RAW columns. Use LOB instead.

To access the data in a LONG column, you can get the column as a Java InputStream object and use the read method of the InputStream object. As an alternative, you can get the data as a String or byte array. In this case, the driver will do the streaming for you.

You can get LONG and LONG RAW data with any of the three stream types. The driver performs conversions for you, depending on the character set of the database and the driver.



Note:

Do not create tables with LONG columns. Use large object (LOB) columns, CLOB, NCLOB, and BLOB, instead. LONG columns are supported only for backward compatibility. Oracle recommends that you convert existing LONG columns to LOB columns. LOB columns are subject to far fewer restrictions than LONG columns.

13.2.2 LONG RAW Data Conversions

A call to getBinaryStream returns RAW data. A call to getAsciiStream converts the RAW data to hexadecimal and returns the ASCII representation. A call to getUnicodeStream converts the RAW data to hexadecimal and returns the Unicode characters.

13.2.3 LONG Data Conversions

When you get LONG data with getAsciiStream, the drivers assume that the underlying data in the database uses an US7ASCII or WE8ISO8859P1 character set. If the assumption is true, then the drivers return bytes corresponding to ASCII characters. If the database is not using an US7ASCII or WE8ISO8859P1 character set, a call to getAsciiStream returns meaningless information.

When you get LONG data with getUnicodeStream, you get a stream of Unicode characters in the UTF-16 encoding. This applies to all underlying database character sets that Oracle supports.

When you get LONG data with getBinaryStream, there are two possible cases:

- If the driver is JDBC OCI and the *client* character set is *not* US7ASCII or WE8IS08859P1, then a call to getBinaryStream returns UTF-8. If the *client* character set is US7ASCII or WE8IS08859P1, then the call returns a US7ASCII stream of bytes.
- If the driver is JDBC Thin and the *database* character set is *not* US7ASCII or WE8IS08859P1, then a call to getBinaryStream returns UTF-8. If the server-side character set is US7ASCII or WE8IS08859P1, then the call returns a US7ASCII stream of bytes.

Note:

Receiving LONG or LONG RAW columns as a stream requires you to pay special attention to the order in which you retrieve columns from the database.

The following table summarizes LONG and LONG RAW data conversions for each stream type.



Data type	BinaryStream	AsciiStream	UnicodeStrea m
LONG	Bytes representing characters in Unicode UTF-8. The bytes can represent characters in US7ASCII or WE8IS08859P1 if the database character set is US7ASCII or WE8IS08859P1.	Bytes representing characters in ISO-Latin-1 (WE8ISO8859P1) encoding	Bytes representing characters in Unicode UTF-16 encoding
LONG RAW	unchanged data	ASCII representation of hexadecimal bytes	Unicode representation of hexadecimal bytes

Related Topics

- Globalization Support
- Relation Between Data Streaming and Multiple Columns

13.2.4 Examples: Streaming LONG RAW Data

One of the features of a getXXXStream method is that it enables you to fetch data incrementally. In contrast, getBytes fetches all the data in one call. This section contains two examples of getting a stream of binary data. The first version uses the getBinaryStream method to obtain LONG RAW data, and the second version uses the getBytes method.

Getting a LONG RAW Data Column with getBinaryStream

This example writes the contents of a LONG RAW column to a file on the local file system. In this case, the driver fetches the data incrementally.

The following code creates the table that stores a column of LONG RAW data associated with the name LESLIE:

```
-- SQL code:
create table streamexample (NAME varchar2 (256), GIFDATA long raw);
insert into streamexample values ('LESLIE', '00010203040506070809');
```

The following Java code snippet writes the data from the LONG RAW column into a file called leslie.gif:



```
int chunk;
while ((chunk = gif_data.read()) != -1)
file.write(chunk);
}
catch (Exception e)
{
String err = e.toString();
System.out.println(err);
}
finally
{
if file != null()
file.close();
}
```

In this example, the InputStream object returned by the call to getBinaryStream reads the data directly from the database connection.

Getting a LONG RAW Data Column with getBytes

This example gets the content of the GIFDATA column with getBytes instead of getBinaryStream. In this case, the driver fetches all the data in one call and stores it in a byte array. The code snippet is as follows:

```
ResultSet rset2 = stmt.executeQuery
                  ("select GIFDATA from streamexample where NAME='LESLIE'");
// get first row
if (rset2.next())
   // Get the GIF data as a stream from Oracle to the client
   byte[] bytes = rset2.getBytes(1);
   try
   {
      FileOutputStream file = null;
      file = new FileOutputStream ("leslie2.gif");
      file.write(bytes);
   }
   catch (Exception e)
   {
      String err = e.toString();
      System.out.println(err);
   }
   finally
   {
      if file != null()
         file.close();
   }
}
```

Because a LONG RAW column can contain up to 2 gigabytes of data, the getBytes example can use much more memory than the getBinaryStream example. Use streams if you do not know the maximum size of the data in your LONG or LONG RAW columns.



13.2.5 About Avoiding Streaming for LONG or LONG RAW

Note:

Starting from Oracle Database 12c Release 1 (12.1), this method is deprecated.

The JDBC driver automatically streams any LONG and LONG RAW columns. However, there may be situations where you want to avoid data streaming. For example, if you have a very small LONG column, then you may want to avoid returning the data incrementally and, instead, return the data in one call.

To avoid streaming, use the defineColumnType method to redefine the type of the LONG column. For example, if you redefine the LONG or LONG RAW column as VARCHAR or VARBINARY type, then the driver will not automatically stream the data.

If you redefine column types with defineColumnType, then you must declare the types of the columns in the query. If you do not declare the types of the columns, then executeQuery will fail. In addition, you must cast the Statement object to oracle.jdbc.OracleStatement.

As an added benefit, using defineColumnType saves the OCI driver a database roundtrip when running the query. Without defineColumnType, these JDBC drivers must request the data types of the column types. The JDBC Thin driver derives no benefit from defineColumnType, because it always uses the minimum number of round-trips.

Using the example from the previous section, the Statement object stmt is cast to OracleStatement and the column containing LONG RAW data is redefined to be of the type VARBINARAY. The data is not streamed. Instead, it is returned in a byte array. The code snippet is as follows:



13.3 About Streaming CHAR, VARCHAR, or RAW Columns

Note:

Starting from Oracle Database 12c Release 1 (12.1), this method is deprecated..

If you use the defineColumnType Oracle extension to redefine a CHAR, VARCHAR, or RAW column as a LONGVARCHAR or LONGVARBINARY, then you can get the column as a stream. The program will behave as if the column were actually of type LONG or LONG RAW. Note that there is not much point to this, because these columns are usually short.

If you try to get a CHAR, VARCHAR, or RAW column as a data stream without redefining the column type, then the JDBC driver will return a Java InputStream, but no real streaming occurs. In the case of these data types, the JDBC driver fully fetches the data into an inmemory buffer during a call to the <code>executeQuery</code> method or the <code>next</code> method. The <code>getXXXStream</code> entry points return a stream that reads data from this buffer.

13.4 About Streaming LOBs and External Files

The term large object (LOB) refers to a data item that is too large to be stored directly in a database table. Instead, a locator is stored in the database table, which points to the location of the actual data. External files are managed similarly. The JDBC drivers can support the following types through the use of streams:

Binary large object (BLOB)

For unstructured binary data

Character large object (CLOB)

For character data

National Character large object (NCLOB)

For national character data

Binary file (BFILE)

For external files

LOBs and BFILEs behave differently from the other types of streaming data described in this chapter. Instead of storing the actual data in the table, a locator is stored. The actual data can be manipulated using this locator, including reading and writing the data as a stream. Even when streaming, only the chunk of data (defined by a size) is streamed across the network. By contrast, when streaming a LONG or LONG RAW, the entire data is streamed across the network.

Streaming BLOBs, CLOBs, and NCLOBs

When a query fetches one or more BLOB, CLOB, or NCLOB columns, the JDBC driver transfers the data to the client. This data can be accessed as a stream. To manipulate BLOB, CLOB, or NCLOB data from JDBC, use methods in the Oracle extension classes <code>oracle.sql.BLOB</code>, <code>oracle.sql.CLOB</code> and <code>oracle.sql.NCLOB</code>. These classes provide specific functionality, such as reading from the BLOB, CLOB, or NCLOB into an input stream, writing from an output stream



into a BLOB, CLOB, or NCLOB, determining the length of a BLOB, CLOB, or NCLOB, and closing a BLOB, CLOB, or NCLOB.

Note:

Starting from Oracle Database 12*c* Release 1 (12.1), the concrete classes in the oracle.sql package are deprecated and replaced with the interfaces in the oracle.jdbc package. Oracle recommends you to use the methods available in the java.sql package, where possible, for standard compatibility and methods available in the oracle.jdbc package for Oracle specific extensions. Refer to MoS Note 1364193.1 for more information about these interfaces.

See Also: "Data Interface for LOBs"

Streaming BFILEs

An external file, or BFILE, is used to store a locator to a file outside the database. The file can be stored somewhere on the file system of the data server. The locator points to the actual location of the file.

When a query fetches one or more BFILE columns, the JDBC driver transfers the file to the client as required. The data can be accessed as a stream To manipulate BFILE data from JDBC, use methods in the Oracle extension class oracle.sql.BFILE. This class provides specific functionality, such as reading from the BFILE into an input stream, writing from an output stream into a BFILE, determining the length of a BFILE, and closing a BFILE.

13.5 Relation Between Data Streaming and Multiple Columns

If a query fetches multiple columns and one of the columns contains a data stream, then the contents of the columns following the stream column are not available until the stream has been read, and the stream column is no longer available once any following column is read. Any attempt to read a column beyond a streaming column closes the streaming column.

Streaming Example with Multiple Columns

Consider the following code:



```
// get the streaming data
InputStream is = rset.getAsciiStream(2);
// Open a file to store the gif data
FileOutputStream file = new FileOutputStream ("ascii.dat");
// Loop, reading from the ascii stream and
// write to the file
int chunk;
while ((chunk = is.read ()) != -1)
file.write(chunk);
// Close the file
file.close();
//get the number column data
int n = rset.getInt(3);
```

The incoming data for each row has the following shape:

<a date><the characters of the long column><a number>

As you process each row of the result set, you must complete any processing of the stream column before reading the number column.

Bypassing Streaming Data Columns

}

There may be situations where you want to avoid reading a column that contains streaming data. If you do not want to read such data, then call the close method of the stream object. This method discards the stream data and enables the driver to continue reading data from all the columns that contain non-streaming data and follow the column containing streaming data. Even though you are intentionally discarding the stream, it is a good programming practice to retrieve the columns in the same order as in the SELECT statement.

In the following example, the stream data in the LONG column is discarded and the data from only the DATE and NUMBER column is recovered:

```
ResultSet rset = stmt.executeQuery
        ("select DATECOL, LONGCOL, NUMBERCOL from TABLE");
while rset.next()
{
     //get the date
     java.sql.Date date = rset.getDate(1);
     // access the stream data and discard it with close()
     InputStream is = rset.getAsciiStream(2);
     is.close();
     // get the number column data
     int n = rset.getInt(3);
}
```

Related Topics

- About Streaming Data Precautions
- About Streaming LOBs and External Files



13.6 Closing a Stream

You can discard the data from a stream at any time by calling the close method. It is a good programming practice to close the stream when you no longer need it. For example:

```
InputStream is = rset.getAsciiStream(2);
is.close();
```

Note:

Closing a stream has little performance effect on a LONG or LONG RAW column. All of the data still move across the network and the driver must read the bits from the network.

Related Topics

- Relation Between Data Streaming and Multiple Columns
- About Streaming Data Precautions

13.7 Notes and Precautions on Streams

This section discusses several cautionary issues regarding the use of streams:

- About Streaming Data Precautions
- About Using Streams to Avoid Limits on setBytes and setString
- Relation Between Streaming and Row Prefetching

13.7.1 About Streaming Data Precautions

This section describes some of the precautions you must take to ensure that you do not accidentally discard or lose your stream data. The drivers automatically discard stream data if you perform any JDBC operation that communicates with the database, other than reading the current stream. Two common precautions are:

• Use the stream data after you access it.

To recover the data from a column containing a data stream, it is not enough to fetch the column. You must immediately process the contents of the column. Otherwise, the contents will be discarded when you fetch the next column.

• Call the stream column in the same order as in the SELECT statement.

If your query fetches multiple columns, the database sends each row as a set of bytes representing the columns in the SELECT order. If one of the columns contains stream data, then the database sends the entire data stream before proceeding to the next column.

If you do not use the order as in the SELECT statement to access data, then you can lose the stream data. That is, if you bypass the stream data column and



access data in a column that follows it, then the stream data will be lost. For example, if you try to access the data for the NUMBER column *before* reading the data from the stream data column, then the JDBC driver first reads then discards the streaming data automatically. This can be very inefficient if the LONG column contains a large amount of data.

If you try to access the LONG column later in the program, then the data will not be available and the driver will return a "Stream Closed" error.

The later point is illustrated in the following example:

If you get the stream but do not use it *before* you get the NUMBER column, then the stream still closes automatically:

13.7.2 About Using Streams to Avoid Limits on setBytes and setString

Starting from Oracle Database 12c, the size limit of the data that is used with the setBytes and setString methods, have been increased significantly. Any Java byte array can be passed to setBytes, and any Java String can be passed to setString. The JDBC driver automatically switches to using setBinaryStream or setCharacterStream or to using setBytesForBlob or setStringForClob, depending on the size of the data, whether the statement is SQL or PL/SQL, and the driver used.

There are some limitation with earlier versions of Oracle Database and in the server-side internal driver.

Related Topics

Data Interface for LOBs

The data interface for LOBs includes a set of Java and OCI APIs that are extended to work with LOB data types.

13.7.3 Relation Between Streaming and Row Prefetching

If the JDBC driver encounters a column containing a data stream, then row fetch size is set back to one. Row fetch size is an Oracle performance enhancement that enables multiple rows of data to be retrieved with each trip to the database.



14 Working with Oracle Object Types

This chapter describes the Java Database Connectivity (JDBC) support for user-defined object types. It discusses functionality of the generic, weakly typed oracle.sql.STRUCT class, as well as how to map to custom Java classes that implement either the JDBC standard SQLData interface or the Oracle-specific OracleData interface.

Note:

Starting from Oracle Database 12c Release 1 (12.1), the oracle.sql.STRUCT class is deprecated and replaced with the oracle.jdbc.OracleStruct interface, which is a part of the oracle.jdbc package. Oracle strongly recommends you to use the methods available in the java.sql package, where possible, for standard compatibility and methods available in the oracle.jdbc package for Oracle specific extensions. Refer to MoS Note 1364193.1 for more information about the oracle.jdbc.OracleStruct interface.

The following topics are covered:

- About Mapping Oracle Objects
- About Using the Default STRUCT Class for Oracle Objects
- About Creating and Using Custom Object Classes for Oracle Objects
- Object-Type Inheritance
- About Describing an Object Type

Related Topics

About Using PL/SQL Types

14.1 About Mapping Oracle Objects

Oracle object types provide support for composite data structures in the database. For example, you can define a Person type that has the attributes name of CHAR type, phoneNumber of CHAR type, and employeeNumber of NUMBER type.

Oracle provides tight integration between its Oracle object features and its JDBC functionality. You can use a standard, generic JDBC type to map to Oracle objects, or you can customize the mapping by creating custom Java type definition classes.



Note:

In this book, Java classes that you create to map to Oracle objects will be referred to as **custom Java classes** or, more specifically, **custom object classes**. This is as opposed to **custom references classes**, which are Java classes that map to object references, and **custom collection classes**, which are Java classes that map to Oracle collections.

Custom object classes can implement either a standard JDBC interface or an Oracle extension interface to read and write data. JDBC materializes Oracle objects as instances of particular Java classes. Two main steps in using JDBC to access Oracle objects are:

- 1. Creating the Java classes for the Oracle objects
- 2. Populating these classes. You have the following options:
 - Let JDBC materialize the object as a STRUCT object.
 - Explicitly specify the mappings between Oracle objects and Java classes.

This includes customizing your Java classes for object data. The driver then must be able to populate instances of the custom object classes that you specify. This imposes a set of constraints on the Java classes. To satisfy these constraints, you can define your classes to implement either the JDBC standard java.sql.SQLData interface or the Oracle extension oracle.jdbc.OracleData interface.

Note:

When you use the SQLData interface, you must use a Java type map to specify your SQL-Java mapping, unless weakly typed java.sql.Struct objects will suffice.

14.2 About Using the Default STRUCT Class for Oracle Objects

This section covers the following topics:

- Overview of Using the Struct Class
- Retrieving STRUCT Objects and Attributes
- About Creating STRUCT Objects
- Binding STRUCT Objects into Statements
- STRUCT Automatic Attribute Buffering



14.2.1 Overview of Using the Struct Class

If you choose not to supply a custom Java class for your SQL-Java mapping for an Oracle object, then Oracle JDBC materializes the object as an object that implements the java.sql.Struct interface.

You would typically want to use STRUCT objects, instead of custom Java objects, in situations where you do not know the actual SQL type. For example, your Java application might be a tool to manipulate arbitrary object data within the database, as opposed to being an end-user application. You can select data from the database into STRUCT objects and create STRUCT objects for inserting data into the database. STRUCT objects completely preserve data, because they maintain the data in SQL format. Using STRUCT objects is more efficient and more precise in situations where you do not need the information in an application specific form.

14.2.2 Retrieving STRUCT Objects and Attributes

This section discusses how to retrieve and manipulate Oracle objects and their attributes, using either Oracle-specific features or JDBC 2.0 standard features.

Note:

The JDBC driver seamlessly handles embedded objects, that is, STRUCT objects that are attributes of STRUCT objects, in the same way that it typically handles objects. When the JDBC driver retrieves an attribute that is an object, it follows the same rules of conversion by using the type map, if it is available, or by using default mapping.

Retrieving an Oracle Object as a java.sql.Struct Object

Alternatively, in the preceding example, you can use standard JDBC functionality, such as getObject, to retrieve an Oracle object from the database as an instance of java.sql.Struct. The getObject method returns a java.lang.Object, so, you must cast the output of the method to Struct. For example:

```
ResultSet rs= stmt.executeQuery("SELECT * FROM struct_table");
java.sql.Struct jdbcStruct = (java.sql.Struct)rs.getObject(1);
```

Retrieving Attributes as oracle.sql Types

If you want to retrieve Oracle object attributes from a STRUCT or Struct instance as oracle.sql types, then use the getOracleAttributes method of the oracle.sql.STRUCT class, as follows:

oracle.sql.Datum[] attrs = oracleSTRUCT.getOracleAttributes();

or:

```
oracle.sql.Datum[] attrs = ((oracle.sql.STRUCT)jdbcStruct).getOracleAttributes();
```



Retrieving Attributes as Standard Java Types

If you want to retrieve Oracle object attributes as standard Java types from a STRUCT or Struct instance, use the standard getAttributes method:

Object[] attrs = jdbcStruct.getAttributes();

Note:

Oracle JDBC drivers cache array and structure descriptors. This provides enormous performance benefits. However, it means that if you change the underlying type definition of a structure type in the database, the cached descriptor for that structure type will become stale and your application will receive a SQLException exception.

14.2.3 About Creating STRUCT Objects

For information about creating STRUCT objects, refer to "Package oracle.sql".

Note:

If you have already fetched from the database a STRUCT of the appropriate SQL object type, then the easiest way to get a STRUCT descriptor is to call getDescriptor on one of the fetched STRUCT objects. Only one STRUCT descriptor is needed for any one SQL object type.

14.2.4 Binding STRUCT Objects into Statements

To bind an oracle.sql.STRUCT object to a prepared statement or callable statement, you can either use the standard setObject method (specifying the type code), or cast the statement object to an Oracle statement type and use the Oracle extension setOracleObject method. For example:

```
PreparedStatement ps= conn.prepareStatement("text_of_prepared_statement");
Struct mySTRUCT = conn.createStruct (...);
ps.setObject(1, mySTRUCT, Types.STRUCT);
```

or:

```
PreparedStatement ps= conn.prepareStatement("text_of_prepared_statement");
Struct mySTRUCT = conn.createStruct (...);
((OraclePreparedStatement)ps).setOracleObject(1, mySTRUCT);
```

14.2.5 STRUCT Automatic Attribute Buffering

Oracle JDBC driver furnishes public methods to enable and disable buffering of STRUCT attributes.



Note:

Starting from Oracle Database 12c Release 1 (12.1), the oracle.sql.STRUCT class is deprecated and replaced with the oracle.jdbc.OracleStruct interface, which is a part of the oracle.jdbc package. Oracle strongly recommends you to use the methods available in the java.sql package, where possible, for standard compatibility and methods available in the oracle.jdbc package for Oracle specific extensions. Refer to MoS Note 1364193.1 for more information about the oracle.jdbc.OracleStruct interface.

The following methods are included with the oracle.sql.STRUCT class:

- public void setAutoBuffering(boolean enable)
- public boolean getAutoBuffering()

The setAutoBuffering(boolean) method enables or disables auto-buffering. The getAutoBuffering method returns the current auto-buffering mode. By default, auto-buffering is disabled.

It is advisable to enable auto-buffering in a JDBC application when the STRUCT attributes are accessed more than once by the getAttributes and getArray methods, presuming the ARRAY data is able to fit into the Java Virtual Machine (JVM) memory without overflow.

Note:

Buffering the converted attributes may cause the JDBC application to consume a significant amount of memory.

When you enable auto-buffering, the oracle.sql.STRUCT object keeps a local copy of all the converted attributes. This data is retained so that subsequent access of this information does not require going through the data format conversion process.

Related Topics

ARRAY Automatic Element Buffering

14.3 About Creating and Using Custom Object Classes for Oracle Objects

This section covers the following topics:

- Overview of Creating and Using Custom Object Classes
- Relative Advantages of OracleData versus SQLData
- About Type Maps for SQLData Implementations
- About Creating Type Map and Defining Mappings for a SQLData Implementation
- About Reading and Writing Data with a SQLData Implementation



- About the OracleData Interface
- About Reading and Writing Data with an OracleData Implementation
- Additional Uses of OracleData

14.3.1 Overview of Creating and Using Custom Object Classes

If you want to create custom object classes for your Oracle objects, then you must define entries in the type map that specify the custom object classes that the drivers instantiate for the corresponding Oracle objects.

You must also provide a way to create and populate instances of the custom object class from the Oracle object and its attribute data. The driver must be able to read from a custom object class and write to it. In addition, the custom object class can provide getXXX and setXXX methods corresponding to the attributes of the Oracle object, although this is not necessary. To create and populate the custom classes and provide these read/write capabilities, you can choose between the following interfaces:

- The JDBC standard SQLData interface
- The OracleData and OracleDataFactory interfaces provided by Oracle

The custom object class you create must implement one of these interfaces. The OracleData interface can also be used to implement the custom reference class corresponding to the custom object class. However, if you are using the SQLData interface, then you can use only weak reference types in Java, such as java.sql.Ref or oracle.sql.REF. The SQLData interface is for mapping SQL objects only.

As an example, assume you have an Oracle object type, EMPLOYEE, in the database that consists of two attributes: Name, which is of the CHAR type and EmpNum, which is of the NUMBER type. You use the type map to specify that the EMPLOYEE object should map to a custom object class that you call JEmployee. You can implement either the SQLData or OracleData interface in the JEmployee class.

Related Topics

Object-Type Inheritance

14.3.2 Relative Advantages of OracleData versus SQLData

In deciding which of the two interface implementations to use, you need to consider the advantages of OracleData and SQLData.

The SQLData interface is for mapping SQL objects only. The OracleData interface is more flexible, enabling you to map SQL objects as well as any other SQL type for which you want to customize processing. You can create an OracleData implementation from any data type found in Oracle Database. This could be useful, for example, for serializing RAW data in Java.

Advantages of the OracleData Interface

The advantages of the OracleData interface are:

- It does not require an entry in the type map for the Oracle object.
- It has awareness of Oracle extensions.



- You can construct an OracleData from an oracle.sql.STRUCT. This is more efficient because it avoids unnecessary conversions to native Java types.
- You can obtain the corresponding JDBC object from OracleData, using the toJDBCObject method.

Advantages of SQLData

SQLData is a JDBC standard that makes your code portable.

14.3.3 About Type Maps for SQLData Implementations

If you use the SQLData interface in a custom object class, then you must create type map entries that specify the custom object class to use in mapping the Oracle object type to Java. You can either use the default type map of the connection object or a type map that you specify when you retrieve the data from the result set. The getObject method of the ResultSet interface has a signature that lets you specify a type map. You can use either of the following:

```
rs.getObject(int columnIndex);
```

rs.getObject(int columnIndex, Map map);

When using a SQLData implementation, if you do not include a type map entry, then the object maps to the oracle.jdbc.OracleStruct interface by default.OracleData implementations, by contrast, have their own mapping functionality so that a type map entry is not required. When using an OracleData implementation, use the Oracle getObject(int columnindex, OracleDataFactory factory) method.

The type map relates a Java class to the SQL type name of an Oracle object. This one-toone mapping is stored in a hash table as a keyword-value pair. When you read data from an Oracle object, the JDBC driver considers the type map to determine which Java class to use to materialize the data from the Oracle object type. When you write data to an Oracle object, the JDBC driver gets the SQL type name from the Java class by calling the getSQLTypeName method of the SQLData interface. The actual conversion between SQL and Java is performed by the driver.

The attributes of the Java class that corresponds to an Oracle object can use either Java native types or Oracle native types to store attributes.

Related Topics

About Creating and Using Custom Object Classes for Oracle Objects

14.3.4 About Creating Type Map and Defining Mappings for a SQLData Implementation

This section covers the following topics:

- Overview of Creating a Type Map and Defining Mappings
- Adding Entries to an Existing Type Map
- Creating a New Type Map
- About Materializing Object Types not Specified in the Type Map



14.3.4.1 Overview of Creating a Type Map and Defining Mappings

When using a SQLData implementation, the JDBC applications programmer is responsible for providing a type map, which must be an instance of a class that implements the standard java.util.Map interface.

You have the option of creating your own class to accomplish this, but the standard java.util.Hashtable class meets the requirement.

Hashtable and other classes used for type maps implement a put method that takes keyword-value pairs as input, where each key is a fully qualified SQL type name and the corresponding value is an instance of a specified Java class.

A type map is associated with a connection instance. The standard java.sql.Connection interface and the Oracle-specific oracle.jdbc.OracleConnection interface include a getTypeMap method. Both return a Map object.

14.3.4.2 Adding Entries to an Existing Type Map

When a connection instance is first established, the default type map is empty. You must populate it.

Perform the following general steps to add entries to an existing type map:

 Use the getTypeMap method of your OracleConnection object to return the type map object of the connection. The getTypeMap method returns a java.util.Map object. For example, presuming an OracleConnection instance oraconn:

java.util.Map myMap = oraconn.getTypeMap();

Note:

If the type map in the OracleConnection instance has not been initialized, then the first call to getTypeMap returns an empty map.

2. Use the put method of the type map to add map entries. The put method takes two arguments: a SQL type name string and an instance of a specified Java class that you want to map to.

myMap.put(sqlTypeName, classObject);

The *sqlTypeName* is a string that represents the fully qualified name of the SQL type in the database. The *classObject* is the Java class object to which you want to map the SQL type. Get the class object with the Class.forName method, as follows:

myMap.put(sqlTypeName, Class.forName(className));

For example, if you have a PERSON SQL data type defined in the CORPORATE database schema, then map it to a Person Java class defined as Person with this statement:



```
myMap.put("CORFORATE.PERSON", Class.forName("Person"));
oraconn.setTypeMap(newMap);
```

The map has an entry that maps the PERSON SQL data type in the CORPORATE database to the Person Java class.



SQL type names in the type map must be all uppercase, because that is how Oracle Database stores SQL names.

14.3.4.3 Creating a New Type Map

Perform the following general steps to create a new type map. This example uses an instance of java.util.Hashtable, which extends java.util.Dictionary and implements java.util.Map.

1. Create a new type map object.

Hashtable newMap = new Hashtable();

2. Use the put method of the type map object to add entries to the map. For example, if you have an EMPLOYEE SQL type defined in the CORPORATE database, then you can map it to an Employee class object defined by Employee.java, as follows:

newMap.put("CORPORATE.EMPLOYEE", class.forName("Employee"));

3. When you finish adding entries to the map, you must use the setTypeMap method of the OracleConnection object to overwrite the existing type map of the connection. For example:

oraconn.setTypeMap(newMap);

In this example, the setTypeMap method overwrites the original map of the oraconn connection object with newMap.

Note:

The default type map of a connection instance is used when mapping is required but no map name is specified, such as for a result set getObject call that does not specify the map as input.

14.3.4.4 About Materializing Object Types not Specified in the Type Map

If you do not provide a type map with an appropriate entry when using a getObject call, then the JDBC driver will materialize an Oracle object as an instance of the oracle.jdbc.OracleStruct interface. If the Oracle object type contains embedded objects and they are not present in the type map, then the driver will materialize the embedded objects as instances of oracle.jdbc.OracleStruct as well. If the embedded objects are present in the type map, then a call to the getAttributes method will return embedded objects as instances of the specified Java classes from the type map.



14.3.5 About Reading and Writing Data with a SQLData Implementation

This section describes how to read data from an Oracle object or write data to an Oracle object if your corresponding Java class implements SQLData.

Reading SQLData Objects from a Result Set

The following text summarizes the steps to read data from an Oracle object into your Java application when you choose the SQLData implementation for your custom object class.

These steps assume you have already defined the Oracle object type, created the corresponding custom object class, updated the type map to define the mapping between the Oracle object and the Java class, and defined a statement object stmt.

1. Query the database to read the Oracle object into a JDBC result set.

```
ResultSet rs = stmt.executeQuery("SELECT emp_col FROM personnel");
```

The PERSONNEL table contains one column, EMP_COL, of SQL type EMP_OBJECT. This SQL type is defined in the type map to map to the Java class Employee.

 Use the getObject method of Oracle result set to populate an instance of your custom object class with data from one row of the result set. The getObject method returns the user-defined SQLData object because the type map contains an entry for Employee.

```
if (rs.next())
    Employee emp = (Employee)rs.getObject(1);
```

Note that if the type map did not have an entry for the object, then the getObject method will return an oracle.jdbc.OracleStruct object. Cast the output to type OracleStruct because the getObject method signature returns the generic java.lang.Object type.

```
if (rs.next())
    OracleStruct empstruct = (OracleStruct)rs.getObject(1);
```

The getObject method calls readSQL, which, in turn, calls readXXX from the SQLData interface.

Note:

If you want to avoid using the defined type map, then use the getSTRUCT method. This method always returns a STRUCT object, even if there is a mapping entry in the type map.

3. If you have get methods in your custom object class, then use them to read data from your object attributes. For example, if EMPLOYEE has the attributes EmpName of type CHAR and EmpNum of type NUMBER, then provide a getEmpName method that returns a Java String and a getEmpNum method that returns an int value. Then call them in your Java application, as follows:



```
String empname = emp.getEmpName();
int empnumber = emp.getEmpNum();
```

Retrieving SQLData Objects from a Callable Statement OUT Parameter

Consider you have a CallableStatement instance, cs, that calls a PL/SQL function GETEMPLOYEE. The program passes an employee number to the function. The function returns the corresponding Employee object. To retrieve this object you do the following:

1. Prepare a CallableStatement to call the GETEMPLOYEE function, as follows:

```
CallableStatement ocs = conn.prepareCall("{ ? = call GETEMPLOYEE(?) }");
```

2. Declare the empnumber as the input parameter to GETEMPLOYEE. Register the SQLData object as the OUT parameter, with the type code OracleTypes.STRUCT. Then, run the statement. This can be done as follows:

```
cs.setInt(2, empnumber);
cs.registerOutParameter(1, OracleTypes.STRUCT, "EMP_OBJECT");
cs.execute();
```

3. Use the getObject method to retrieve the employee object.

```
Employee emp = (Employee)cs.getObject(1);
```

If there is no type map entry, then the getObject method will return a java.sql.Struct object.

```
Struct emp = cs.getObject(1);
```

Passing SQLData Objects to a Callable Statement as an IN Parameter

Suppose you have a PL/SQL function addEmployee (?) that takes an Employee object as an IN parameter and adds it to the PERSONNEL table. In this example, emp is a valid Employee object.

1. Prepare an CallableStatement to call the addEmployee (?) function.

```
CallableStatement cs =
   conn.prepareCall("{ call addEmployee(?) }");
```

2. Use setObject to pass the emp object as an IN parameter to the callable statement. Then, call the statement.

```
cs.setObject(1, emp);
cs.execute();
```

Writing Data to an Oracle Object Using a SQLData Implementation

The following text describes the steps in writing data to an Oracle object from your Java application when you choose the SQLData implementation for your custom object class.

This description assumes you have already defined the Oracle object type, created the corresponding Java class, and updated the type map to define the mapping between the Oracle object and the Java class.

1. If you have set methods in your custom object class, then use them to write data from Java variables in your application to attributes of your Java data type object.

```
emp.setEmpName(empname);
emp.setEmpNum(empnumber);
```



2. Prepare a statement that updates an Oracle object in a row of a database table, as appropriate, using the data provided in your Java data type object.

3. Use the setObject method of the prepared statement to bind your Java data type object to the prepared statement.

pstmt.setObject(1, emp);

4. Run the statement, which updates the database.

pstmt.executeUpdate();

14.3.6 About the OracleData Interface

You can create a custom object class that implements the <code>oracle.jdbc.OracleData</code> and the <code>oracle.jdbc.OracleDataFactory</code> interfaces to make an Oracle object and its attribute data available to Java applications. The <code>OracleData</code> and <code>OracleDataFactory</code> interfaces are Oracle-specific and are not a part of the JDBC standard.

Note:

Starting from Oracle Database 12c Release 1 (12.1), the OracleData and the OracleDataFactory interfaces replace the ORAData and the ORADataFactory interfaces.

Understanding the OracleData Interface Features

The OracleData interface has the following advantages:

- It supports Oracle extensions to the standard JDBC types.
- It does not require a type map to specify the names of the Java custom classes you want to create.
- It provides better performance. OracleData works directly with Datum types, the internal format the driver uses to hold Oracle objects.

The OracleData and the OracleDataFactory interfaces perform the following:

- The toJDBCObject method of the OracleData class transforms the data into an oracle.jdbc.* representation.
- OracleDataFactory specifies a create method equivalent to a constructor for the custom object class. It creates and returns an OracleData instance. The JDBC driver uses the create method to return an instance of the custom object class to your Java application. It takes as input a java.lang.Object object and an integer indicating the corresponding SQL type code as specified in the OracleTypes class.

OracleData and OracleDataFactory have the following definitions:

```
package oracle.jdbc;
import java.sql.Connection;
import java.sql.SQLException;
public interface OracleData
{
```



```
public Object toJDBCObject(Connection conn) throws SQLException;
}
package oracle.jdbc;
import java.sql.SQLException;
public interface OracleDataFactory
{
    public OracleData create(Object jdbcValue, int sqlType) throws SQLException;
}
```

Where *conn* represents the Connection object, *jdbcValue* represents an object of type java.lang.object that is to be used to initialize the Object being created, and *sqlType* represents the SQL type of the specified Datum object.

Retrieving and Inserting Object Data

The JDBC drivers provide the following methods to retrieve and insert object data as instances of OracleData.

You can retrieve the object data in one of the following ways:

• Use the following getObject method of the Oracle-specific OracleResultSet interface:

```
ors.getObject(int col_index, OracleDataFactory factory
);
```

This method takes as input the column index of the data in your result set and an OracleDataFactory instance. For example, you can implement a getOracleDataFactory method in your custom object class to produce the OracleDataFactory instance to input to the getObject method. The type map is not required when using Java classes that implement OracleData.

• Use the standard getObject(*index*, *map*) method specified by the ResultSet interface to retrieve data as instances of OracleData. In this case, you must have an entry in the type map that identifies the factory class to be used for the given object type and its corresponding SQL type name.

You can insert object data in one of the following ways:

• Use the following setObject method of the Oracle-specific OraclePreparedStatement class:

setObject(int bind_index, Object custom_object);

This method takes as input the parameter index of the bind variable and an instance of OracleData as the name of the object containing the variable.

• Use the standard setObject method specified by the PreparedStatement interface. You can also use this method, in its different forms, to insert OracleData instances without requiring a type map.

The following sections describe the getObject and setObject methods.

To continue the example of an Oracle object EMPLOYEE, you might have something like the following in your Java application:

OracleData obj = ors.getObject(1, Employee.getOracleDataFactory());



In this example, ors is an instance of the OracleResultSet interface, getObject is a method in the OracleResultSet interface used to retrieve an OracleData object, and the EMPLOYEE is in column 1 of the result set. The static

Employee.getOracleDataFactory method will return an OracleDataFactory to the JDBC driver. The JDBC driver will call create () from this object, returning to your Java application an instance of the Employee class populated with data from the result set.

Note:

- OracleData and OracleDataFactory are defined as separate interfaces so that different Java classes can implement them if you wish.
- To use the OracleData interface, your custom object classes must import oracle.jdbc.*.

14.3.7 About Reading and Writing Data with an OracleData Implementation

This section describes how to read data from an Oracle object or write data to an Oracle object if your corresponding Java class implements OracleData.

Reading Data from an Oracle Object Using an OracleData Implementation

The following text summarizes the steps in reading data from an Oracle object into your Java application. These steps apply whether you implement <code>OracleData</code> manually or use Oracle JVM Web Service Call-Out utility to produce your custom object classes.

These steps assume you have already defined the Oracle object type, created the corresponding custom object class or had Oracle JVM Web Service Call-Out utility create it for you, and defined a statement object stmt.

1. Query the database to read the Oracle object into a result set, casting it to an Oracle result set.

Where PERSONNEL is a one-column table. The column name is Emp_col of type Employee object.

2. Use the getObject method of Oracle result set to populate an instance of your custom object class with data from one row of the result set. The getObject method returns a java.lang.Object object, which you can cast to your specific custom object class.

```
if (ors.next())
    Employee emp = (Employee)ors.getObject(1,
Employee.getOracleDataFactory());
Or:
```

```
if (ors.next())
    Object obj = ors.getObject(1, Employee.getOracleDataFactory());
```



This example assumes that Employee is the name of your custom object class and ors is the name of your OracleResultSet instance.

For example, if the SQL type name for your object is EMPLOYEE, then the corresponding Java class is Employee, which will implement OracleData. The corresponding Factory class is EmployeeFactory, which will implement OracleDataFactory.

Use this statement to declare the EmployeeFactory entry for your type map:

map.put ("EMPLOYEE", Class.forName ("EmployeeFactory"));

Then use the form of getObject where you specify the map object:

Employee emp = (Employee) rs.getObject (1, map);

If the default type map of the connection already has an entry that identifies the factory class to be used for the given object type and its corresponding SQL type name, then you can use this form of getObject:

Employee emp = (Employee) rs.getObject (1);

3. If you have get methods in your custom object class, then use them to read data from your object attributes into Java variables in your application. For example, if EMPLOYEE has EmpName of type CHAR and EmpNum of type NUMBER, provide a getEmpName method that returns a Java String and a getEmpNum method that returns an integer. Then call them in your Java application as follows:

```
String empname = emp.getEmpName();
int empnumber = emp.getEmpNum();
```

Writing Data to an Oracle Object Using an OracleData Implementation

The following text summarizes the steps in writing data to an Oracle object from your Java application. These steps apply whether you implement OracleData manually or use Oracle JVM Web Service Call-Out utility to produce your custom object classes.

These steps assume you have already defined the Oracle object type and created the corresponding custom object class.

Note:

The type map is not used when you are performing database INSERT and UPDATE operations.

1. If you have set methods in your custom object class, then use them to write data from Java variables in your application to attributes of your Java data type object.

```
emp.setEmpName(empname);
emp.setEmpNum(empnumber);
```

2. Write an Oracle prepared statement that updates an Oracle object in a row of a database table, as appropriate, using the data provided in your Java data type object.

```
OraclePreparedStatement opstmt = conn.prepareStatement
  ("UPDATE PERSONNEL SET Employee = ? WHERE Employee.EmpNum = 28959);
```

This assumes conn is your Connection object.



3. Use the setObject method of the OraclePreparedStatement interface to bind your Java data type object to the prepared statement.

opstmt.setObject(1,emp);

The setObject method calls the toJDBCObject method of the custom object class instance to retrieve an oracle.jdbc.OracleStruct object that can be written to the database.

Note:

You can use your Java data type objects as either ${\tt IN}$ or ${\tt OUT}$ bind variables.

14.3.8 Additional Uses of OracleData

The <code>OracleData</code> interface offers far more flexibility than the <code>SQLData</code> interface. The <code>SQLData</code> interface is designed to let you customize the mapping of only Oracle object types to Java types of your choice. Implementing the <code>SQLData</code> interface lets the JDBC driver populate fields of a custom Java class instance from the original SQL object data, and the reverse, after performing the appropriate conversions between Java and SQL types.

The OracleData interface goes beyond supporting the customization of Oracle object types to Java types. It lets you provide a mapping between Java object types and *any* SQL type supported by the oracle.sql package.

You may find it useful to provide custom Java classes to wrap <code>oracle.sql.*</code> types and then implement customized conversions or functionality as well. The following are some possible scenarios:

- Performing encryption and decryption or validation of data
- Performing logging of values that have been read or are being written
- Parsing character columns, such as character fields containing URL information, into smaller components
- Mapping character strings into numeric constants
- Making data into more desirable Java formats, such as mapping a DATE field to java.util.Date format
- Customizing data representation, for example, data in a table column is in feet but you want it represented in meters after it is selected
- Serializing and deserializing Java objects

For example, use <code>OracleData</code> to store instances of Java objects that do not correspond to a particular SQL object type in the database in columns of SQL type <code>RAW</code>. The <code>create</code> method in <code>OracleDataFactory</code> would have to implement a conversion from an object of type <code>oracle.sql.RAW</code> to the desired Java object. The <code>toJDBCObject</code> method in <code>OracleData</code> would have to implement a conversion from the Java object to an <code>oracle.sql.RAW</code> object. You can also achieve this using Java serialization.



Upon retrieval, the JDBC driver transparently retrieves the raw bytes of data in the form of an oracle.sql.RAW and calls the create method of OracleDataFactory to convert the oracle.sql.RAW object to the desired Java class.

When you insert the Java object into the database, you can simply bind it to a column of type RAW to store it. The driver transparently calls the <code>OracleData.toJDBCObject</code> method to convert the Java object to an <code>oracle.sql.RAW</code> object. This object is then stored in a column of type RAW in the database.

Support for the <code>OracleData</code> interfaces is also highly efficient because the conversions are designed to work using <code>oracle.sql.*</code> formats, which happen to be the internal formats used by the JDBC drivers. Moreover, the type map, which is necessary for the <code>SQLData</code> interface, is not required when using Java classes that implement <code>OracleData</code>.

Related Topics

• About the OracleData Interface

14.4 Object-Type Inheritance

Object-type inheritance allows a new object type to be created by extending another object type. The new object type is then a subtype of the object type from which it extends. The subtype automatically inherits all the attributes and methods defined in the supertype. The subtype can add attributes and methods and overload or override methods inherited from the supertype.

Object-type inheritance introduces **substitutability**. Substitutability is the ability of a slot declared to hold a value of type T in addition to any subtype of type T. Oracle JDBC drivers handle substitutability transparently.

A database object is returned with its most specific type without losing information. For example, if the STUDENT_T object is stored in a PERSON_T slot, Oracle JDBC driver returns a Java object that represents the STUDENT T object.

This section covers the following topics:

- About Creating Subtypes
- About Implementing Customized Classes for Subtypes
- About Retrieving Subtype Objects
- Creating Subtype Objects
- Sending Subtype Objects
- Accessing Subtype Data Fields
- Inheritance Metadata Methods

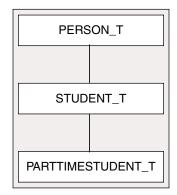
14.4.1 About Creating Subtypes

Create custom object classes if you want to have Java classes that explicitly correspond to the Oracle object types. If you have a hierarchy of object types, you may want a corresponding hierarchy of Java classes.

The most common way to create a database subtype in JDBC is to run a SQL CREATE TYPE command using the execute method of the java.sql.Statement interface. For example, you want to create a type inheritance hierarchy as depicted in the following figure:



Figure 14-1 Type Inheritance Hierarchy



The JDBC code for this can be as follows:

```
Statement s = conn.createStatement();
s.execute ("CREATE TYPE Person_T (SSN NUMBER, name VARCHAR2(30),
   address VARCHAR2(255))");
s.execute ("CREATE TYPE Student_T UNDER Person_t (deptid NUMBER,
   major VARCHAR2(100))");
s.execute ("CREATE TYPE PartTimeStudent t UNDER Student t (numHours NUMBER)");
```

In the following code, the foo member procedure in type ST is overloaded and the member procedure print overwrites the copy it inherits from type T.

```
CREATE TYPE T AS OBJECT (...,
MEMBER PROCEDURE foo(x NUMBER),
MEMBER PROCEDURE Print(),
...
NOT FINAL;
CREATE TYPE ST UNDER T (...,
MEMBER PROCEDURE foo(x DATE), <-- overload "foo"
OVERRIDING MEMBER PROCEDURE Print(), <-- override "print"
STATIC FUNCTION bar(...) ...
...
);
```

Once the subtypes have been created, they can be used as both columns of a base table as well as attributes of an object type.



14.4.2 About Implementing Customized Classes for Subtypes

In most cases, a customized Java class represents a database object type. When you create a customized Java class for a subtype, the Java class can either mirror the database object type hierarchy or not.



You can use either the <code>OracleData</code> or <code>SQLData</code> solution in creating classes to map to the hierarchy of object types.

This section covers the following topics:

- About Using OracleData for Type Inheritance Hierarchy
- About UsingSQLData for Type Inheritance Hierarchy

14.4.2.1 About Using OracleData for Type Inheritance Hierarchy

Oracle recommends customized mappings, where Java classes implement the oracle.sql.OracleData interface. OracleData mapping requires the JDBC application to implement the OracleData and OracleDataFactory interfaces. The class implementing the OracleDataFactory interface contains a factory method that produces objects. Each object represents a database object.

The hierarchy of the class implementing the <code>OracleData</code> interface can mirror the database object type hierarchy. For example, the Java classes mapping to <code>PERSON_T</code> and <code>STUDENT_T</code> are as follows:

Person.java using OracleData

Code for the Person.java class which implements the OracleData and OracleDataFactory interfaces:

```
public static OracleDataFactory getOracleDataFactory()
  {
   return personFactory;
  }
 public Person () {}
 public Person(NUMBER ssn, CHAR name, CHAR address)
  {
    this.ssn = ssn;
   this.name = name;
    this.address = address;
  }
 public Object toJDBCObject(OracleConnection c) throws SQLException
   Object [] attributes = { ssn, name, address };
 Struct struct = c.createStruct("HR.PERSON T", attributes);
   return struct;
  }
  public OracleData create(Object jdbcValue, int sqlType) throws SQLException
   if (d == null) return null;
   Object [] attributes = ((STRUCT) d).getOracleAttributes();
   return new Person((NUMBER) attributes[0],
                     (CHAR) attributes[1],
                      (CHAR) attributes[2]);
  }
}
```

Student.java extending Person.java

Code for the Student.java class, which extends the Person.java class:



```
class Student extends Person
{
  static final Student studentFactory = new Student ();
 public NUMBER deptid;
 public CHAR major;
 public static OracleDataFactory getOracleDataFactory()
    return studentFactory;
  }
 public Student () {}
  public Student (NUMBER ssn, CHAR name, CHAR address,
                 NUMBER deptid, CHAR major)
  {
   super (ssn, name, address);
   this.deptid = deptid;
   this.major = major;
  }
 public Object toJDBCObject (OracleConnection c) throws SQLException
  {
   Object [] attributes = { ssn, name, address, deptid, major };
   Struct struct = c.createStruct("HR.STUDENT T", attributes);
   return struct;
  }
  public OracleData create(Object jdbcValue, int sqlType) throws SQLException
    if (d == null) return null;
   Object [] attributes = ((STRUCT) d).getOracleAttributes();
    return new Student((NUMBER) attributes[0],
                       (CHAR) attributes[1],
                       (CHAR) attributes[2],
                       (NUMBER) attributes[3],
                       (CHAR) attributes[4]);
 }
}
```

Customized classes that implement the <code>OracleData</code> interface do not have to mirror the database object type hierarchy. For example, you could have declared the <code>Student</code> class without a superclass. In this case, <code>Student</code> would contain fields to hold the inherited attributes from <code>PERSON T</code> as well as the attributes declared by <code>STUDENT T</code>.

OracleDataFactory Implementation

The JDBC application uses the factory class in querying the database to return instances of Person or its subclasses, as in the following example:

```
ResultSet rset = stmt.executeQuery ("select person from tab1");
while (rset.next())
{
    rset.getOracleData(1,Person.getOracleDataFactory());
    ...
}
```



A class implementing the OracleDataFactory interface should be able to produce instances of the associated custom object type, as well as instances of any subtype, or at least all the types you expect to support.

In the following example, the PersonFactory.getOracleDataFactory method returns a factory that can handle PERSON_T, STUDENT_T, and PARTTIMESTUDENT_T objects, by returning person, student, or parttimestudent Java instances.

```
class PersonFactory implements OracleDataFactory
{
  static final PersonFactory factory = new PersonFactory ();
  public static OracleDataFactory getOracleDataFactory()
   return factory;
  public OracleData create(Object jdbcValue, int sqlType) throws SQLException
  {
   STRUCT s = (STRUCT) jdbcValue;
   if (s.getSQLTypeName ().equals ("HR.PERSON T"))
     return Person.getOracleDataFactory ().create (jdbcValue, sqlType);
   else if (s.getSQLTypeName ().equals ("HR.STUDENT T"))
     return Student.getOracleDataFactory ().create(jdbcValue, sqlType);
   else if (s.getSQLTypeName ().equals ("HR.PARTTIMESTUDENT T"))
     return ParttimeStudent.getOracleDataFactory ().create(jdbcValue, sqlType);
   else
     return null;
 }
}
```

The following example assumes a table tabl1, such as the following:

```
CREATE TABLE tabl1 (idx NUMBER, person PERSON_T);
INSERT INTO tabl1 VALUES (1, PERSON_T (1000, 'HR', '100 Oracle Parkway'));
INSERT INTO tabl1 VALUES (2, STUDENT_T (1001, 'Peter', '200 Oracle Parkway', 101,
'CS'));
INSERT INTO tabl1 VALUES (3, PARTTIMESTUDENT_T (1002, 'David', '300 Oracle Parkway',
102, 'EE'));
```

14.4.2.2 About UsingSQLData for Type Inheritance Hierarchy

The customized classes that implement the java.sql.SQLData interface can mirror the database object type hierarchy. The readSQL and writeSQL methods of a subclass typically call the corresponding superclass methods to read or write the superclass attributes before reading or writing the subclass attributes. For example, the Java classes mapping to PERSON_T and STUDENT_T are as follows:

Person.java using SQLData

Code for the Person.java class, which implements the SQLData interface:

```
import java.sql.*;
public class Person implements SQLData
{
    private String sql_type;
    public int ssn;
    public String name;
    public String address;
```



```
public Person () {}
public String getSQLTypeName() throws SQLException { return sql_type; }
public void readSQL(SQLInput stream, String typeName) throws SQLException
{
    sql_type = typeName;
    ssn = stream.readInt();
    name = stream.readString();
    address = stream.readString();
}
public void writeSQL(SQLOutput stream) throws SQLException
{
    stream.writeInt (ssn);
    stream.writeString (name);
    stream.writeString (address);
}
```

Student.java extending Student.java

Code for the Student.java class, which extends the Person.java class:

```
import java.sql.*;
public class Student extends Person
{
  private String sql type;
 public int deptid;
 public String major;
  public Student () { super(); }
  public String getSQLTypeName() throws SQLException { return sql type; }
  public void readSQL(SQLInput stream, String typeName) throws SQLException
    super.readSQL (stream, typeName);
                                       // read supertype attributes
    sql type = typeName;
    deptid = stream.readInt();
    major = stream.readString();
  }
  public void writeSQL(SQLOutput stream) throws SQLException
  {
    super.writeSQL (stream);
                                    // write supertype
                                         // attributes
    stream.writeInt (deptid);
    stream.writeString (major);
  }
}
```

Although not required, it is recommended that the customized classes, which implement the SQLData interface, mirror the database object type hierarchy. For example, you could have declared the Student class without a superclass. In this case, Student would contain fields to hold the inherited attributes from PERSON_T as well as the attributes declared by STUDENT_T.



Student.java using SQLData

Code for the Student.java class, which does not extend the Person.java class, but implements the SQLData interface directly:

```
import java.sql.*;
public class Student implements SQLData
  private String sql type;
  public int ssn;
  public String name;
  public String address;
  public int deptid;
  public String major;
  public Student () {}
  public String getSQLTypeName() throws SQLException { return sql type; }
  public void readSQL(SQLInput stream, String typeName) throws SQLException
  {
    sql type = typeName;
    ssn = stream.readInt();
    name = stream.readString();
    address = stream.readString();
    deptid = stream.readInt();
   major = stream.readString();
  1
  public void writeSQL(SQLOutput stream) throws SQLException
  {
    stream.writeInt (ssn);
    stream.writeString (name);
    stream.writeString (address);
    stream.writeInt (deptid);
    stream.writeString (major);
}
```

14.4.3 About Retrieving Subtype Objects

In a typical JDBC application, a subtype object is returned as one of the following:

- A query result
- A PL/SQL OUT parameter
- A type attribute

You can use either the default mapping or the SQLData mapping or the OracleData mapping to retrieve a subtype.

Using Default Mapping

By default, a database object is returned as an instance of the oracle.jdbc.OracleStruct interface. This instance may represent an object of either the declared type or subtype of the declared type. If the OracleStruct interface represents a subtype object in the database, then it contains the attributes of its supertype as well as those defined in the subtype.



Oracle JDBC driver returns database objects in their most specific type. The JDBC application can use the getSQLTypeName method of the OracleStruct interface to determine the SQL type of the STRUCT object. The following code shows this:

```
// tab1.person column can store PERSON_T, STUDENT_T and PARTIMESTUDENT_T objects
ResultSet rset = stmt.executeQuery ("select person from tab1");
while (rset.next())
{
    oracle.sql.STRUCT s = (oracle.sql.STRUCT) rset.getObject(1);
    if (s != null)
        System.out.println (s.getSQLTypeName()); // print out the type name which
        // may be HR.PERSON_T, HR.STUDENT_T or HR.PARTTIMESTUDENT_T
}
```

Using SQLData Mapping

With SQLData mapping, the JDBC driver returns the database object as an instance of the class implementing the SQLData interface.

To use SQLData mapping in retrieving database objects, do the following:

- Implement the container classes that implement the SQLData interface for the desired object types.
- 2. Populate the connection type map with entries that specify what custom Java type corresponds to each Oracle object type.
- 3. Use the getObject method to access the SQL object values.

The JDBC driver checks the type map for an entry match. If one exists, then the driver returns the database object as an instance of the class implementing the SQLData interface.

The following code shows the whole SQLData customized mapping process:

```
// The JDBC application developer implements Person.java for PERSON T,
// Student.java for STUDENT T
// and ParttimeStudent.java for PARTTIMESTUDEN_T.
Connection conn = ...; // make a JDBC connection
// obtains the connection typemap
java.util.Map map = conn.getTypeMap ();
// populate the type map
map.put ("HR.PERSON T", Class.forName ("Person"));
map.put ("HR.STUDENT T", Class.forName ("Student"));
map.put ("HR.PARTTIMESTUDENT T", Class.forName ("ParttimeStudent"));
// tabl.person column can store PERSON T, STUDENT T and PARTTIMESTUDENT T
objects
ResultSet rset = stmt.executeQuery ("select person from tab1");
while (rset.next())
{
  // "s" is instance of Person, Student or ParttimeStudent
  Object s = rset.getObject(1);
  if (s != null)
  {
    if (s instanceof Person)
      System.out.println ("This is a Person");
    else if (s instanceof Student)
```



```
System.out.println ("This is a Student");
else if (s instanceof ParttimeStudent)
System.out.pritnln ("This is a PartimeStudent");
else
System.out.println ("Unknown type");
}
}
```

The JDBC drivers check the connection type map for each call to the following:

- getObject method of the java.sql.ResultSet and java.sql.CallableStatement interfaces
- getAttribute method of the java.sql.Struct interface
- getArray method of the java.sql.Array interface
- getValue method of the oracle.sql.REF interface

Using OracleData Mapping

With OracleData mapping, the JDBC driver returns the database object as an instance of the class implementing the OracleData interface.

Oracle JDBC driver needs to be informed of what Java class is mapped to the Oracle object type. The following are the two ways to inform Oracle JDBC drivers:

- The JDBC application uses the getObject(int idx, OracleDataFactory f) method to access database objects. The second parameter of the getObject method specifies an instance of the factory class that produces the customized class. The getObject method is available in the OracleResultSet and OracleCallableStatement interfaces.
- The JDBC application populates the connection type map with entries that specify what custom Java type corresponds to each Oracle object type. The getObject method is used to access the Oracle object values.

The second approach involves the use of the standard getObject method. The following code example demonstrates the first approach:

```
// tabl.person column can store both PERSON T and STUDENT T objects
ResultSet rset = stmt.executeQuery ("select person from tab1");
while (rset.next())
  Object s = rset.getObject(1, PersonFactory.getOracleDataFactory());
  if (s != null)
  {
    if (s instanceof Person)
      System.out.println ("This is a Person");
    else if (s instanceof Student)
      System.out.println ("This is a Student");
    else if (s instanceof ParttimeStudent)
      System.out.pritnln ("This is a PartimeStudent");
    else
      System.out.println ("Unknown type");
  }
}
```



14.4.4 Creating Subtype Objects

There are cases where JDBC applications create database subtype objects with JDBC drivers. These objects are sent either to the database as bind variables or are used to exchange information within the JDBC application.

With customized mapping, the JDBC application creates either SQLData-based or OracleData-based objects, depending on the approach you choose, to represent database subtype objects. With default mapping, the JDBC application creates STRUCT objects to represent database subtype objects. All the data fields inherited from the supertype as well as all the fields defined in the subtype must have values. The following code demonstrates this:

s is initialized with data fields inherited from <code>PERSON_T</code> and <code>STUDENT_T</code>, and data fields defined in <code>PARTTIMESTUDENT T</code>.

14.4.5 Sending Subtype Objects

In a typical JDBC application, a Java object that represents a database object is sent to the databases as one of the following:

- A data manipulation language (DML) bind variable
- A PL/SQL IN parameter
- An object type attribute value

The Java object can be an instance of the STRUCT class or an instance of the class implementing either the SQLData or OracleData interface. Oracle JDBC driver will convert the Java object into the linearized format acceptable to the database SQL engine. Binding a subtype object is the same as binding a standard object.

14.4.6 Accessing Subtype Data Fields

While the logic to access subtype data fields is part of the customized class, this logic for default mapping is defined in the JDBC application itself. The database objects are returned as instances of the oracle.jdbc.OracleStruct class. The JDBC application needs to call one of the following access methods in the STRUCT class to access the data fields:

- Object[] getAttribute()
- oracle.sql.Datum[] getOracleAttribute()



Subtype Data Fields from the getAttribute Method

The getAttribute method of the java.sql.Struct interface is used in JDBC 2.0 to access object data fields. This method returns a java.lang.Object array, where each array element represents an object attribute. You can determine the individual element type by referencing the corresponding attribute type in the JDBC conversion matrix. For example, a SQL NUMBER attribute is converted to a java.math.BigDecimal object. The getAttribute method returns all the data fields defined in the supertype of the object type as well as data fields defined in the subtype. The supertype data fields are listed first followed by the subtype data fields.

Subtype Data Fields from the getOracleAttribute Method

The getOracleAttribute method is an Oracle extension method and is more efficient than the getAttribute method. The getOracleAttribute method returns an oracle.sql.Datum array to hold the data fields. Each element in the oracle.sql.Datum array represents an attribute. You can determine the individual element type by referencing the corresponding attribute type in the Oracle conversion matrix. For example, a SQL NUMBER attribute is converted to an oracle.sql.NUMBER object. The getOracleAttribute method returns all the attributes defined in the supertype of the object type, as well as attributes defined in the subtype. The supertype data fields are listed first followed by the subtype data fields.

The following code shows the use of the getAttribute method:

```
// tabl.person column can store PERSON T, STUDENT T and PARTIMESTUDENT T objects
ResultSet rset = stmt.executeQuery ("select person from tab1");
while (rset.next())
{
  oracle.sql.STRUCT s = (oracle.sql.STRUCT) rset.getObject(1);
 if (s != null)
    String sqlname = s.getSQLTypeName();
   Object[] attrs = s.getAttribute();
    if (sqlname.equals ("HR.PERSON")
    {
     System.out.println ("ssn="+((BigDecimal)attrs[0]).intValue());
     System.out.println ("name="+((String)attrs[1]));
     System.out.println ("address="+((String)attrs[2]));
    else if (sqlname.equals ("HR.STUDENT"))
    {
     System.out.println ("ssn="+((BigDecimal)attrs[0]).intValue());
     System.out.println ("name="+((String)attrs[1]));
     System.out.println ("address="+((String)attrs[2]));
     System.out.println ("deptid="+((BigDecimal)attrs[3]).intValue());
     System.out.println ("major="+((String)attrs[4]));
    else if (sqlname.equals ("HR.PARTTIMESTUDENT"))
     System.out.println ("ssn="+((BigDecimal)attrs[0]).intValue());
     System.out.println ("name="+((String)attrs[1]));
     System.out.println ("address="+((String)attrs[2]));
     System.out.println ("deptid="+((BigDecimal)attrs[3]).intValue());
     System.out.println ("major="+((String)attrs[4]));
      System.out.println ("numHours="+((BigDecimal)attrs[5]).intValue());
    }
    else
```



```
throw new Exception ("Invalid type name: "+sqlname);
}
rset.close ();
stmt.close ();
conn.close ();
```

14.4.7 Inheritance Metadata Methods

Oracle JDBC drivers provide a set of metadata methods to access inheritance properties. The inheritance metadata methods are defined in the oracle.sql.StructDescriptor and oracle.jdbc.StructMetaData classes.

The StructMetaData class provides inheritance metadata methods for subtype attributes. The getMetaData method of the StructDescriptor class returns an instance of StructMetaData of the type. The StructMetaData class contains the following inheritance metadata methods:

14.5 About Describing an Object Type

Oracle JDBC includes functionality to retrieve information about a structured object type regarding its attribute names and types. This is similar conceptually to retrieving information from a result set about its column names and types, and in fact uses an almost identical method.

This section covers the following topics:

- Functionality for Getting Object Metadata
- Retrieving Object Metadata

14.5.1 Functionality for Getting Object Metadata

The oracle.sql.StructDescriptor class includes functionality to retrieve metadata about a structured object type. The StructDescriptor class has a getMetaData method with the same functionality as the standard getMetaData method available in result set objects. It returns a set of attribute information, such as attribute names and types. Call this method on a StructDescriptor object to get metadata about the Oracle object type that the StructDescriptor object describes.

The signature of the StructDescriptor class getMetaData method is the same as the signature specified for getMetaData in the standard ResultSet interface. The signature is as follows:

ResultSetMetaData getMetaData() throws SQLException

However, this method actually returns an instance of oracle.jdbc.StructMetaData, a class that supports structured object metadata in the same way that the standard java.sql.ResultSetMetaData interface specifies support for result set metadata.

The following method is also supported by StructMetaData:

String getOracleColumnClassName(int column) throws SQLException

This method returns the fully qualified name of the oracle.sql.Datum subclass whose instances are manufactured if the OracleResultSet interface getOracleObject



method is called to retrieve the value of the specified attribute. For example, oracle.sql.NUMBER.

To use the getOracleColumnClassName method, you must cast the ResultSetMetaData object, which that was returned by the getMetaData method, to StructMetaData.

Note: In all the preceding method signatures, column is something of a misnomer. Where you specify a value of 4 for column, you really refer to the fourth attribute of the object.

14.5.2 Retrieving Object Metadata

Use the following steps to obtain metadata about a structured object type:

- 1. Create or acquire a StructDescriptor instance that describes the relevant structured object type.
- 2. Call the getMetaData method on the StructDescriptor instance.
- 3. Call the metadata getter methods, getColumnName, getColumnType, and getColumnTypeName, as desired.

Note:

If one of the structured object attributes is itself a structured object, repeat steps 1 through 3.

Example 14-1 Example

The following method shows how to retrieve information about the attributes of a structured object type. This includes the initial step of creating a StructDescriptor instance.

```
//
// Print out the ADT's attribute names and types
11
void getAttributeInfo (Connection conn, String type name) throws SQLException
{
  // get the type descriptor
  StructDescriptor desc = StructDescriptor.createDescriptor (type name, conn);
  // get type metadata
  ResultSetMetaData md = desc.getMetaData ();
  // get # of attrs of this type
  int numAttrs = desc.length ();
  // temporary buffers
  String attr name;
  int attr_type;
  String attr typeName;
  System.out.println ("Attributes of "+type name+" :");
```



```
for (int i=0; i<numAttrs; i++)
{
    attr_name = md.getColumnName (i+1);
    attr_type = md.getColumnType (i+1);
    System.out.println (" index"+(i+1)+" name="+attr_name+" type="+attr_type);
    // drill down nested object
    if (attrType == OracleTypes.STRUCT)
    {
        attr_typeName = md.getColumnTypeName (i+1);
        // recursive calls to print out nested object metadata
        getAttributeInfo (conn, attr_typeName);
    }
}</pre>
```

15 Working with Large Objects and SecureFiles

Large Objects (LOBs) are a set of data types that are designed to hold large amounts of data. This chapter describes how to use Java Database Connectivity (JDBC) to access and manipulate LOBs and SecureFiles using either the data interface or the locator interface.

This chapter contains the following sections:

- The LOB Data Types
- Persistent LOBs
- Temporary LOBs
- Data Interface for LOBs
- Locator Interface for LOBs
- BFILEs
- JDBC Best Practices for LOB

15.1 The LOB Data Types

Oracle Database supports the following four LOB data types:

Binary large object (BLOB)

This data type is used for unstructured binary data.

Character large object (CLOB)

This data type is used for character data.

National character large object (NCLOB)

This data type is used for national character data.

• BFILE

This data type is used for large binary data objects stored in operating system files, outside of database tablespaces.

BLOBs, CLOBs, and NCLOBs are stored persistently in a database table and all operations performed on these data types are under transaction control. You can also create temporary LOBs of types BLOB, CLOB, or NCLOB to hold transient data. Both persistent and temporary LOBs can be accessed and manipulated using the Data Interface and the Locator Interface.

BFILE is an Oracle proprietary data type that provides read-only access to data located outside the database tablespaces on tertiary storage devices, such as hard disks, network mounted files systems, CD-ROMs, PhotoCDs, and DVDs. BFILE data is not under transaction control and is not stored by database backups.

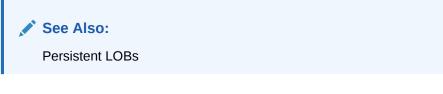
The PL/SQL language supports the LOB data types and the JDBC interface allows passing IN parameters to PL/SQL procedures or functions, and retrieval of OUT parameters or returns.



See Also: Introduction to Large Objects and SecureFiles

15.2 Persistent LOBs

A persistent LOB is a LOB instance that exists in a table row in the database. You can store persistent LOBs as SecureFiles or BasicFiles.



SecureFiles is the default storage mechanism for LOBs in database tables. SecureFile LOBs can only be created in tablespaces managed with Automatic Segment Space Management (ASSM). Oracle strongly recommends SecureFiles for storing and managing BLOBs, CLOBs, and NCLOBs.

Following Features of Oracle SecureFiles are transparently available to JDBC programs through the existing APIs:

- Compression enables users to compress data to save disk space.
- Encryption provides an encryption facility that enables random read and write operations of the encrypted data.
- Deduplication enables automatically detect duplicate LOB data and conserve space by storing only one copy of the data.

isSecureFile Method

You can check whether or not your BLOB or CLOB data uses Oracle SecureFile storage. To achieve this, use the following method from the oracle.jdbc.OracleBlob or the oracle.jdbc.OracleClob class:

public boolean isSecureFile() throws SQLException

If this method returns ${\tt true},$ then your data uses SecureFile storage.

Both persistent and temporary LOBs can be accessed and manipulated using the Data Interface and the Locator Interface.

See Also:

- Data Interface for LOBs
- Locator Interface for LOBs

15.3 Temporary LOBs

You can use temporary LOBs to store transient data. Temporary LOBs reside in either the PGA memory or the temporary tablespace, depending on their size.

You can insert temporary LOBs into a regular database table. In such a case, a permanent copy of the LOB is created and stored.



Creating a Temporary LOB

You create a temporary LOB with the static createTemporary method, defined in both the oracle.sql.BLOB and oracle.sql.CLOB classes. You can also create a temporary LOB by using the connection factory methods available in JDBC 4.0. The Oracle JDBC drivers implement the factory methods, createBlob, createClob, and createNClob in the java.sql.Connection interface to create temporary LOBs.

Freeing a Temporary LOB

You free a temporary LOB using the freeTemporary method. You can test whether a LOB is temporary or not by calling the isTemporary method. If the LOB was created by calling the createTemporary method, then the isTemporary method returns true, else it returns false.

Starting with Oracle Database Release 21c, you do not need to check whether a LOB is temporary or persistent before releasing the temporary LOB. If you call the DBMS_LOB.FREETEMPORARY procedure or the OCILobFreeTemporary() function on a LOB, it will perform either of the following operations:

- For a temporary LOB, it will release the LOB.
- For a persistent LOB, it will do nothing (no-op).

Note:

- You must free any temporary LOBs before ending the session or call. If you do not free a temporary LOB, then it will make the storage used by that LOB in the database unavailable. Frequent failure to free temporary LOBs will result in filling up temporary table space with unavailable LOB storage.
- The JDBC 4.0 free method, present in the java.sql.Blob, java.sql.Clob, and java.sql.NClob interfaces, supersedes the freeTemporary method.

Both persistent and temporary LOBs can be accessed and manipulated using the Data Interface and the Locator Interface.



See Also:

- Data Interface for LOBs
- Locator Interface for LOBs

15.4 Data Interface for LOBs

The data interface for LOBs includes a set of Java and OCI APIs that are extended to work with LOB data types.

The data interface uses standard JDBC methods such as the getString method and the setBytes method to read and write LOB data. Unlike the standard java.sql.Blob, java.sql.Clob, and java.sql.NClob interfaces, the data interface does not provide random access capability, that is, it does not use LOB locator and cannot access data beyond a size of 2 gigabytes.



You can use the data interface for LOBs to store and manipulate character data and binary data in a LOB column as if it were stored in the corresponding legacy data types like VARCHAR2, LONG, RAW, and so on. This section describes the following topics:

- Input
- Output
- CallableSatement and IN OUT Parameter
- Size Limitations

15.4.1 Input

The setBytes, setBinaryStream, setString, setCharacterStream, and setAsciiStream methods of the PreparedStatement interface are extended to enhance the ability to work with BLOB, CLOB, and NCLOB target columns. If the length of the data is known, then for better performance, use the versions of setBinaryStream or setCharacterStream methods that accept the data length as a parameter.

Note:

These methods do not work with BFILE data because it is read-only.

For the JDBC Oracle Call Interface (OCI) and Thin drivers, there is no limitation on the size of the byte array, String, or the length specified for the stream functions, except the limits imposed by the Java language.



Note:

In Java, the array size is limited to positive Java int or 2 gigabytes of size.

For the server-side internal driver, currently there is a limitation of 32767 bytes for operations on SQL statements, such as an INSERT statement. This limitation does not apply for PL/SQL statements. You can use the following workaround for an INSERT statement, where you can wrap the LOB in a PL/SQL block:

```
BEGIN
INSERT id, c INTO clob_tab VALUES(?,?);
END;
```

Input Modes for LOBs

LOBs have the following three input modes:

Direct binding

This binding is limited in size but most efficient. It places the data for all input columns inline in the block of data sent to the server. All data, including multiple executions of a batch, is sent in a single network operation.

Stream binding

This binding places data at the end. It limits batch size to one and may require multiple round trips to complete.

LOB binding

This binding creates a temporary LOB, copies data to the LOB, and binds the LOB locator. The temporary LOB is automatically freed after execution. The operation to create the temporary LOB and then to writing to the LOB requires multiple round trips. The input of the locators may be batched.

You must bear in mind the following automatic switching of the input mode for LOBs:

- For SQL statements:
 - The setBytes and setBinaryStream methods use direct binding for data less than 32767 bytes.
 - The setBytes and setBinaryStream methods use stream binding for data larger than 32767 bytes.
 - Starting from JDBC 4.0, there are two forms of setAsciiStream, setBinaryStream, and setCharacterStream methods. The form that accepts a long argument as length, uses LOB binding for length larger than 2147483648. The form, where the length is not specified, always uses LOB binding.
 - The setString, setCharacterStream, and setAsciiStream methods use direct binding for data smaller than 32767 characters.
 - The setString, setCharacterStream, and setAsciiStream methods use stream binding for data larger than 32766 characters.
- For PL/SQL statements:
 - The setBytes and setBinary stream methods use direct binding for data less than 32767 bytes.



- The setBytes and setBinaryStream methods use LOB binding for data larger than 32766 bytes.
- The setString, setCharacterStream, and setAsciiStream methods use direct binding for data smaller than 32767 bytes in the database character set.
- The setString, setCharacterStream, and setAsciiStream methods use LOB binding for data larger than 32766 bytes in the database character set.
- Forced conversion to LOBs

The setBytesForBlob and setStringForClob methods, present in the oracle.jdbc.OraclePreparedStatement interface, use LOB binding for any data size.

Impact of Automatic Switching of Input Mode

The automatic switching of the input mode for large data has impact on certain programs. Previously, you used to get ORA-17157 errors for attempts to use the setString method for String values larger than 32766 characters. Now, depending on the type of the target parameter, an error may occur while the statement is executed or the operation may succeed.

Another impact is that the automatic switching may result in additional server-side parsing to adapt to the change in the parameter type. This results in a performance effect, if the data sizes vary above and below the limit for repeated executions of the statement. Switching to the stream modes effects batching as well.

15.4.2 Output

The getBytes, getBinaryStream, getString, getCharacterStream, and getAsciiStream methods of the ResultSet and CallableStatement interfaces work with BLOB, CLOB, and BFILE columns or OUT parameters. These methods work for any LOB of length less than 2147483648.

Note:

The getString and getNString methods cannot be used for retrieving BLOB column values.

The data interface operates by accessing the LOB locators within the driver and is transparent to the application programming interface.

You can read BFILE data, and read and write BLOB or CLOB data using the defineColumnType method. To read, use the defineColumnType(nn, Types.LONGVARBINARY) or the defineColumnType(nn, Types.LONGVARCHAR) method on the column. This produces a direct stream on the data as if it were a LONG RAW or LONG column, and gives the fastest read performance on LOBs.

You can also use LOB prefetching to reduce or eliminate any additional database round trips.

Related Topics

New Methods for National Character Set Type Data in JDK 6



Locator Interface for LOBs

Locators are small data structures, which contain information that may be used to access the actual data of the LOB. In a database table, the locator is stored directly in the table, while the data may be in the table or in separate storage.

15.4.3 CallableSatement and IN OUT Parameter

If you have an IN OUT CLOB parameter of a stored procedure and want to use the setString method for setting the value for this parameter, then for any IN and OUT parameter, the binds must be of the same type.

Note:

It is a PL/SQL requirement that the Java types used as input and output for an IN OUT parameter must be the same. The automatic switching of types done by the extensions described in this chapter may cause problems with this.

The automatic switching of the input mode causes problems if you are not sure of the data sizes. For example, if you know that neither the input data nor the output data will ever be larger than 32766 bytes, then you can use the setString method for the input parameter and register the OUT parameter as Types.VARCHAR and use the getString method for the output parameter.

A better solution is to change the stored procedure to have separate IN and OUT parameters. That is, if you have the following piece of code in your application:

CREATE PROCEDURE clob_proc(c IN OUT CLOB);

then, change it to:

CREATE PROCEDURE clob_proc(c_in IN CLOB, c_out OUT CLOB);

Another workaround is to use a container block to make the call. The clob_proc procedure can be wrapped with a Java String to use for the prepareCall statement, as follows:

"DECLARE c temp; BEGIN c temp := ?; clob proc(c temp); ? := c temp; END;"

In either case, you can use the setString method on the first parameter and the registerOutParameter method with Types.CLOB on the second.

15.4.4 Size Limitations

You must be aware of the effect on the performance of the Java memory management system due to creation of a very large byte array or a String. Read the information provided by your Java virtual machine (JVM) vendor about the impact of very large data elements on memory management, and consider using the stream interfaces instead.



15.5 Locator Interface for LOBs

Locators are small data structures, which contain information that may be used to access the actual data of the LOB. In a database table, the locator is stored directly in the table, while the data may be in the table or in separate storage.



Starting from JDBC 4.0, you must use the java.sql.Blob, java.sql.Clob, and java.sql.NClob interfaces for performing read and write operations on LOBs. These interfaces provide random access to the data in the LOB.

The Oracle implementation classes <code>oracle.sql.BLOB</code>, <code>oracle.sql.CLOB</code>, and <code>oracle.sql.NCLOB</code> store the locator and access the data with it. The <code>oracle.sql.BLOB</code> and <code>oracle.sql.CLOB</code> classes implement the <code>java.sql.Blob</code> and <code>java.sql.Clob</code> interfaces respectively. In <code>ojdbc6.jar</code>, <code>oracle.sql.NCLOB</code> implements java.sql.NClob, but in <code>ojdbc5.jar</code>, it implements the <code>java.sql.Clob</code> interface.

Note:

Starting from Oracle Database 12c Release 1 (12.1), the oracle.sql.BLOB and oracle.sql.CLOB classes are deprecated and replaced with the oracle.jdbc.OracleBlob and oracle.jdbc.OracleClob interfaces. Oracle recommends you to use the methods available in the java.sql package, where possible, for standard compatibility and methods available in the oracle.jdbc package for Oracle specific extensions. Refer to MoS Note 1364193.1 for more information about these interface.

In Oracle Database 23c, the Oracle JDBC drivers support the JDBC 4.0 java.sql.NClob interface in ojdbc11.jar, which is compiled with JDK 11.

In contrast, the oracle.sql.BFILE is an Oracle extension, without a corresponding java.sql interface.





15.5.1 LOB prefetching

For the current release of JDBC drivers, the number of round trips is reduced by prefetching the metadata such as the LOB length, the chunk size, and the beginning of the LOB data, along with the locator during regular fetch operations.

If you select LOB columns into a result set, then some or all of the data is prefetched to the client, when the locator is fetched. It saves the first round trip to retrieve data by deferring all preceding operations until fetching from the locator.

Note:

- LOBs are not prefetched in abstract data types (ADTs) like STRUCTs and ARRAYs. This behavior is exhibited even if you set the value of the oracle.jdbc.defaultLobPrefetchSize connection property. If the client application performs an operation that depends on the value of a LOB, which is embedded in such a data type, then additional round-trips are required to fetch the value.
- The benefits of prefetching are more for small LOBs and less for larger LOBs.
- You must be aware of the possible memory consumption while setting large LOB prefetch sizes in combination with a large row prefetch size and a large number of LOB columns.

The default prefetch size is 32768. You can specify the prefetch size in bytes for BLOBs and in characters for CLOBs, using the oracle.jdbc.defaultLobPrefetchSize connection property. You can override the value of this property in the following two ways:

- At the statement level: By using the oracle.jdbc.OracleStatement.setLobPrefetchSize(int) method
- At the column level: By using the form of the defineColumnType method that takes length as argument



15.5.2 LOB Open and Close Operations

This section discusses how to open and close your LOBs.

This section discusses how to open and close your LOBs. The JDBC implementation of this functionality is provided using the following methods available in the <code>oracle.sql.BLOB</code> and <code>oracle.sql.CLOB</code> interfaces:

- void open (int mode)
- void close()



boolean isOpen()

Note:

You do not have to necessarily open and close your LOBs. You may choose to open and close those for performance reasons.

See Also:

LOB Open and Close Operations

15.6 BFILEs

BFILEs are data objects stored in operating system files, outside the database tablespaces. Data stored in a table column of type BFILE is physically located in an operating system file, not in the database. The BFILE column stores a reference to the operating system file.

BFILEs are read-only. The body of the data resides in the operating system (OS) file system and you can write to BFILEs using only OS tools and commands. You can create a BFILE for an existing external file by executing the appropriate SQL statement using either JDBC APIs or any other way to execute SQL. However, using SQL or JDBC, you cannot create an OS file that a BFILE refers to. Such OS files are created only by an external process that has access to server file systems.



This section describes how to use file locators to perform read and write operations on BFILEs. This section covers the following topics:

- Retrieving BFILE Locators
- Inserting BFILES

Retrieving BFILE Locators

Both the BFILE data type and the oracle.jdbc.OracleBfile interface to work with the BFILEs are Oracle proprietary. So, there is no standard interface for them. You must use Oracle extensions for this type of data.

If you have a standard JDBC result set or callable statement object that includes BFILE locators, then you can access the locators by using the standard result set getObjectmethod. This method returns an oracle.jdbc.OracleBfile object.

You can also access the locators by casting your result set to <code>OracleResultSet</code> or your callable statement to <code>OracleCallableStatement</code> and using the <code>getOracleObject</code> or <code>getBFILE</code> method.



Note:

If you are using getObject or getOracleObject methods, then remember to cast the output, as necessary.

Once you have a locator, you can access the BFILE data through the APIs present in the oracle.jdbc.OracleBfile class. These APIs are similar to the read methods of the oracle.jdbc.OracleBfile interface.

Inserting BFILES

You can use an instance of the oracle.jdbc.OracleBfile interface as input to a SQL statement or to a PL/SQL procedure. You can achieve this by performing one of the following:

- Use the standard setObject method.
- Cast the statement to OraclePreparedStatement or OracleCallableStatement, and use the setOracleObject or setBFILE method. These methods take the parameter index and an oracle.jdbc.OracleBfile object as input.

Note:

- There is no standard java.sql interface for BFILEs.
- Use the getBFILE methods in the OracleResultSet and OracleCallableStatement interfaces to retrieve an oracle.jdbc.OracleBfile object. The setBFILE methods in OraclePreparedStatement and OracleCallableStatement interfaces accept oracle.jdbc.OracleBfile object as an argument. Use these methods to write to a BFILE.
- Oracle recommends that you use the getBFILE, setBFILE, and updateBFILE methods instead of the getBfile, setBfile, and updateBfile methods. For example, use the setBFILE method instead of the setBfile method.

15.7 JDBC Best Practices for LOB

You can enhance the performance of your applications if you reduce the number of roundtrips to the database. This section describes how to minimize the number of round-trips to the database.

If you know the maximum size of your LOB data, and you intend to perform read or write operation on the entire LOB, then use the Data Interface following these guidelines:

- For read operations, define the LOB as character type or binary type using the DefineColumnType method.
- For write operations, bind the LOB as character type or binary type using the setString or the setBytes method.



If you do not know the maximum size of your LOB data, then use the LOB APIs with LOB prefetching for read operations. Also, define the LOB prefetch size to a value that can accommodate majority of the LOB values in the column.

See Also: LOB prefetching



16 Using Oracle Object References

This chapter describes the standard Java Database Connectivity (JDBC) that let you access and manipulate object references.

This section discusses the following topics:

- Oracle Extensions for Object References
- Retrieving and Passing an Object Reference
- Accessing and Updating Object Values Through an Object Reference

16.1 Oracle Extensions for Object References

Oracle supports the use of references to database objects. Oracle JDBC provides support for object references as:

- Columns in a SELECT clause
- IN or OUT bind variables
- Attributes in an Oracle object
- Elements in a collection type object

In SQL, an object reference (REF) is strongly typed. For example, a reference to an EMPLOYEE object would be defined as an EMPLOYEE REF, not just a REF.

When you select an object reference, be aware that you are retrieving only a pointer to an object, not the object itself. You have the choice of materializing the reference as a <code>java.sql.Ref</code> instance for portability, or materializing it as an instance of a custom Java class that you have created in advance, which is strongly typed. Custom Java classes used for object references are referred to as **custom reference classes** and must implement the <code>oracle.jdbc.OracleData</code> interface.

You can retrieve a REF instance through a result set or callable statement object, and pass an updated REF instance back to the database through a prepared statement or callable statement object. The REF class includes functionality to get and set underlying object attribute values, and get the SQL base type name of the underlying object.

Custom reference classes include this same functionality, as well as having the advantage of being strongly typed. This can help you find coding errors during compilation that might not otherwise be discovered until run time.



Note:

- If you are using the oracle.jdbc.OracleData interface for custom object classes, then you will presumably use OracleData for corresponding custom reference classes as well. However, if you are using the standard java.sql.SQLData interface for custom object classes, then you can only use weak Java types for references. The SQLData interface is for mapping SQL object types only.
- You can create and retrieve REF objects in your JDBC application only by running SQL statements. There is no JDBC-specific functionality for creating and retrieving REF objects.
- You cannot have a reference to an array, even though arrays, like objects, are structured types.

16.2 Retrieving and Passing an Object Reference

This section discusses JDBC functionality for retrieving and passing object references. It covers the following topics:

- Retrieving an Object Reference from a Result Set
- Retrieving an Object Reference from a Callable Statement
- Passing an Object Reference to a Prepared Statement

16.2.1 Retrieving an Object Reference from a Result Set

To demonstrate how to retrieve object references, the following example first defines an Oracle object type ADDRESS, which is then referenced in the PEOPLE table:

```
create type ADDRESS as object
  (street_name VARCHAR2(30),
   house_no NUMBER);
create table PEOPLE
  (coll VARCHAR2(30),
   col2 NUMBER,
   col3 REF ADDRESS);
```

The ADDRESS object type has two attributes: a street name and a house number. The PEOPLE table has three columns: a column for character data, a column for numeric data, and a column containing a reference to an ADDRESS object.

To retrieve an object reference, follow these general steps:

- 1. Use a standard SQL SELECT statement to retrieve the reference from a database table REF column.
- 2. Use getRef to get the address reference from the result set into an OracleRef instance.
- 3. Let Address be the Java custom class corresponding to the SQL object type ADDRESS.



- 4. Add the correspondence between the Java class Address and the SQL type ADDRESS to your type map.
- 5. Use the getObject method to retrieve the contents of the Address reference. Cast the output to Address.

The PEOPLE database table is defined earlier in this section. The code for the preceding steps, except the step of adding Address to the type map, is as follows:

```
ResultSet rs = stmt.executeQuery("SELECT col3 FROM PEOPLE");
while (rs.next())
{
    OracleRef ref = rs.getRef(1);
    Address a = (Address)ref.getObject();
}
```

Note:

In the preceding code, stmt is a previously defined statement object.

16.2.2 Retrieving an Object Reference from a Callable Statement

To retrieve an object reference as an OUT parameter in PL/SQL blocks, you must register the bind type for your OUT parameter.

1. Cast your callable statement to OracleCallableStatement, as follows:

```
OracleCallableStatement ocs =
   (OracleCallableStatement)conn.prepareCall("{? = call func()}");
```

2. Register the OUT parameter with the following form of the registerOutParameter method:

ocs.registerOutParameter (int param_index, int sql_type, String sql_type_name);

param_index is the parameter index and sql_type is the SQL type code. The sql_type_name is the name of the structured object type that this reference is used for. For example, if the OUT parameter is a reference to an ADDRESS object, then ADDRESS is the sql_type_name that should be passed in.

3. Run the call, as follows:

ocs.execute();

16.2.3 Passing an Object Reference to a Prepared Statement

Pass an object reference to a prepared statement in the same way as you would pass any other SQL type. Use either the setObject method or the setREF method of a prepared statement object.

Use a prepared statement to update an address reference based on ROWID, as follows:

```
PreparedStatement pstmt =
    conn.prepareStatement ("update PEOPLE set ADDR_REF = ? where ROWID = ?");
pstmt.setRef (1, addr_ref);
    pstmt.setRowId (2, rowid);
```



16.3 Accessing and Updating Object Values Through an Object Reference

You can use the Ref object setObject method to update the value of an object in the database through an object reference. To do this, you must first retrieve the reference to the database object and create a Java object that corresponds to the database object.

For example, you can use the code in the "Retrieving and Passing an Object Reference" section to retrieve the reference to a database ADDRESS object, as shown in the following code snippet:

```
ResultSet rs = stmt.executeQuery("SELECT col3 FROM PEOPLE");
if (rs.next())
{
    Ref ref = rs.getRef(1);
    Address a = (Address)ref.getObject();
}
```

Then, you can create a Java Address object that corresponds to the database ADDRESS object. Use the setObject method of the Ref interface to set the value of the database object, as follows:

```
Address addr = new Address(...);
ref.setObject(addr);
```

Here, the setValue method updates the database ADDRESS object immediately.

Related Topics

Retrieving and Passing an Object Reference



17 Working with Oracle Collections

This chapter describes Oracle extensions to standard Java Database Connectivity (JDBC) that let you access and manipulate Oracle collections, which map to Java arrays, and their data. The following topics are discussed:

Note:

Starting from Oracle Database 12c Release 1 (12.1), the oracle.sql.ARRAY class is deprecated and replaced with the oracle.jdbc.OracleArray interface, which is a part of the oracle.jdbc package. Oracle recommends you to use the methods available in the java.sql package, where possible, for standard compatibility and methods available in the oracle.jdbc package for Oracle specific extensions. Refer to MoS Note 1364193.1 for more information about the oracle.jdbc.OracleArray interface.

- Oracle Extensions for Collections
- Overview of Collection Functionality
- ARRAY Performance Extension Methods
- Creating and Using Arrays
- Using a Type Map to Map Array Elements

17.1 Oracle Extensions for Collections

This section covers the following topics:

- Overview of Oracle Collections
- Choices in Materializing Collections
- Creating Collections
- Creating Multilevel Collection Types

17.1.1 Overview of Oracle Collections

An Oracle collection, either a variable array (VARRAY) or a nested table in the database, maps to an array in Java. JDBC 2.0 arrays are used to materialize Oracle collections in Java. The terms collection and array are sometimes used interchangeably. However, collection is more appropriate on the database side and array is more appropriate on the JDBC application side.

Oracle supports only named collections, where you specify a SQL type name to describe a type of collection. JDBC enables you to use arrays as any of the following:

• Columns in a SELECT clause



- IN or OUT bind variables
- Attributes in an Oracle object
- Elements of other arrays

17.1.2 Choices in Materializing Collections

In your application, you have the choice of materializing a collection as an instance of the oracle.sql.ARRAY class, which is weakly typed, or materializing it as an instance of a custom Java class that you have created in advance, which is strongly typed. Custom Java classes used for collections are referred to as custom collection classes. A custom collection class must implement the Oracle oracle.jdbc.OracleData interface. In addition, the custom class or a companion class must implement oracle.jdbc.OracleDataFactory. The standard java.sql.SQLData interface is for mapping SQL object types only.

The oracle.sql.ARRAY class implements the standard java.sql.Array interface.

The ARRAY class includes functionality to retrieve the array as a whole, retrieve a subset of the array elements, and retrieve the SQL base type name of the array elements. However, you cannot write to the array, because there are no setter methods.

Custom collection classes, as with the ARRAY class, enable you to retrieve all or part of the array and get the SQL base type name. They also have the advantage of being strongly typed, which can help you find coding errors during compilation that might not otherwise be discovered until run time.

Note:

There is no difference in the code between accessing VARRAYs and accessing nested tables. ARRAY class methods can determine if they are being applied to a VARRAY or nested table, and respond by taking the appropriate actions.

17.1.3 Creating Collections

Because Oracle supports only named collections, you must declare a particular VARRAY type name or nested table type name. VARRAY and nested table are not types themselves, but categories of types.

A SQL type name is assigned to a collection when you create it using the SQL CREATE TYPE statement:

CREATE TYPE <sql_type_name> AS <datatype>;

A VARRAY is an array of varying size. It has an ordered set of data elements, and all the elements are of the same data type. Each element has an index, which is a number corresponding to the position of the element in the VARRAY. The number of elements in a VARRAY is the size of the VARRAY. You must specify a maximum size when you declare the VARRAY type. For example:

CREATE TYPE myNumType AS VARRAY(10) OF NUMBER;



This statement defines myNumType as a SQL type name that describes a VARRAY of NUMBER values that can contain no more than 10 elements.

A nested table is an unordered set of data elements, all of the same data type. The database stores a nested table in a separate table which has a single column, and the type of that column is a built-in type or an object type. If the table is an object type, then it can also be viewed as a multi-column table, with a column for each attribute of the object type. You can create a nested table as follows:

CREATE TYPE myNumList AS TABLE OF integer;

This statement identifies myNumList as a SQL type name that defines the table type used for the nested tables of the type INTEGER.

17.1.4 Creating Multilevel Collection Types

The most common way to create a new multilevel collection type in JDBC is to pass the SQL CREATE TYPE statement to the execute method of the java.sql.Statement class. The following code creates a one-level nested table, first_level, and a two-levels nested table, second level:

```
Connection conn = ....
                                                  // make a database
                                                  // connection
Statement stmt = conn.createStatement();
                                                  // open a database
                                                  // cursor
stmt.execute("CREATE TYPE first level AS TABLE OF NUMBER"); // create a nested
                                                 // table of number
stmt.execute("CREATE TYPE second level AS TABLE OF first level"); // create a
          // two-levels nested table
         // other operations here
. . .
stmt.close();
                                                  // release the
                                                  // resource
                                                  // close the
conn.close();
                                                  // database connection
```

Once the multilevel collection types have been created, they can be used as both columns of a base table as well as attributes of a object type.

Note:

Multilevel collection types are available only for Oracle9i and later.

17.2 Overview of Collection Functionality

You can obtain collection data in an array instance through a result set or callable statement and pass it back as a bind variable in a prepared statement or callable statement.

The oracle.sql.ARRAY class, which implements the standard java.sql.Array interface, provides the necessary functionality to access and update the data of an Oracle collection.

This section covers Array Getter and Setter Methods. Use the following result set, callable statement, and prepared statement methods to retrieve and pass collections as Java arrays.



Note:

Starting from Oracle Database 12c Release 1 (12.1), the oracle.sql.ARRAY class is deprecated and replaced with the oracle.jdbc.OracleArray interface, which is a part of the oracle.jdbc package. Oracle recommends you to use the methods available in the java.sql package, where possible, for standard compatibility and methods available in the oracle.jdbc package for Oracle specific extensions. Refer to MoS Note 1364193.1 for more information about the oracle.jdbc.OracleArray interface.

Result Set and Callable Statement Getter Methods

The OracleResultSet and OracleCallableStatement interfaces support getARRAY and getArray methods to retrieve ARRAY objects as output parameters, either as oracle.sql.ARRAY instances or java.sql.Array instances. You can also use the getObject method. These methods take as input a String column name or int column index.

Note:

The Oracle JDBC drivers cache array and structure descriptors. This provides enormous performance benefits; however, it means that if you change the underlying type definition of an array type in the database, the cached descriptor for that array type will become stale and your application will receive a SQLException.

Prepared and Callable Statement Setter Methods

The OraclePreparedStatement and OracleCallableStatement classes support setARRAY and setArray methods to take updated ARRAY objects as bind variables and pass them to the database. You can also use the setObject method. These methods take as input a String parameter name or int parameter index as well as an oracle.sql.ARRAY instance or a java.sql.Array instance.

17.3 ARRAY Performance Extension Methods

This section discusses the following topics:

- About Accessing oracle.sql.ARRAY Elements as Arrays of Java Primitive Types
- ARRAY Automatic Element Buffering
- ARRAY Automatic Indexing

17.3.1 About Accessing oracle.sql.ARRAY Elements as Arrays of Java Primitive Types

The oracle.sql.ARRAY class contains methods that return array elements as Java primitive types. These methods enable you to access collection elements more



efficiently than accessing them as Datum instances and then converting each Datum instance to its Java primitive value.

Note:

These specialized methods of the oracle.sql.ARRAY class are restricted to numeric collections.

Each method using the first signature returns collection elements as an *XXX*[], where *XXX* is a Java primitive type. Each method using the second signature returns a slice of the collection containing the number of elements specified by count, starting at the index location.

17.3.2 ARRAY Automatic Element Buffering

Oracle JDBC driver provides public methods to enable and disable buffering of ARRAY contents.

The following methods are included with the oracle.sql.ARRAY class:

- setAutoBuffering
- getAutoBuffering

It is advisable to enable auto-buffering in a JDBC application when the ARRAY elements will be accessed more than once by the getAttributes and getArray methods, presuming the ARRAY data is able to fit into the Java Virtual Machine (JVM) memory without overflow.

Note:

Buffering the converted elements may cause the JDBC application to consume a significant amount of memory.

When you enable auto-buffering, the oracle.sql.ARRAY object keeps a local copy of all the converted elements. This data is retained so that a second access of this information does not require going through the data format conversion process.

17.3.3 ARRAY Automatic Indexing

If an array is in auto-indexing mode, then the array object maintains an index table to hasten array element access.

The oracle.sql.ARRAY class contains the following methods to support automatic array-indexing:

- setAutoIndexing(boolean)
- setAutoIndexing(boolean, int)

By default, auto-indexing is not enabled. For a JDBC application, enable auto-indexing for ARRAY objects if random access of array elements may occur through the getArray and getResultSet methods.



17.4 Creating and Using Arrays

This section discusses how to create array objects and how to retrieve and pass collections as array objects, including the following topics.

- Creating ARRAY Objects
- Retrieving an Array and Its Elements
- Passing Arrays to Statement Objects

17.4.1 Creating ARRAY Objects

Note:

Oracle JDBC does not support the JDBC 4.0 method createArrayOf method of java.sql.Connection interface. This method only allows anonymous array types, while all Oracle array types are named. Use the Oracle specific method oracle.jdbc.OracleConnection.createARRAY instead.

This section describes how to create ARRAY objects. This section covers the following topics:

- Steps in Creating ARRAY Objects
- Example 17-1

Steps in Creating ARRAY Objects

Starting from Oracle Database 11g Release 1, you can use the createARRAY factory method of oracle.jdbc.OracleConnection interface to create an array object. The factory method for creating arrays has been defined as follows:

```
public ARRAY createARRAY(java.lang.String typeName,java.lang.Object
elements)throws SQLException
```

where, typeName is the name of the SQL type of the created object and elements is the elements of the created object.

Perform the following to create an array:

1. Create a collection with the CREATE TYPE statement as follows:

```
CREATE TYPE elements AS varray(22) OF NUMBER(5,2);
```

The two possibilities for the contents of elements are:

- An array of Java primitives. For example, int[].
- An array of Java objects, such as xxx[], where xxx is the name of a Java class. For example, Integer[].



The setARRAY, setArray, and setObject methods of the OraclePreparedStatement class take an object of the type oracle.sql.ARRAY as an argument, not an array of objects.

2. Construct the ARRAY object by passing the Java string specifying the user-defined SQL type name of the array and a Java object containing the individual elements you want the array to contain.

ARRAY array = oracle.jdbc.OracleConnection.createARRAY(sql type name, elements);

Note:

The name of the collection type is not the same as the type name of the elements. For example:

```
CREATE TYPE person AS object
(c1 NUMBER(5), c2 VARCHAR2(30));
CREATE TYPE array_of_persons AS varray(10)
OF person;
```

In the preceding statements, the name of the collection type is <code>ARRAY_OF_PERSON</code>. The SQL type name of the collection elements is <code>PERSON</code>.

Example 17-1 Creating Multilevel Collections

As with single-level collections, the JDBC application can create an oracle.sql.ARRAY instance to represent a multilevel collection, and then send the instance to the database. The same createARRAY factory method you use to create single-level collections, can be used to create multilevel collections as well. To create a single-level collection, the elements are a one dimensional Java array, while to create a multilevel collection, the elements can be either an array of oracle.sql.ARRAY[] elements or a nested Java array or the combinations.

The following code shows how to create collection types with a nested Java array:

```
// prepare the multilevel collection elements as a nested Java array
int[][][] elements = { {{1}, {1, 2}}, {{2, 3}}, {{3, 4}} };
// create the ARRAY using the factory method
ARRAY array = oracle.jdbc.OracleConnection.createARRAY(sql type name, elements);
```

17.4.2 Retrieving an Array and Its Elements

This section first discusses how to retrieve an ARRAY instance as a whole from a result set, and then how to retrieve the elements from the ARRAY instance. This section covers the following topics:

- About Retrieving the Array
- Data Retrieval Methods
- Comparing the Data Retrieval Methods



- Retrieving Elements of a Structured Object Array According to a Type Map
- Retrieving a Subset of Array Elements
- Retrieving Array Elements into an oracle.sql.Datum Array
- About Accessing Multilevel Collection Elements

17.4.2.1 About Retrieving the Array

You can retrieve a SQL array from a result set by casting the result set to OracleResultSet and using the getARRAY method, which returns an oracle.sql.ARRAY object. If you want to avoid casting the result set, then you can get the data with the standard getObject method specified by the java.sql.ResultSet interface and cast the output to oracle.sql.ARRAY.

17.4.2.2 Data Retrieval Methods

Once you have an ARRAY object, you can retrieve the data using one of these three overloaded methods of the oracle.sql.ARRAY class:

- getArray
- getOracleArray
- getResultSet

Oracle also provides methods that enable you to retrieve all the elements of an array, or a subset.

Note:

In case you are working with an array of structured objects, Oracle provides versions of these three methods that enable you to specify a type map so that you can choose how to map the objects to Java.

getOracleArray

The getOracleArray method is an Oracle-specific extension that is not specified in the standard Array interface. The getOracleArray method retrieves the element values of the array into a Datum[] array. The elements are of the oracle.sql.* data type corresponding to the SQL type of the data in the original array.

For an array of structured objects, this method will use <code>oracle.jdbc.OracleStruct</code> instances for the elements.

Oracle also provides a getOracleArray(*index*, *count*) method to get a subset of the array elements.

getResultSet

The getResultSet method returns a result set that contains elements of the array designated by the ARRAY object. The result set contains one row for each array element, with two columns in each row. The first column stores the index into the array for that element, and the second column stores the element value. In the case of VARRAYs, the index represents the position of the element in the array. In the case of



nested tables, which are by definition unordered, the index reflects only the return order of the elements in the particular query.

Oracle recommends using getResultSet when getting data from nested tables. Nested tables can have an unlimited number of elements. The ResultSet object returned by the method initially points at the first row of data. You get the contents of the nested table by using the next method and the appropriate getXXX method. In contrast, getArray returns the entire contents of the nested table at one time.

The getResultSet method uses the default type map of the connection to determine the mapping between the SQL type of the Oracle object and its corresponding Java data type. If you do not want to use the default type map of the connection, another version of the method, getResultSet (map), enables you to specify an alternate type map.

Oracle also provides the getResultSet(*index*, *count*) and getResultSet(*index*, *count*, *map*) methods to retrieve a subset of the array elements.

getArray

The getArray method is a standard JDBC method that returns the array elements as a java.lang.Object, which you can cast as appropriate. The elements are converted to the Java types corresponding to the SQL type of the data in the original array.

Oracle also provides a getArray(index, count) method to retrieve a subset of the array elements.

17.4.2.3 Comparing the Data Retrieval Methods

If you use getOracleArray to return the array elements, then the use by that method of oracle.sql.Datum instances avoids the expense of data conversion from SQL to Java. The non-character data inside the instance of a Datum class or any of its subclass remains in raw SQL format.

If you use getResultSet to return an array of primitive data types, then the JDBC driver returns a ResultSet object that contains, for each element, the index into the array for the element and the element value. For example:

ResultSet rset = intArray.getResultSet();

In this case, the result set contains one row for each array element, with two columns in each row. The first column stores the index into the array and the second column stores the element value.

If the elements of an array are of a SQL type that maps to a Java type, then getArray returns an array of elements of this Java type. The return type of the getArray method is java.lang.Object. Therefore, the result must be cast before it can be used.

BigDecimal[] values = (BigDecimal[]) intArray.getArray();

Here intArray is an oracle.sql.ARRAY, corresponding to a VARRAY of type NUMBER. The values array contains an array of elements of type java.math.BigDecimal, because the SQL NUMBER data type maps to Java BigDecimal, by default, according to Oracle JDBC drivers.



Using BigDecimal is a resource-intensive operation in Java. Because Oracle JDBC maps numeric SQL data to BigDecimal by default, using getArray may impact performance, and is not recommended for numeric collections.

17.4.2.4 Retrieving Elements of a Structured Object Array According to a Type Map

By default, if you are working with an array whose elements are structured objects, and you use getArray or getResultSet, then the Oracle objects in the array will be mapped to their corresponding Java data types according to the default mapping. This is because these methods use the default type map of the connection to determine the mapping.

However, if you do not want default behavior, then you can use the getArray(map) or getResultSet(map) method to specify a type map that contains alternate mappings. If there are entries in the type map corresponding to the Oracle objects in the array, then each object in the array is mapped to the corresponding Java type specified in the type map. For example:

Object[] object = (Object[])objArray.getArray(map);

Where objArray is an oracle.sql.ARRAY object and map is a java.util.Map object.

If the type map does not contain an entry for a particular Oracle object, then the element is returned as an oracle.jdbc.OracleStruct object.

The getResultSet (map) method behaves similarly to the getArray (map) method.

Related Topics

Using a Type Map to Map Array Elements

17.4.2.5 Retrieving a Subset of Array Elements

If you do not want to retrieve the entire contents of an array, then you can use signatures of getArray, getResultSet, and getOracleArray that let you retrieve a subset. To retrieve a subset of the array, pass in an index and a count to indicate where in the array you want to start and how many elements you want to retrieve. As previously described, you can specify a type map or use the default type map for your connection to convert to Java types. For example:

Object object = arr.getArray(index, count, map);
Object object = arr.getArray(index, count);

Similar examples using getResultSet are:

```
ResultSet rset = arr.getResultSet(index, count, map);
ResultSet rset = arr.getResultSet(index, count);
```

A similar example using getOracleArray is:

```
Datum[] arr = arr.getOracleArray(index, count);
```



Where arr is an oracle.sql.ARRAY object, index is type long, count is type int, and map is a java.util.Map object.

Note:

There is no performance advantage in retrieving a subset of an array, as opposed to the entire array.

17.4.2.6 Retrieving Array Elements into an oracle.sql.Datum Array

Use getOracleArray to return an oracle.sql.Datum[] array. The elements of the returned array is of oracle.sql.* type that correspond to the SQL data type of the elements of the original array. For example:

```
Datum arraydata[] = arr.getOracleArray();
```

arr is an oracle.sql.ARRAY object.

The following example assumes that a connection object conn and a statement object stmt have already been created. In the example, an array with the SQL type name NUM_ARRAY is created to store a VARRAY of NUMBER data. The NUM_ARRAY is in turn stored in a table VARRAY TABLE.

A query selects the contents of the VARRAY_TABLE. The result set is cast to OracleResultSet. The getARRAY method is applied to it to retrieve the array data into my_array, which is an oracle.sql.ARRAY object.

Because my_array is of type oracle.sql.ARRAY, you can apply the methods getSQLTypeName and getBaseType to it to return the name of the SQL type of each element in the array and its integer code.

The program then prints the contents of the array. Because the contents of NUM_ARRAY are of the SQL data type NUMBER, the elements of my_array are of the type, BigDecimal. Before you can use the elements, they must first be cast to BigDecimal. In the for loop, the individual values of the array are cast to BigDecimal and printed to standard output.

```
stmt.execute ("CREATE TYPE num_varray AS VARRAY(10) OF NUMBER(12, 2)");
stmt.execute ("CREATE TABLE varray_table (col1 num_varray)");
stmt.execute ("INSERT INTO varray_table VALUES (num_varray(100, 200))");
ResultSet rs = stmt.executeQuery("SELECT * FROM varray_table");
ARRAY my_array = ((OracleResultSet)rs).getARRAY(1);
// return the SQL type names, integer codes,
// and lengths of the columns
System.out.println ("Array is of type " + array.getSQLTypeName());
System.out.println ("Array element is of type code " + array.getBaseType());
System.out.println ("Array is of length " + array.length());
// get Array elements
BigDecimal[] values = (BigDecimal[]) my_array.getArray();
for (int i=0; i<values.length; i++)
{
BigDecimal out value = (BigDecimal) values[i];
```



```
System.out.println(">> index " + i + " = " + out_value.intValue());
```

Note that if you use getResultSet to obtain the array, then you must would first get the result set object, and then use the next method to iterate through it. Notice the use of the parameter indexes in the getInt method to retrieve the element index and the element value.

```
ResultSet rset = my_array.getResultSet();
while (rset.next())
{
    // The first column contains the element index and the
    // second column contains the element value
    System.out.println(">> index " + rset.getInt(1)+" = " + rset.getInt(2));
}
```

17.4.2.7 About Accessing Multilevel Collection Elements

The oracle.sql.ARRAY class provides three methods, which are overloaded, to access collection elements. The JDBC drivers extend these methods to support multilevel collections. These methods are:

• getArray method

}

- getOracleArray method
- getResultSet method

The getArray method returns a Java array that holds the collection elements. The array element type is determined by the collection element type and the JDBC default conversion matrix.

For example, the getArray method returns a java.math.BigDecimal array for collection of SQL NUMBER. The getOracleArray method returns a Datum array that holds the collection elements in Datum format. For multilevel collections, the getArray and getOracleArray methods both return a Java array of oracle.sql.ARRAY elements.

The getResultSet method returns a ResultSet object that wraps the multilevel collection elements. For multilevel collections, the JDBC applications use the getObject, getArRAY, or getArray method of the ResultSet class to access the collection elements as instances of oracle.sql.ARRAY.

The following code shows how to use the getOracleArray, getArray, and getResultSet methods:

```
Connection conn = ...; // make a JDBC connection
Statement stmt = conn.createStatement ();
ResultSet rset = stmt.executeQuery ("select col2 from tab2 where idx=1");
while (rset.next())
{
    ARRAY varray3 = (ARRAY) rset.getObject (1);
    Object varrayElems = varray3.getArray (1);
// access array elements of "varray3"
    Datum[] varray3Elems = (Datum[]) varrayElems;
for (int i=0; i<varray3Elems.length; i++)
    {
        ARRAY varray2 = (ARRAY) varray3Elems[i];
```



```
Datum[] varray2Elems = varray2.getOracleArray();
    // access array elements of "varray2"
    for (int j=0; j<varray2Elems.length; j++)</pre>
    {
      ARRAY varray1 = (ARRAY) varray2Elems[j];
      ResultSet varray1Elems = varray1.getResultSet();
      // access array elements of "varray1"
      while (varray1Elems.next())
        System.out.println ("idx="+varraylElems.getInt(1)+"
          value="+varray1Elems.getInt(2));
    }
  }
}
rset.close ();
stmt.close ();
conn.close ();
```

17.4.3 Passing Arrays to Statement Objects

This section discusses how to pass arrays to prepared statement objects or callable statement objects.

Passing an Array to a Prepared Statement

Pass an array to a prepared statement as follows.

Note: you can use arrays as either IN or OUT bind variables.

1. Define the array that you want to pass to the prepared statement as an oracle.sql.ARRAY object.

```
ARRAY array = oracle.jdbc.OracleConnection.createARRAY(sql type name, elements);
```

sql_type_name is a Java string specifying the user-defined SQL type name of the array and elements is a java.lang.Object containing a Java array of the elements.

- 2. Create a java.sql.PreparedStatement object containing the SQL statement to be run.
- 3. Cast your prepared statement to OraclePreparedStatement, and use setARRAY to pass the array to the prepared statement.

(OraclePreparedStatement)stmt.setARRAY(parameterIndex, array);

parameterIndex is the parameter index and *array* is the oracle.sql.ARRAY object you constructed previously.

4. Run the prepared statement.

Passing an Array to a Callable Statement

To retrieve a collection as an OUT parameter in PL/SQL blocks, perform the following to register the bind type for your OUT parameter.



1. Cast your callable statement to OracleCallableStatement, as follows:

```
OracleCallableStatement ocs = (OracleCallableStatement)conn.prepareCall("{?
= call func()}");
```

 Register the OUT parameter with the following form of the registerOutParameter method:

ocs.registerOutParameter
 (int param_index, int sql_type, string sql_type_name);

param_index is the parameter index, *sql_type* is the SQL type code, and *sql_type_name* is the name of the array type. In this case, the *sql_type* is OracleTypes.ARRAY.

3. Run the call, as follows:

ocs.execute();

4. Get the value, as follows:

oracle.sql.ARRAY array = ocs.getARRAY(1);

17.5 Using a Type Map to Map Array Elements

If your array contains Oracle objects, then you can use a type map to associate the objects in the array with the corresponding Java class. If you do not specify a type map, or if the type map does not contain an entry for a particular Oracle object, then each element is returned as an oracle.jdbc.OracleStruct object.

If you want the type map to determine the mapping between the Oracle objects in the array and their associated Java classes, then you must add an appropriate entry to the map.

The following example illustrates how you can use a type map to map the elements of an array to a custom Java object class. In this case, the array is a nested table. The example begins by defining an EMPLOYEE object that has a name attribute and employee number attribute. EMPLOYEE_LIST is a nested table type of EMPLOYEE objects. Then an EMPLOYEE_TABLE is created to store the names of departments within a corporation and the employees associated with each department. In the EMPLOYEE_TABLE, the employees are stored in the form of EMPLOYEE_LIST tables.

```
stmt.execute("CREATE TYPE EMPLOYEE AS OBJECT
                              (EmpName VARCHAR2(50),EmpNo INTEGER))");
stmt.execute("CREATE TYPE EMPLOYEE_LIST AS TABLE OF EMPLOYEE");
stmt.execute("CREATE TABLE EMPLOYEE_TABLE (DeptName VARCHAR2(20),
        Employees EMPLOYEE_LIST) NESTED TABLE Employees STORE AS ntable1");
stmt.execute("INSERT INTO EMPLOYEE_TABLE VALUES ("SALES", EMPLOYEE_LIST
                         (EMPLOYEE('Susan Smith', 123), EMPLOYEE('Lee Brown', 124)))");
```

If you want to retrieve all the employees belonging to the SALES department into an array of instances of the custom object class EmployeeObj, then you must add an entry to the type map to specify mapping between the EMPLOYEE SQL type and the EmployeeObj custom object class.

To do this, first create your statement and result set objects, then select the EMPLOYEE_LIST associated with the SALES department into the result set. Cast the



result set to OracleResultSet so you can use the getARRAY method to retrieve the EMPLOYEE LIST into an ARRAY object (employeeArray in the following example).

The EmployeeObj custom object class in this example implements the SQLData interface.

```
Statement s = conn.createStatement();
OracleResultSet rs = (OracleResultSet)s.executeQuery
    ("SELECT Employees FROM employee_table WHERE DeptName = 'SALES'");
```

```
// get the array object
ARRAY employeeArray = ((OracleResultSet)rs).getARRAY(1);
```

Now that you have the EMPLOYEE_LIST object, get the existing type map and add an entry that maps the EMPLOYEE SQL type to the EmployeeObj Java type.

```
// add type map entry to map SQL type
// "EMPLOYEE" to Java type "EmployeeObj"
Map map = conn.getTypeMap();
map.put("EMPLOYEE", Class.forName("EmployeeObj"));
```

Next, retrieve the SQL EMPLOYEE objects from the EMPLOYEE_LIST. To do this, call the getArray method of the employeeArray array object. This method returns an array of objects. The getArray method returns the EMPLOYEE objects into the employees object array.

```
// Retrieve array elements
Object[] employees = (Object[]) employeeArray.getArray();
```

Finally, create a loop to assign each of the EMPLOYEE SQL objects to the EmployeeObj Java object emp.

```
// Each array element is mapped to EmployeeObj object.
for (int i=0; i<employees.length; i++)
{
    EmployeeObj emp = (EmployeeObj) employees[i];
    ...
}</pre>
```



18 Result Set

Standard Java Database Connectivity (JDBC) features in Java Development Kit (JDK) include enhancements to result set functionality, such as processing forward or backward, positioning relatively or absolutely, seeing changes to the database made internally or externally, and updating result set data and then copying the changes to the database.

This chapter discusses the following topics:

- Oracle JDBC Implementation Overview for Result Set Support
- Resultset Limitations and Downgrade Rules
- About Avoiding Update Conflicts
- Row Fetch Size
- About Refetching Rows
- About Viewing Database Changes Made Internally and Externally

18.1 Oracle JDBC Implementation Overview for Result Set Support

This section discusses key aspects of the Oracle JDBC implementation of result set support for scrollability, through use of a client-side cache, and for updatability, through use of ROWIDS.

It is permissible for customers to implement their own client-side caching mechanism, and Oracle provides an interface to use in doing so.

Oracle JDBC Implementation for Result Set Scrollability

Because the underlying server does *not* support scrollable cursors, Oracle JDBC must implement scrollability in a separate layer.

It is important to be aware that this is accomplished by using a client-side memory cache to store rows of a scrollable result set.

Note:

Because all rows of any scrollable result set are stored in the client-side cache, a situation, where the result set contains many rows, many columns, or very large columns, might cause the client-side Java Virtual Machine (JVM) to fail. Do not specify scrollability for a large result set.



Oracle JDBC Implementation for Result Set Updatability

To support updatability, Oracle JDBC uses ROWID to uniquely identify database rows that appear in a result set. For every query into an updatable result set, Oracle JDBC driver automatically retrieves the ROWID along with the columns you select.

Note:

Client-side caching is not required by updatability in and of itself. In particular, a forward-only updatable result set will not require a client-side cache.

18.2 Resultset Limitations and Downgrade Rules

Some types of result sets are not feasible for certain kinds of queries. If you specify an unfeasible result set type or concurrency type for the query you run, then the JDBC driver follows a set of rules to determine the best feasible types to use instead.

The actual result set type and concurrency type are determined when the statement is run, with the driver issuing a SQLWarning on the statement object if the desired result set type or concurrency type is not feasible. The SQLWarning object will contain the reason why the requested type was not feasible. Check for warnings to verify whether you received the type of result set that you requested.

Result Set Limitations

The following limitations are placed on queries for enhanced result sets. Failure to follow these guidelines results in the JDBC driver choosing an alternative result set type or concurrency type.

To produce an updatable result set:

• A query can select from only a single table and cannot contain any join operations.

In addition, for inserts to be feasible, the query must select all non-nullable columns and all columns that do not have a default value.

• A query cannot use SELECT * .

However, there is a workaround for this.

• A query must select table columns only.

It cannot select derived columns or aggregates, such as the $\ensuremath{\texttt{SUM}}$ or $\ensuremath{\texttt{MAX}}$ of a set of columns.

To produce a scroll-sensitive result set:

• A query cannot use SELECT * .

However, there is a workaround for this.

• A query can select from only a single table.

Scrollable and updatable result sets cannot have any column as Stream. When the server has to fetch a Stream column, it reduces the fetch size to one and blocks all



columns following the Stream column until the Stream column is read. As a result, columns cannot be fetched in bulk and scrolled through.

Workaround

As a workaround for the SELECT * limitation, you can use table aliases, as shown in the following example:

SELECT t.* FROM TABLE t ...

Note:

There is a simple way to determine if your query will probably produce a scrollsensitive or updatable result set: If you can legally add a ROWID column to the query list, then the query is probably suitable for either a scroll-sensitive or an updatable result set.

Result Set Downgrade Rules

If the specified result set type or concurrency type is not feasible, then Oracle JDBC driver uses the following rules in choosing alternate types:

- If the specified result set type is TYPE_SCROLL_SENSITIVE, but the JDBC driver cannot fulfill that request, then the driver attempts a downgrade to TYPE SCROLL INSENSITIVE.
- If the specified or downgraded result set type is TYPE_SCROLL_INSENSITIVE, but the JDBC driver cannot fulfill that request, then the driver attempts a downgrade to TYPE FORWARD ONLY.
- If the specified concurrency type is CONCUR_UPDATABLE, but the JDBC driver cannot fulfill that request, then the JDBC driver attempts a downgrade to CONCUR READ ONLY.

Note:

Any manipulations of the result set type and concurrency type by the JDBC driver are independent of each other.

Verifying Result Set Type and Concurrency Type

After a query has been run, you can verify the result set type and concurrency type that the JDBC driver actually used, by calling methods on the result set object.

int getType() throws SQLException

This method returns an int value for the result set type used for the query. ResultSet.TYPE_FORWARD_ONLY, ResultSet.TYPE_SCROLL_SENSITIVE, or ResultSet.TYPE_SCROLL_INSENSITIVE are the possible values.

• int getConcurrency() throws SQLException

This method returns an int value for the concurrency type used for the query. ResultSet.CONCUR READ ONLY or ResultSet.CONCUR UPDATABLE are the possible values.



18.3 About Avoiding Update Conflicts

It is important to be aware of the following facts regarding updatable result sets with the JDBC drivers:

- The drivers do not enforce write locks for an updatable result set.
- The drivers do not check for conflicts with a result set DELETE or UPDATE operation.

A conflict will occur if you try to perform a DELETE or UPDATE operation on a row updated by another committed transaction.

Oracle JDBC drivers use the ROWID to uniquely identify a row in a database table. As long as the ROWID is valid when a driver tries to send an UPDATE or DELETE operation to the database, the operation will be run.

The driver will not report any changes made by another committed transaction. Any conflicts are silently ignored and your changes will overwrite the previous changes.

To avoid such conflicts, use the Oracle FOR UPDATE feature when running the query that produces the result set. This will avoid conflicts, but will also prevent simultaneous access to the data. Only a single write lock can be held concurrently on a data item.

18.4 Row Fetch Size

By default, when Oracle JDBC runs a query, it retrieves a result set of 10 rows at a time from the database cursor. This is the default Oracle row fetch size value. You can change the number of rows retrieved with each trip to the database cursor by changing the row fetch size value.

Standard JDBC also enables you to specify the number of rows fetched with each database round-trip for a query, and this number is referred to as the fetch size. In Oracle JDBC, the row-prefetch value is used as the default fetch size in a statement object. Setting the fetch size overrides the row-prefetch setting and affects subsequent queries run through that statement object.

Fetch size is also used in a result set. When the statement object run a query, the fetch size of the statement object is passed to the result set object produced by the query. However, you can also set the fetch size in the result set object to override the statement fetch size that was passed to it.

Note:

Changes made to the fetch size of a statement object after a result set is produced will have no affect on that result set.

The result set fetch size, either set explicitly, or by default equal to the statement fetch size that was passed to it, determines the number of rows that are retrieved in any subsequent trips to the database for that result set. This includes any trips that are still required to complete the original query, as well as any refetching of data into the result set. Data can be refetched, either explicitly or implicitly, to update a scroll-sensitive or scroll-insensitive/updatable result set.



18.4.1 Setting the Fetch Size

The following methods are available in all Statement, PreparedStatement, CallableStatement, and ResultSet objects for setting and getting the fetch size:

- void setFetchSize(int rows) throws SQLException
- int getFetchSize() throws SQLException

To set the fetch size for a query, call <code>setFetchSize</code> on the statement object prior to running the query. If you set the fetch size to N, then N rows are fetched with each trip to the database.

After you have run the query, you can call setFetchSize on the result set object to override the statement object fetch size that was passed to it. This will affect any subsequent trips to the database to get more rows for the original query, as well as affecting any later refetching of rows.

18.4.2 Presetting the Fetch Direction

The standard JDBC enables to pre-specify the direction, known as the fetch direction, for use in processing a result set. This allows the JDBC driver to optimize its processing. The following result set methods are specified:

- void setFetchDirection(int direction) throws SQLException
- int getFetchDirection() throws SQLException

Oracle JDBC drivers support only the forward preset value, which you can specify by entering the ResultSet.FETCH FORWARD static constant value.

The values ResultSet.FETCH_REVERSE and ResultSet.FETCH_UNKNOWN are not supported. Attempting to specify them causes a SQL warning, and the settings are ignored.

18.5 About Refetching Rows

The result set refreshRow method is supported for some types of result sets for refetching data. This consists of going back to the database to re-obtain the database rows that correspond to *n* rows in the result set, starting with the current row, where *n* is the fetch size. This lets you see the latest updates to the database that were made outside of your result set, subject to the isolation level of the enclosing transaction.

Because refetching re-obtains only rows that correspond to rows already in your result set, it does nothing about rows that have been inserted or deleted in the database since the original query. It ignores rows that have been inserted, and rows will remain in your result set even after the corresponding rows have been deleted from the database. When there is an attempt to refetch a row that has been deleted in the database, the corresponding row in the result set will maintain its original values.



If you declare a TYPE_SCROLL_SENSITIVE Result Set based on a query with certain criteria and then externally update the row so that the column values no longer match the query criteria, the driver behaves as if the row has been deleted from the database and the row is not retrieved by the query issued. So, you do not see the updates to the particular row when you call the refreshRow method.

Following is the signature of the refreshRow method:

void refreshRow() throws SQLException

You must be at a valid current row when you call this method, not outside the row bounds and not at the insert-row.

The refreshRow method is supported for the following result set categories:

- scroll-sensitive/read-only
- scroll-sensitive/updatable
- scroll-insensitive/updatable

Note:

Scroll-sensitive result set functionality is implemented through implicit calls to refreshRow.

18.6 About Viewing Database Changes Made Internally and Externally

This section discusses the ability of a result set to view the following:

- Own changes of the result set, referred to as internal changes
- Changes made from elsewhere, either from your own transaction outside the result set, or from other committed transactions, referred to as external changes

Note:

External changes are referred to as other's changes in the standard JDBC specification.

This section covers the following topics:

- Visibility versus Detection of External Changes
- Summary of Visibility of Internal and External Changes



Oracle Implementation of Scroll-Sensitive Result Sets

18.6.1 Visibility versus Detection of External Changes

Regarding the changes made to an underlying database by external sources, there are two similar but distinct concepts with respect to visibility of the changes from your local result set:

- Visibility of changes
- Detection of changes

A "visible" change means that when you look at a row in the result set, you can see new data values from changes made by external sources, to the corresponding row in the database.

A "detected" change, however, means that the result set is aware that this is a new value since the result set was first populated.

Even when an Oracle result set sees new data, as with an external UPDATE in a scrollsensitive result set, it has no awareness that this data has changed since the result set was populated. Such changes are not detected.

18.6.2 Summary of Visibility of Internal and External Changes

Table 18-1 summarizes how a result set object in the Oracle JDBC implementation can see changes made internally through the result set itself, and changes made externally to the underlying database from elsewhere in your transaction or from other committed transactions.

Result Set Type	Can See Internal DELETE?	Can See Internal UPDATE?	Can See Internal INSERT?	Can See External DELETE?	Can See External UPDATE?	Can See External INSERT?
forward-only	no	yes	no	no	no	no
scroll-sensitive	yes	yes	no	no	yes	no
scroll-insensitive	yes	yes	no	no	no	no

Table 18-1 Visibility of Internal and External Changes for Oracle JDBC

Note:

- Remember that explicit use of the refreshRow method, is distinct from the concept of visibility of external changes.
- Remember that even when external changes are visible, as with UPDATE operations underlying a scroll-sensitive result set, they are not detected. The result set rowDeleted, rowUpdated, and rowInserted methods always return false.



18.6.3 Oracle Implementation of Scroll-Sensitive Result Sets

The Oracle implementation of scroll-sensitive result sets involves the concept of a window, with a window size that is based on the fetch size. The window size affects how often rows are updated in the result set.

Once you establish a current row by moving to a specified row, the window consists of the *n* rows in the result set starting with that row, where *n* is the fetch size being used by the result set. Note that there is no current row, and therefore no window, when a result set is first created. The default position is before the first row, which is not a valid current row.

As you move from row to row, the window remains unchanged as long as the current row stays within that window. However, once you move to a new current row outside the window, you redefine the window to be the N rows starting with the new current row.

Whenever the window is redefined, the N rows in the database corresponding to the rows in the new window are automatically refetched through an implicit call to the refreshRow method, thereby updating the data throughout the new window.

So external updates are not instantaneously visible in a scroll-sensitive result set. They are only visible after the automatic refetches just described.

Note:

This kind of refetching is not a highly efficient or optimized methodology and it has significant performance concerns. Consider carefully before using scroll-sensitive result sets as currently implemented. There is also a significant trade-off between sensitivity and performance. The most sensitive result set is one with a fetch size of 1, which would result in the new current row being refetched every time you move between rows. However, this would have a significant impact on the performance of your application.



19 JDBC RowSets

This chapter contains the following sections:

- Overview of JDBC RowSets
- About CachedRowSet
- About JdbcRowSet
- About WebRowSet
- About FilteredRowSet
- About JoinRowSet

19.1 Overview of JDBC RowSets

A RowSet is an object that encapsulates a set of rows from either java Database Connectivity (JDBC) result sets or tabular data sources. RowSets support component-based development models like JavaBeans, with a standard set of properties and an event notification mechanism.

RowSets were introduced in JDBC 2.0 through the optional packages. However, the implementation of RowSets was standardized in the JDBC RowSet Implementations Specification (JSR-114), which is available as non-optional package since Java Platform, Standard Edition (Java SE) 5.0. Java SE 6.0 RowSets contain more APIs supporting features like RowId, National Language Charactersets, and so on. The Java SE Javadocs provide information about the standard interfaces and base classes for JDBC RowSet implementations.

The JSR-114 specification includes implementation details for five types of RowSet:

- CachedRowSet
- JdbcRowSet
- WebRowSet
- FilteredRowSet
- JoinRowSet

Oracle JDBC supports all five types of RowSets through the interfaces and classes present in the oracle.jdbc.rowset package. Since Oracle Database 11g Release 1, RowSets support has been added in the server-side drivers. Therefore, starting from Oracle Database 11g Release 1, RowSets support is uniform across all Oracle JDBC driver types. The standard Oracle JDBC Java Archive (JAR) files, for example, ojdbc6.jar and ojdbc7.jar contain the oracle.jdbc.rowset package.



- The other JAR files with different file suffix names, for example, ojdbc6_g.jar, ojdbc6dms.jar, and so on also contain the oracle.jdbc.rowset package.
- In Oracle Database 10g release 2, the implementation classes were packaged in the ojdbc14.jar file.
- Prior to Oracle Database 10g release 2, the implementation classes were packaged in the ocrs12.jar file.
- Prior to Oracle Database 11g Release 1, RowSets support was not available in the server-side drivers.

Note:

In Oracle Database 10g release 2 (10.2), this package is included in the standard Oracle JDBC JAR files: classes12.jar, ojdbc5.jar, and ojdbc6.jar. Prior to Oracle Database 10g release 2 (10.2), the row set implementation classes were packaged in the ocrs12.jar file.

To use the Oracle RowSet implementations, you need to import either the entire oracle.jdbc.rowset package or specific classes and interfaces from the package for the required RowSet type. For client-side usage, you also need to include the standard Oracle JAR files like ojdbc6.jar or ojdbc7.jar in the CLASSPATH environment variable.

This section covers the following topics:

- RowSet Properties
- Events and Event Listeners
- Command Parameters and Command Execution
- About Traversing RowSets

19.1.1 RowSet Properties

The javax.sql.RowSet interface provides a set of JavaBeans properties that can be altered to access the data in the data source through a single interface. Example of properties are connection string, user name, password, type of connection, and the query string.

🖍 See Also:

The Java 2 Platform, Standard Edition (J2SE) Javadoc for a complete list of properties and property descriptions at http://docs.oracle.com/javase/ 1.5.0/docs/api/javax/sql/RowSet.html



The interface provides standard accessor methods for setting and retrieving the property values. The following code illustrates setting some of the RowSet properties:

```
...
rowset.setUrl("jdbc:oracle:oci:@");
rowset.setUsername("HR");
rowset.setPassword("hr");
rowset.setCommand("SELECT employee_id, first_name, last_name, salary FROM employees");
...
```

In this example, the URL, user name, password, and SQL query are set as the RowSet properties to retrieve the employee number, employee name, and salary of all the employees into the RowSet object.

19.1.2 Events and Event Listeners

RowSets support JavaBeans events. The following types of events are supported by the RowSet interface:

cursorMoved

This event is generated whenever there is a cursor movement. For example, when the next or previous method is called.

rowChanged

This event is generated when a row is inserted, updated, or deleted from the RowSet.

rowSetChanged

This event is generated when the whole RowSet is created or changed. For example, when the execute method is called.

An application component can implement a RowSet listener to listen to these RowSet events and perform desired operations when the event occurs. Application components, which are interested in these events, must implement the standard javax.sql.RowSetListener interface and register such listener objects with a RowSet object. A listener can be registered using the RowSet.addRowSetListener method and unregistered using the RowSet.removeRowSetListener method. Multiple listeners can be registered with the same RowSet object.

The following code illustrates the registration of a RowSet listener:

```
MyRowSetListener rowsetListener = new MyRowSetListener ();
// adding a rowset listener
rowset.addRowSetListener (rowsetListener);
...
```

The following code illustrates a listener implementation:

```
public class MyRowSetListener implements RowSetListener
{
   public void cursorMoved(RowSetEvent event)
   {
      // action on cursor movement
   }
   public void rowChanged(RowSetEvent event)
   {
      // action on change of row
```



```
}
public void rowSetChanged(RowSetEvent event)
{
    // action on changing of rowset
}
}// end of class MyRowSetListener
```

Applications that need to handle only selected events can implement only the required event handling methods by using the

oracle.jdbc.rowset.OracleRowSetListenerAdapter class, which is an abstract class with empty implementation for all the event handling methods. In the following code, only the rowSetChanged event is handled, while the remaining events are not handled by the application:

```
...
rowset.addRowSetListener(new oracle.jdbc.rowset.OracleRowSetListenerAdapter ()
{
    public void rowSetChanged(RowSetEvent event)
    {
        // your action for rowSetChanged
    }
    }
);
...
```

19.1.3 Command Parameters and Command Execution

The command property of a RowSet object typically represents a SQL query string, which when processed would populate the RowSet object with actual data. Like in regular JDBC processing, this query string can take input or bind parameters. The javax.sql.RowSet interface also provides methods for setting input parameters to this SQL query. After the required input parameters are set, the SQL query can be processed to populate the RowSet object with data from the underlying data source. The following code illustrates this simple sequence:

```
rowset.setCommand("SELECT first_name, last_name, salary FROM employees WHERE
employee_id = ?");
    // setting the employee number input parameter for employee named "Douglas"
    rowset.setInt(1, 199);
    rowset.execute();
    ...
```

In the preceding example, the employee number 199 is set as the input or bind parameter for the SQL query specified in the command property of the RowSet object. When the SQL query is processed, the RowSet object is filled with the employee name and salary information of the employee whose employee number is 199.

19.1.4 About Traversing RowSets

The javax.sql.RowSet interface extends the java.sql.ResultSet interface. The RowSet interface, therefore, provides cursor movement and positioning methods, which are inherited from the ResultSet interface, for traversing through data in a RowSet object. Some of the inherited methods are absolute, beforeFirst, afterLast, next, and previous.



The RowSet interface can be used just like a ResultSet interface for retrieving and updating data. The RowSet interface provides an optional way to implement a scrollable and updatable result set. All the fields and methods provided by the ResultSet interface are implemented in RowSet.

Note:

The Oracle implementation of ResultSet provides the scrollable and updatable properties of the java.sql.ResultSet interface.

The following code illustrates how to scroll through a RowSet:

```
/**
 * Scrolling forward, and printing the empno in
 * the order in which it was fetched.
 */
...
rowset.setCommand("SELECT empno, ename, sal FROM emp");
rowset.execute();
...
// going to the first row of the rowset
rowset.beforeFirst ();
while (rowset.next ())
 System.out.println ("empno: " +rowset.getInt (1));
```

In the preceding code, the cursor position is initialized to the position before the first row of the RowSet by the beforeFirst method. The rows are retrieved in forward direction using the next method.

The following code illustrates how to scroll through a RowSet in the reverse direction:

```
/**
 * Scrolling backward, and printing the empno in
 * the reverse order as it was fetched.
 */
//going to the last row of the rowset
rowset.afterLast ();
while (rowset.previous ())
 System.out.println ("empno: " +rowset.getInt (1));
```

In the preceding code, the cursor position is initialized to the position after the last row of the RowSet. The rows are retrieved in reverse direction using the previous method of RowSet.

Inserting, updating, and deleting rows are supported by the Row Set feature as they are in the Result Set feature. In order to make the Row Set updatable, you must call the setReadOnly(false) and acceptChanges methods.

The following code illustrates the insertion of a row at the fifth position of a Row Set:

```
/**
    * Make rowset updatable
    */
rowset.setReadOnly (false);
/**
    * Inserting a row in the 5th position of the rowset.
    */
```



```
// moving the cursor to the 5th position in the rowset
if (rowset.absolute(5))
{
    rowset.moveToInsertRow ();
    rowset.updateInt (1, 193);
    rowset.updateString (2, "Smith");
    rowset.updateInt (3, 7200);
    // inserting a row in the rowset
    rowset.insertRow ();
    // Synchronizing the data in RowSet with that in the database.
    rowset.acceptChanges ();
}
...
```

In the preceding code, a call to the absolute method with a parameter 5 takes the cursor to the fifth position of the RowSet and a call to the moveToInsertRow method creates a place for the insertion of a new row into the RowSet. The updateXXX methods are used to update the newly created row. When all the columns of the row are updated, the insertRow is called to update the RowSet. The changes are committed through acceptChanges method.

19.2 About CachedRowSet

A CachedRowSet is a RowSet in which the rows are cached and the RowSet is disconnected, that is, it does not maintain an active connection to the database. The oracle.jdbc.rowset.OracleCachedRowSet class is the Oracle implementation of CachedRowSet. It can interoperate with the standard reference implementation. The OracleCachedRowSet class in the ojdbc6.jar and ojdbc7.jar files implements the standard JSR-114 interface javax.sql.rowset.CachedRowSet.

In the following code, an OracleCachedRowSet object is created and the connection URL, user name, password, and the SQL query for the RowSet object is set as properties. The RowSet object is populated using the execute method. After the execute method has been processed, the RowSet object can be used as a java.sql.ResultSet object to retrieve, scroll, insert, delete, or update data.

```
. . .
RowSet rowset = new OracleCachedRowSet();
rowset.setUrl("jdbc:oracle:oci:@");
rowset.setUsername("HR");
rowset.setPassword("hr");
rowset.setCommand("SELECT employee id, first name, last name, salary FROM
emplovees");
rowset.execute();
while (rowset.next ())
{
  System.out.println("employee id: " +rowset.getInt (1));
  System.out.println("first_name: " +rowset.getString (2));
  System.out.println("last name: " +rowset.getString (3));
  System.out.println("sal: " +rowset.getInt (4));
}
. . .
```

To populate a CachedRowSet object with a query, complete the following steps:

1. Instantiate OracleCachedRowSet.

- 2. Set the Url, which is the connection URL, Username, Password, and Command, which is the query string, properties for the RowSet object. You can also set the connection type, but it is optional.
- 3. Call the execute method to populate the CachedRowSet object. Calling execute runs the query set as a property on this RowSet.

```
OracleCachedRowSet rowset = new OracleCachedRowSet ();
rowset.setUrl ("jdbc:oracle:oci:@");
rowset.setUsername ("HR");
rowset.setPassword ("hr");
rowset.setCommand ("SELECT employee_id, first_name, last_name, salary FROM
employees");
rowset.execute ();
```

A CachedRowSet object can be populated with an existing ResultSet object, using the populate method. To do so, complete the following steps:

- 1. Instantiate OracleCachedRowSet.
- 2. Pass the already available ResultSet object to the populate method to populate the RowSet object.

```
// Executing a query to get the ResultSet object.
ResultSet rset = pstmt.executeQuery ();
OracleCachedRowSet rowset = new OracleCachedRowSet ();
// the obtained ResultSet object is passed to the populate method
// to populate the data in the rowset object.
rowset.populate (rset);
```

In the preceding example, a ResultSet object is obtained by running a query and the retrieved ResultSet object is passed to the populate method of the CachedRowSet object to populate the contents of the result set into the CachedRowSet.

Note:

Connection properties, like transaction isolation or the concurrency mode of the result set, and the bind properties cannot be set in the case where a pre-existent ResultSet object is used to populate the CachedRowSet object, because the connection or result set on which the property applies would have already been created.

The following code illustrates how an OracleCachedRowSet object is serialized to a file and then retrieved:

```
// writing the serialized OracleCachedRowSet object
{
   FileOutputStream fileOutputStream = new FileOutputStream("emp_tab.dmp");
   ObjectOutputStream ostream = new ObjectOutputStream(fileOutputStream);
   ostream.writeObject(rowset);
   ostream.close();
   fileOutputStream.close();
}
```



```
// reading the serialized OracleCachedRowSet object
{
   FileInputStream fileInputStream = new FileInputStream("emp_tab.dmp");
   ObjectInputStream istream = new ObjectInputStream(fileInputStream);
   RowSet rowset1 = (RowSet) istream.readObject();
   istream.close();
   fileInputStream.close();
}
```

In the preceding code, a FileOutputStream object is opened for an emp_tab.dmp file, and the populated OracleCachedRowSet object is written to the file using ObjectOutputStream. The serialized OracleCachedRowSet object is retrieved using the FileInputStream and ObjectInputStream objects.

OracleCachedRowSet takes care of the serialization of non-serializable form of data like InputStream, OutputStream, binary large objects (BLOBs), and character large objects (CLOBs). OracleCachedRowSets also implements metadata of its own, which could be obtained without any extra server round-trip. The following code illustrates how you can obtain metadata for the RowSet:

```
...
ResultSetMetaData metaData = rowset.getMetaData();
int maxCol = metaData.getColumnCount();
for (int i = 1; i <= maxCol; ++i)
    System.out.println("Column (" + i +") " + metaData.getColumnName(i));
...</pre>
```

Because the OracleCachedRowSet class is serializable, it can be passed across a network or between Java Virtual Machines (JVMs), as done in Remote Method Invocation (RMI). Once the OracleCachedRowSet class is populated, it can move around any JVM, or any environment that does not have JDBC drivers. Committing the data in the RowSet requires the presence of JDBC drivers.

The complete process of retrieving the data and populating it in the OracleCachedRowSet class is performed on the server and the populated RowSet is passed on to the client using suitable architectures like RMI or Enterprise Java Beans (EJB). The client would be able to perform all the operations like retrieving, scrolling, inserting, updating, and deleting on the RowSet without any connection to the database. Whenever data is committed to the database, the acceptChanges method is called, which synchronizes the data in the RowSet to that in the database. This method makes use of JDBC drivers, which require the JVM environment to contain JDBC implementation. This architecture would be suitable for systems involving a Thin client like a Personal Digital Assistant (PDA).

After populating the CachedRowSet object, it can be used as a ResultSet object or any other object, which can be passed over the network using RMI or any other suitable architecture.

Some of the other key-features of CachedRowSet are the following:

- Cloning a RowSet
- Creating a copy of a RowSet
- Creating a shared copy of a RowSet



CachedRowSet Constraints

All the constraints that apply to an updatable result set are applicable here, except serialization, because <code>OracleCachedRowSet</code> is serializable. The SQL query has the following constraints:

- · References only a single table in the database
- Contains no join operations
- Selects the primary key of the table it references

In addition, a SQL query should also satisfy the following conditions, if new rows are to be inserted:

- Selects all non-nullable columns in the underlying table
- Selects all columns that do not have a default value

Note:

The CachedRowSet cannot hold a large quantity of data, because all the data is cached in memory. Oracle, therefore, recommends against using OracleCachedRowSet with queries that could potentially return a large volume of data.

Connection properties like, transaction isolation and concurrency mode of the result set, cannot be set after populating the RowSet, because the properties cannot be applied to the connection after retrieving the data from the same.

19.3 About JdbcRowSet

A JdbcRowSet is a RowSet that wraps around a ResultSet object. It is a connected RowSet that provides JDBC interfaces in the form of a JavaBean interface. The Oracle implementation of JdbcRowSet is oracle.jdbc.rowset.OracleJDBCRowSet. The OracleJDBCRowSet class in ojdbc6.jar and ojdbc7.jar implements the standard JSR-114 interface javax.sql.rowset.JdbcRowSet.

Table 19-1 shows how the JdbcRowSet interface differs from CachedRowSet interface.

RowSet Type	Serializable	Connected to Database	Movable Across JVMs	Synchronization of data to database	Presence of JDBC Drivers
JDBC	Yes	Yes	No	No	Yes
Cached	Yes	No	Yes	Yes	No

Table 19-1 The JDBC and Cached Row Sets Compared

JdbcRowSet is a connected RowSet, which has a live connection to the database and all the calls on the JdbcRowSet are percolated to the mapping call in the JDBC connection, statement, or result set. A CachedRowSet does not have any connection to the database open.



JdbcRowSet requires the presence of JDBC drivers unlike a CachedRowSet, which does not require JDBC drivers during manipulation. However, both JdbcRowSet and CachedRowSet require JDBC drivers during population of the RowSet and while committing the changes of the RowSet.

The following code illustrates how a JdbcRowSet is used:

```
RowSet rowset = new OracleJDBCRowSet();
RowSet.setUrl("java:oracle:oci:@");
rowset.setUsername("HR");
rowset.setPassword("hr");
rowset.setCommand("SELECT empno, ename, sal FROM emp");
rowset.execute();
while (rowset.next())
{
   System.out.println("empno: " + rowset.getInt(1));
   System.out.println("ename: " + rowset.getString(2));
   System.out.println("sal: " + rowset.getInt(3));
}...
```

In the preceding example, the connection URL, user name, password, and SQL query are set as properties of the RowSet object, the SQL query is processed using the execute method, and the rows are retrieved and printed by traversing through the data populated in the RowSet object.

19.4 About WebRowSet

A WebRowSet is an extension to CachedRowSet. It represents a set of fetched rows or tabular data that can be passed between tiers and components in a way such that no active connections with the data source need to be maintained. The WebRowSet interface provides support for the production and consumption of result sets and their synchronization with the data source, both in Extensible Markup Language (XML) format and in disconnected fashion. This allows result sets to be shipped across tiers and over Internet protocols.

The Oracle implementation of WebRowSet is oracle.jdbc.rowset.OracleWebRowSet. This class, which is in the ojdbc6.jar and ojdbc7.jar files, implements the standard JSR-114 interface javax.sql.rowset.WebRowSet. This class also extends the oracle.jdbc.rowset.OracleCachedRowSet class. Besides the methods available in OracleCachedRowSet, the OracleWebRowSet class provides the following methods:

public OracleWebRowSet() throws SQLException

This is the constructor for creating an OracleWebRowSet object, which is initialized with the default values for an OracleCachedRowSet object, a default OracleWebRowSetXmlReader, and a default OracleWebRowSetXmlWriter.

• public void writeXml(java.io.Writer writer) throws SQLException public void writeXml(java.io.OutputStream ostream) throws SQLException

These methods write the <code>OracleWebRowSet</code> object to the supplied <code>Writer</code> or <code>OutputStream</code> object in the XML format that conforms to the JSR-114 XML schema. In addition to the RowSet data, the properties and metadata of the RowSet are written.



• public void writeXml(ResultSet rset, java.io.Writer writer) throws SQLException public void writeXml(ResultSet rset, java.io.OutputStream ostream) throws SQLException

These methods create an OracleWebRowSet object, populate it with the data in the given ResultSet object, and write it to the supplied Writer or OutputStream object in the XML format that conforms to the JSR-114 XML schema.

 public void readXml(java.io.Reader reader) throws SQLException public void readXml(java.io.InputStream istream) throws SQLException

These methods read the OracleWebRowSet object in the XML format according to its JSR-114 XML schema, using the supplied Reader or InsputStream object.

The Oracle WebRowSet implementation supports Java API for XML Processing (JAXP) 1.2. Both Simple API for XML (SAX) 2.0 and Document Object Model (DOM) JAXP-conforming XML parsers are supported. It follows the current JSR-114 W3C XML schema for WebRowSet.

Applications that use the readXml(...) methods should set one of the following two standard JAXP system properties before calling the methods:

• javax.xml.parsers.SAXParserFactory

This property is for a SAX parser.

javax.xml.parsers.DocumentBuilderFactory

This property is for a DOM parser.

The following code illustrates the use of OracleWebRowSet for both writing and reading in XML format:

```
import java.sql.*;
import java.io.*;
import oracle.jdbc.rowset.*;
String url = "jdbc:oracle:oci8:@";
Connection conn = DriverManager.getConnection(url, "HR", "hr");
Statement stmt = conn.createStatement();
ResultSet rset = stmt.executeQuery("select * from employees");
// Create an OracleWebRowSet object and populate it with the ResultSet object
OracleWebRowSet wset = new OracleWebRowSet();
wset.populate(rset);
try
{
  // Create a java.io.Writer object
 FileWriter out = new FileWriter("xml.out");
  // Now generate the XML and write it out
  wset.writeXml(out);
}
catch (IOException exc)
{
  System.out.println("Couldn't construct a FileWriter");
}
System.out.println("XML output file generated.");
```



```
// Create a new OracleWebRowSet for reading from XML input
OracleWebRowSet wset2 = new OracleWebRowSet();
// Use Oracle JAXP SAX parser
System.setProperty("javax.xml.parsers.SAXParserFactory","oracle.xml.jaxp.JXSAXPar
serFactory");
try
  // Use the preceding output file as input
  FileReader fr = new FileReader("xml.out");
  // Now read XML stream from the FileReader
  wset2.readXml(fr);
}
catch (IOException exc)
{
 System.out.println("Couldn't construct a FileReader");
}
. . .
```

The preceding code uses the Oracle SAX XML parser, which supports schema validation.

19.5 About FilteredRowSet

A FilteredRowSet is an extension to WebRowSet that provides programmatic support for filtering its content. This enables you to avoid the overhead of supplying a query and the processing involved. The Oracle implementation of FilteredRowSet is oracle.jdbc.rowset.OracleFilteredRowSet. The OracleFilteredRowSet class in the ojdbc7.jar files implements the standard JSR-114 interface javax.sql.rowset.FilteredRowSet.

The OracleFilteredRowSet class defines the following new methods:

• public Predicate getFilter();

This method returns a Predicate object that defines the filtering criteria active on the OracleFilteredRowSet object.

• public void setFilter(Predicate p) throws SQLException;

This method takes a Predicate object as a parameter. The Predicate object defines the filtering criteria to be applied on the OracleFilteredRowSet object. The methods throws a SQLException exception.



If you are using classes12.jar instead of ojdbc5.jar and ojdbc6.jar for FilteredRowSet features, then use OraclePredicate instead of Predicate. The oracle.jdbc.rowset.OraclePredicate interface is Oracle specific and is equivalent to Predicate. This interface is used when the JSR-114 packages are not available.

The predicate set on an OracleFilteredRowSet object defines a filtering criteria that is applied to all the rows in the object to obtain the set of visible rows. The predicate also defines the criteria for inserting, deleting, and modifying rows. The set filtering criteria acts as a gating mechanism for all views and updates to the OracleFilteredRowSet object. Any attempt to update the OracleFilteredRowSet object, which violates the filtering criteria, throws a SQLException exception.

The filtering criteria set on an OracleFilteredRowSet object can be modified by applying a new Predicate object. The new criteria is immediately applied on the object, and all further views and updates must adhere to this new criteria. A new filtering criteria can be applied only if there are no reference to the OracleFilteredRowSet object.

Rows that fall outside of the filtering criteria set on the object cannot be modified until the filtering criteria is removed or a new filtering criteria is applied. Also, only the rows that fall within the bounds of the filtering criteria will be synchronized with the data source, if an attempt is made to persist the object.

The following code example illustrates the use of OracleFilteredRowSet. Assume a table, test_table, with two NUMBER columns, col1 and col2. The code retrieves those rows from the table that have value of col1 between 50 and 100 and value of col2 between 100 and 200.

The predicate defining the filtering criteria is as follows:

```
public class PredicateImpl implements Predicate
{
 private int low[];
 private int high[];
 private int columnIndexes[];
  public PredicateImpl(int[] lo, int[] hi, int[] indexes)
  {
    low = lo;
   high = hi;
    columnIndexes = indexes;
  }
  public boolean evaluate (RowSet rs)
  {
   boolean result = true;
    for (int i = 0; i < columnIndexes.length; i++)</pre>
    {
     int columnValue = rs.getInt(columnIndexes[i]);
     if (columnValue < low[i] || columnValue > high[i])
        result = false;
    }
    return result;
```



```
// the other two evaluate(...) methods simply return true
}
```

The predicate defined in the preceding code is used for filtering content in an OracleFilteredRowSet object, as follows:

```
...
OracleFilteredRowSet ofrs = new OracleFilteredRowSet();
int low[] = {50, 100};
int high[] = {100, 200};
int indexes[] = {1, 2};
ofrs.setCommand("select col1, col2 from test_table");
// set other properties on ofrs like usr/pwd ...
...
ofrs.execute();
ofrs.setPredicate(new PredicateImpl(low, high, indexes));
// this will only get rows with col1 in (50,100) and col2 in (100,200)
while (ofrs.next()) {...}
```

19.6 About JoinRowSet

A JoinRowSet is an extension to WebRowSet that consists of related data from different RowSets. There is no standard way to establish a SQL JOIN between disconnected RowSets without connecting to the data source. A JoinRowSet addresses this issue. The Oracle implementation of JoinRowSet is the oracle.jdbc.rowset.OracleJoinRowSet class. This class, which is in the ojdbc7.jar files, implements the standard JSR-114 interface javax.sql.rowset.JoinRowSet.

Any number of RowSet objects, which implement the Joinable interface, can be added to a JoinRowSet object, provided they can be related in a SQL JOIN. All five types of RowSet support the Joinable interface. The Joinable interface provides methods for specifying the columns based on which the JOIN will be performed, that is, the match columns.

Note:

If you are using classes12.jar instead of ojdbc5.jar and ojdbc6.jar for JoinRowSet features, then use OracleJoinable instead of Joinable. The oracle.jdbc.rowset.OracleJoinable interface is Oracle-specific and is equivalent to Joinable. This interface is used when the JSR-114 packages are not available.

A match column can be specified in the following ways:

• Using the setMatchColumn method

This method is defined in the Joinable interface. It is the only method that can be used to set the match column before a RowSet object is added to a JoinRowSet object. This method can also be used to reset the match column at any time.



• Using the addRowSet method

This is an overloaded method in JoinRowSet. Four of the five implementations of this method take a match column as a parameter. These four methods can be used to set or reset a match column at the time a RowSet object is being added to a JoinRowSet object.

In addition to the inherited methods, OracleJoinRowSet provides the following methods:

```
public void addRowSet(Joinable joinable) throws SQLException;
public void addRowSet(RowSet rowSet, int i) throws SQLException;
public void addRowSet(RowSet rowSet, String s) throws SQLException;
public void addRowSet(RowSet arowSet[], int an[]) throws SQLException;
public void addRowSet(RowSet arowSet[], String as[]) throws SQLException;
```

These methods are used to add a RowSet object to the OracleJoinRowSet object. You can pass one or more RowSet objects to be added to the OracleJoinRowSet object. You can also pass names or indexes of one or more columns, which need to be set as match column.

public Collection getRowSets() throws SQLException;

This method retrieves the RowSet objects added to the OracleJoinRowSet object. The method returns a java.util.Collection object that contains the RowSet objects.

public String[] getRowSetNames() throws SQLException;

This method returns a String array containing the names of the RowSet objects that are added to the OracleJoinRowSet object.

```
    public boolean supportsCrossJoin();
    public boolean supportsFullJoin();
    public boolean supportsInnerJoin();
    public boolean supportsLeftOuterJoin();
    public boolean supportsRightOuterJoin();
```

These methods return a boolean value indicating whether the <code>OracleJoinRowSet</code> object supports the corresponding <code>JOIN</code> type.

• public void setJoinType(int i) throws SQLException;

This method is used to set the JOIN type on the OracleJoinRowSet object. It takes an integer constant as defined in the javax.sql.rowset.JoinRowSet interface that specifies the JOIN type.

public int getJoinType() throws SQLException;

This method returns an integer value that indicates the JOIN type set on the OracleJoinRowSet object. This method throws a SQLException exception.

public CachedRowSet toCachedRowSet() throws SQLException;

This method creates a CachedRowSet object containing the data in the OracleJoinRowSet object.

public String getWhereClause() throws SQLException;

This method returns a String containing the SQL-like description of the WHERE clause used in the OracleJoinRowSet object. This methods throws a SQLException exception.



The following code illustrates how <code>OracleJoinRowSet</code> is used to perform an inner join on two RowSets, whose data come from two different tables. The resulting RowSet contains data as if they were the result of an inner join on these two tables. Assume that there are two tables, an <code>Order</code> table with two <code>NUMBER</code> columns <code>Order_id</code> and <code>Person_id</code>, and a <code>Person</code> table with a <code>NUMBER</code> column <code>Person_id</code> and a <code>VARCHAR2</code> column <code>Name</code>.

```
. . .
// RowSet holding data from table Order
OracleCachedRowSet ocrsOrder = new OracleCachedRowSet();
. . .
ocrsOrder.setCommand("select order_id, person_id from order");
// Join on person id column
ocrsOrder.setMatchColumn(2);
ocrsOrder.execute();
// Creating the JoinRowSet
OracleJoinRowSet ojrs = new OracleJoinRowSet();
ojrs.addRowSet(ocrsOrder);
// RowSet holding data from table Person
OracleCachedRowSet ocrsPerson = new OracleCachedRowSet();
. . .
ocrsPerson.setCommand("select person id, name from person");
. . .
// do not set match column on this RowSet using setMatchColumn().
//use addRowSet() to set match column
ocrsPerson.execute();
// Join on person id column, in another way
ojrs.addRowSet(ocrsPerson, 1);
// now we can go the JoinRowSet as usual
ojrs.beforeFirst();
while (ojrs.next())
System.out.println("order id = " + ojrs.getInt(1) + ", " + "person id = " +
ojrs.getInt(2) + ", " + "person's name = " + ojrs.getString(3));
. . .
```

20 Globalization Support

The Oracle Java Database Connectivity (JDBC) drivers provide globalization support, formerly known as National Language Support (NLS). Globalization support enables you to retrieve data or insert data into a database in any character set that Oracle supports. If the clients and the server use different character sets, then the driver provides the support to perform the conversions between the database character set and the client character set.

This chapter contains the following sections:

- About Providing Globalization Support
- NCHAR_ NVARCHAR2_ NCLOB and the defaultNChar Property
- New Methods for National Character Set Type Data in JDK 6

Note:

- Starting from Oracle Database 10g, the NLS_LANG variable is no longer part of the JDBC globalization mechanism. The JDBC driver does not check NLS environment. So, setting it has no effect.
- The JDBC server-side internal driver provides complete globalization support and does not require any globalization extension files.
- JDBC 4.0 includes methods for reading and writing national character set values. You should use these methods when using JSE 6 or later.

Related Topics

- Oracle Character Data Types Support
- Oracle Database Globalization Support Guide

20.1 About Providing Globalization Support

The basic Java Archive (JAR) file ojdbc7.jar, contains all the necessary classes to provide complete globalization support for:

- Oracle character sets for CHAR, VARCHAR, LONGVARCHAR, or CLOB data that is not being retrieved or inserted as a data member of an Oracle object or collection type.
- CHAR **or** VARCHAR **data members of object and collection for the character sets** US7ASCII, WE8DEC, WE8IS08859P1, WE8MSWIN1252, **and** UTF8.

To use any other character sets in CHAR or VARCHAR data members of objects or collections, you must include orail8n.jar in the CLASSPATH environment variable:

ORACLE_HOME/jlib/orai18n.jar



Previous releases depended on the nls_charset12.zip file. This file is now obsolete.

Compressing orai18n.jar

The orai18n.jar file contains many important character set and globalization support files. You can reduce the size of orai18n.jar using the built-in customization tool, as follows:

```
java -jar orail8n.jar -custom-charsets-jar [jar/zip_filename] -charset
characterset_name [characterset_name ...]
```

For example, if you want to create a custom character set file, custom_orai18n_ja.jar, that includes the JA16SJIS and JA16EUC character sets, then issue the following command:

```
$ java -jar orail8n.jar -custom-charsets-jar custom_orail8n_ja.jar -charset
JA16SJIS JA16EUC
```

The output of the command is as follows:

Added Character set : JA16SJIS Added Character set : JA16EUC

If you do not specify a file name for your custom JAR/ZIP file, then a file with the name jdbc_orai18n_cs.jar is created in the current working directory. Also, for your custom JAR/ZIP file, you cannot specify a name that starts with orai18n.

If any invalid or unsupported character set name is specified in the command, then no output JAR/ZIP file will be created. If the custom JAR/ZIP file exists, then the file will not be updated or removed.

The custom character set JAR/ZIP does not accept any command. However, it prints the version information and the command that was used to generate the JAR/ZIP file. For example, you have jdbc_orail8n_cs.zip, the command that displays the information and the displayed information is as follows:

```
$ java -jar jdbc_orail8n_cs.jar
Oracle Globalization Development Kit - 12.1.X.X.X Release
This custom character set jar/zip file was created with the following command:
java -jar orail8n.jar -custom-charsets-jar jdbc_orail8n_cs.jar -charset
WE8ISO8859P15
```

The limitation to the number of character sets that can be specified depends on that of the shell or command prompt of the operating system. It is certified that all supported character sets can be specified with the command.



If you are using a custom character set, then you need to perform the following so that JDBC supports the custom character set:

 After creating the .nlt and .nlb files as part of the process of creating a custom character set, create .glb files for the newly created character set and also for the lx0boot.nlt file using the following command:

java -classpath \$ORACLE_HOME/jlib/orai18n.jar:\$ORACLE_HOME/lib/ xmlparserv2.jar Ginstall -[add | a] <NLT_file_name>

2. Add the generated files and <code>\$ORACLE_HOME/jlib/orail8n-mappings.jar</code> into the <code>classpath</code> environment variable while executing the JDBC code that connects to the database with the custom character set.

20.2 NCHAR, NVARCHAR2, NCLOB and the defaultNChar Property

By default, the oracle.jdbc.OraclePreparedStatement interface treats the data type of all the columns in the same way as they are encoded in the database character set. However, since Oracle Database 10g, if you set the value of oracle.jdbc.defaultNChar system property to true, then JDBC treats all character columns as being national-language.

The default value of defaultNChar is false. If the value of defaultNChar is false, then you must call the setFormOfUse(<column_Index>, OraclePreparedStatement.FORM_NCHAR) method for those columns that specifically need national-language characters. For example:

```
PreparedStatement pstmt =
conn.prepareStatement("insert into TEST values(?,?,?)");
pstmt.setFormOfUse(1, OraclePreparedStatement.FORM_NCHAR);
pstmt.setString(1, myUnicodeString1); // NCHAR column
pstmt.setFormOfUse(2, OraclePreparedeStatement.FORM_NCHAR);
pstmt.setString(2, myUnicodeString2); // NVARCHAR2 column
```

If you want to set the value of defaultNChar to true, then specify the following at the command-line:

java -Doracle.jdbc.defaultNChar=true myApplication

If you prefer, then you can also specify defaultNChar as a connection property and access NCHAR, NVARCHAR2, or NCLOB data.

```
Properties props = new Properties();
props.put(OracleConnection.CONNECTION_PROPERTY_DEFAULTNCHAR, "true");
// set URL, username, password, and so on.
...
Connection conn = DriverManager.getConnection(props);
```

If the value of defaultNChar is true, then you should call the setFormOfUse(<column_Index>, FORM_CHAR) for columns that do not need national-language characters. For example:

```
PreparedStatement pstmt =
conn.prepareStatement("insert into TEST values(?,?,?)");
```



pstmt.setFormOfUse(3, OraclePreparedStatement.FORM_CHAR);
pstmt.setString(3, myString); // CHAR column

Note:

If you set the value of defaultNChar to true and then access CHAR columns, then the database will implicitly convert all CHAR data into NCHAR. This conversion has a substantial performance impact.

Note:

- Always use java.lang.String for character data instead of oracle.sql.CHAR. CHAR is provided only for backward compatibility.
- You can also use the setObject method to access national character set types, but if the setObject method is used, then the target data type must be specified as Types.NCHAR, Types.NCLOB, Types.NVARCHAR, or Types.LONGNVARCHAR.

Note:

In Oracle Database, SQL strings are converted to the database character set. Therefore you need to keep in mind the following:

- In Oracle Database 10g release 1 (10.1) and earlier releases, JDBC drivers do not support any NCHAR literal (n'...') containing Unicode characters that are not representable in the database character set. All Unicode characters that are not representable in the database character set get corrupted.
- If an Oracle Database 10g release 2 (10.2) JDBC driver is connected to an Oracle Database 10g release 2 (10.2) database server, then all NCHAR literals (n'...') are converted to Unicode literals (u'...') and all non-ASCII characters are converted to their corresponding Unicode escape sequence. This is done automatically to prevent data corruption.
- If an Oracle Database 10g release 2 (10.2) JDBC driver is connected to an Oracle Database 10g release 1 (10.1) or earlier database server, then NCHAR literals (n'...') are not converted and any character that is not representable in the database character set gets corrupted.

20.3 New Methods for National Character Set Type Data in JDK 6

JDBC 4.0 introduces support for the following four additional SQL types to access the national character set types:



- NCHAR
- NVARCHAR
- LONGNVARCHAR
- NCLOB

These types are similar to the CHAR, VARCHAR, LONGVARCHAR, and CLOB types, except that the values are encoded using the national character set. The JDBC specification uses the String class to represent NCHAR, NVARCHAR, and LONGNVARCHAR data, and the NClob class to represent NCLOB values.

To retrieve a national character value, an application calls one of the following methods:

- getNString
- getNClob
- getNCharacterStream
- getObject

Note:

The getClob method may be used to return an NClob object since NClob implements Clob.

To specify a value for a parameter marker of national character type, an application calls one of the following methods:

- setNString
- setNCharacterStream
- setNClob
- setObject

Note:

You can use the setFormOfUse method to specify a national character value in JDK 6. But this practice is discouraged because this method will be deprecated in future release. So, Oracle recommends you to use the methods discussed in this section.

See Also:

If the setObject method is used, then the target data type must be specified as Types.NCHAR, Types.NCLOB, Types.NVARCHAR, Or Types.LONGNVARCHAR.



Part V

Performance and Scalability

This part consists of chapters that discuss the Oracle Java Database Connectivity (JDBC) features that enhance performance, such as Statement caching and Oracle Call Interface (OCI) connection pooling. It also includes a chapter that provides information about Oracle performance extensions, such as update batching and row prefetching.

Part V contains the following chapters:

- Statement and Result Set Caching
- Performance Extensions
- JDBC Reactive Extensions
- Support for Java library for Reactive Streams Ingestion
- Support for Pipelined Database Operations
- OCI Connection Pooling
- Database Resident Connection Pooling
- JDBC Support for Database Sharding
- Oracle Advanced Queuing
- Continuous Query Notification



21 Statement and Result Set Caching

This chapter describes the benefits and use of Statement caching, an Oracle Java Database Connectivity (JDBC) extension.

Note:

Use statement caching only when you are sure that the table structure remains the same in the database. If you alter the table structure and then reuse a statement that was created and executed before changing the table structure, then you may get an error.

This chapter contains the following sections:

- About Statement Caching
- About Using Statement Caching
- About Reusing Statements Objects
- About Result Set Caching

21.1 About Statement Caching

Statement caching improves performance by caching executable statements that are used repeatedly, such as in a loop or in a method that is called repeatedly. Starting from JDBC 3.0, JDBC standards define a statement-caching interface.

Statement caching can do the following:

- Prevent the overhead of repeated cursor creation
- Prevent repeated statement parsing and creation
- Reuse data structures in the client

This section covers the following topics:

- Basics of Statement Caching
- Implicit Statement Caching
- Explicit Statement Caching



Note:

Oracle strongly recommends you use the implicit Statement cache. Oracle JDBC drivers are designed on the assumption that the implicit Statement cache is enabled. So, not using the Statement cache will have a negative impact on performance.

21.1.1 Basics of Statement Caching

Applications use the Statement cache to cache statements associated with a particular physical connection. The cache is associated with an <code>OracleConnection</code> object. <code>OracleConnection</code> includes methods to enable Statement caching. When you enable Statement caching, a statement object is cached when you call the close method.

Because each physical connection has its own cache, multiple caches can exist if you enable Statement caching for multiple physical connections. When you enable Statement caching on a connection cache, the logical connections benefit from the Statement caching that is enabled on the underlying physical connection. If you try to enable Statement caching on a logical connection held by a connection cache, then this will throw an exception.

There are two types of Statement caching: implicit and explicit. Each type of Statement cache can be enabled or disabled independent of the other. You can have either, neither, or both in effect. Both types of Statement caching share a single cache per connection.

21.1.2 Implicit Statement Caching

When you enable implicit Statement caching, JDBC automatically caches the prepared or callable statement when you call the close method of this statement object. The prepared and callable statements are cached and retrieved using standard connection object and statement object methods.

Plain statements are not implicitly cached, because implicit Statement caching uses a SQL string as a key and plain statements are created without a SQL string. Therefore, implicit Statement caching applies only to the <code>OraclePreparedStatement</code> and <code>OracleCallableStatement</code> objects, which are created with a SQL string. You cannot use implicit Statement caching with <code>OracleStatement</code>. When you create an <code>OraclePreparedStatement</code> or <code>OracleCallableStatement</code>, the JDBC driver automatically searches the cache for a matching statement. The match criteria are the following:

- The SQL string in the statement must be identical to one in the cache.
- The statement type must be the same, that is, prepared or callable.
- The scrollable type of result sets produced by the statement must be the same, that is, forward-only or scrollable.

If a match is found during the cache search, then the cached statement is returned. If a match is not found, then a new statement is created and returned. In either case, the statement, along with its cursor and state are cached when you call the close method of the statement object.



When a cached OraclePreparedStatement or OracleCallableStatement object is retrieved, the state and data information are automatically reinitialized and reset to default values, while metadata is saved. Statements are removed from the cache to conform to the maximum size using a Least Recently Used (LRU) algorithm.

Note:

The JDBC driver does not clear metadata. However, although metadata is saved for performance reasons, it has no semantic impact. A statement that comes from the implicit cache appears as if it were newly created.

You can prevent a particular statement from being implicitly cached.

Related Topics

About Using Implicit Statement Caching

21.1.3 Explicit Statement Caching

Explicit Statement caching enables you to cache and retrieve selected prepared and callable statements. Explicit Statement caching relies on a key, an arbitrary Java String that you provide.

Note: Plain statements cannot be cached.

Because explicit Statement caching retains statement data and state as well as metadata, it has a performance edge over implicit Statement caching, which retains only metadata. However, you must be cautious when using this type of caching, because explicit Statement caching saves all three types of information for reuse and you may not be aware of what data and state are retained from prior use of the statements.

Implicit and explicit Statement caching can be differentiated on the following points:

Retrieving statements

In the case of implicit Statement caching, you take no special action to retrieve statements from a cache. Instead, whenever you call prepareStatement or prepareCall, JDBC automatically checks the cache for a matching statement and returns it if found. However, in the case of explicit Statement caching, you use specialized Oracle WithKey methods to cache and retrieve statement objects.

Providing key

Implicit Statement caching uses the SQL string of a prepared or callable statement as the key, requiring no action on your part. In contrast, explicit Statement caching requires you to provide a Java String, which it uses as the key.

Returning statements



During implicit Statement caching, if the JDBC driver cannot find a statement in cache, then it will automatically create one. However, during explicit Statement caching, if the JDBC driver cannot find a matching statement in cache, then it will return a null value.

Table 21-1 compares the different methods employed in implicit and explicit Statement caching.

Table 21-1 Comparing Methods Used in Statement Caching

Type of Caching	Allocate	Insert Into Cache	Retrieve From Cache		
Implicit	prepareStatement prepareCall	close	prepareStatement prepareCall		
Explicit	createStatement prepareStatement prepareCall	closeWithKey	getStatementWithKey getCallWithKey		

21.2 About Using Statement Caching

This section discusses the following topics:

- About Enabling and Disabling Statement Caching
- About Closing a Cached Statement
- About Using Implicit Statement Caching
- About Using Explicit Statement Caching

21.2.1 About Enabling and Disabling Statement Caching

When using the OracleConnection API, implicit and explicit Statement caching can be enabled or disabled independent of one other. You can have either, neither, or both of them in effect.

Enabling Implicit Statement Caching

There are two ways to enable implicit Statement caching. The first method enables Statement caching on a nonpooled physical connection, where you need to explicitly specify the Statement size for every connection, using the setStatementCacheSize method. The second method enables Statement caching on a pooled logical connection. Each connection in the pool has its own Statement cache with the same maximum size that can be specified by setting the MaxStatementsLimit property.

Method 1

Perform the following steps:

 Call the OracleDataSource.setImplicitCachingEnabled(true) method on the connection to set the OracleDataSource property implicitCachingEnabled to true. For example:

```
OracleDataSource ods = new OracleDataSource();
...
```



```
ods.setImplicitCachingEnabled(true);
...
```

• Call the OracleConnection.setStatementCacheSize method on the physical connection. The argument you supply is the maximum number of statements in the cache. For example, the following code specifies a cache size of ten statements:

```
((OracleConnection)conn).setStatementCacheSize(10);
```

Method 2

Perform the following steps:

• Set the OracleDataSource properties implicitCachingEnabled and connectionCachingEnabled to true. For example:

```
OracleDataSource ods = new OracleDataSource();
...
ods.setConnectionCachingEnabled( true );
ods.setImplicitCachingEnabled( true );
...
```

• Set the MaxStatementsLimit property to a positive integer on the connection cache, when using the connection cache. For example:

```
Properties cacheProps = new Properties();
...
cacheProps.put( "MaxStatementsLimit", "50" );
```

To determine whether implicit caching is enabled, call getImplicitCachingEnabled, which returns true if implicit caching is enabled, false otherwise.

Note:

Enabling Statement caching enables both implicit and explicit Statement caching.

Disabling Implicit Statement Caching

Disable implicit Statement caching by calling setImplicitCachingEnabled(false) on the connection or by setting the ImplicitCachingEnabled property to false.

Enabling Explicit Statement Caching

To enable explicit Statement caching you must first set the Statement cache size. For setting the cache size, call <code>OracleConnection.setStatementCacheSize</code> method on the physical connection. The argument you supply is the maximum number of statements in the cache. An argument of <code>0</code> specifies no caching. To check the cache size, use the <code>getStatementCacheSize</code> method in the following way:

```
System.out.println("Stmt Cache size is " +
   ((OracleConnection)conn).getStatementCacheSize());
```

The following code specifies a cache size of ten statements:

((OracleConnection)conn).setStatementCacheSize(10);

Enable explicit Statement caching by calling setExplicitCachingEnabled(true) on the connection.



To determine whether explicit caching is enabled, call getExplicitCachingEnabled, which returns true if explicit caching is enabled, false otherwise.

Note:

- You enable implicit and explicit caching for a particular physical connection independently. Therefore, it is possible to do Statement caching both implicitly and explicitly during the same session.
- Implicit and explicit Statement caching share the same cache. Remember this when you set the statement cache size.

Disabling Explicit Statement Caching

Disable explicit Statement caching by calling setExplicitCachingEnabled(false). Disabling caching or closing the cache purges the cache. The following example disables explicit Statement caching:

((OracleConnection)conn).setExplicitCachingEnabled(false);

21.2.2 About Closing a Cached Statement

Perform the following to close a Statement and assure that it is not returned to the cache:

In J2SE 5.0

Disable caching for that statement

stmt.setDisableStmtCaching(true);

Call the close method of the statement object

stmt.close();

In JSE 6.0

```
stmt.setPoolable(false);
stmt.close();
```

Physically Closing a Cached Statement

With implicit Statement caching enabled, you cannot physically close statements manually. The close method of a statement object caches the statement instead of closing it. The statement is physically closed automatically under one of following three conditions:

- When the associated connection is closed
- When the cache reaches its size limit and the least recently used statement object is preempted from cache by the LRU algorithm
- If you call the close method on a statement for which Statement caching is disabled



21.2.3 About Using Implicit Statement Caching

Once you enable implicit Statement caching, by default, all prepared and callable statements are automatically cached. Implicit Statement caching includes the following steps:

- 1. Enable implicit Statement caching.
- 2. Allocate a statement using one of the standard methods.
- **3.** Disable implicit Statement caching for any particular statement you do not want to cache. This is an optional step.
- 4. Cache the statement using the close method.
- 5. Retrieve the implicitly cached statement by calling the appropriate standard prepare method.

Allocating a Statement for Implicit Caching

To allocate a statement for implicit Statement caching, use either the prepareStatement or prepareCall method as you would typically.

The following code allocates a new statement object called pstmt:

```
PreparedStatement pstmt = conn.prepareStatement
  ("UPDATE emp SET ename = ? WHERE rowid = ?");
```

Disabling Implicit Statement Caching for a Particular Statement

With implicit Statement caching enabled for a connection, by default, all callable and prepared statements of that connection are automatically cached. To prevent a particular callable or prepared statement from being implicitly cached, use the setDisableStmtCaching method of the statement object. You can manage cache space by calling the setDisableStmtCaching method on any infrequently used statement.

The following code disables implicit Statement caching for pstmt:

```
PreparedStatement pstmt = conn.prepareStatement("SELECT 1 from DUAL");
((OraclePreparedStatement)pstmt).setDisableStmtCaching(true);
pstmt.close ();
```

Note:

If you are using JSE 6, then you can disable Statement caching by using the standard JDBC 4.0 method setPoolable:

PreparedStatement.setPoolable(false);

Use the following to check whether the Statement object is poolable:

Statement.isPoolable();

Implicitly Caching a Statement

To cache an allocated statement, call the close method of the statement object. When you call the close method on an OraclePreparedStatement or OracleCallableStatement object,



the JDBC driver automatically puts this statement in cache, unless you have disabled caching for this statement.

The following code caches the pstmt statement:

pstmt.close ();

Retrieving an Implicitly Cached Statement

To retrieve an implicitly cached statement, call either the prepareStatement or prepareCall method, depending on the statement type.

The following code retrieves pstmt from cache using the prepareStatement method:

pstmt = conn.prepareStatement ("UPDATE emp SET ename = ? WHERE rowid = ?");

21.2.3.1 Methods Used in Statement Allocation and Implicit Statement Caching

Table 21-2 describes the methods used to allocate statements and retrieve implicitly cached statements.

Method	Functionality for Implicit Statement Caching
prepareStatement	Performs a cache search that either finds and returns the desired cached OraclePreparedStatement object or allocates a new OraclePreparedStatement object if a match is not found
prepareCall	Performs a cache search that either finds and returns the desired cached OracleCallableStatement object or allocates a new OracleCallableStatement object if a match is not found

 Table 21-2
 Methods Used in Statement Allocation and Implicit Statement

 Caching
 Implicit Statement

Example 21-1 provides a sample code that shows how to enable implicit statement caching.

Example 21-1 Using Implicit Statement Cache

```
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Properties;
import javax.sql.DataSource;
import oracle.jdbc.OracleConnection;
import oracle.jdbc.pool.OracleDataSource;
public class TestJdbc
{
  /**
  * Get a Connection, prepare a statement, execute a query, fetch the results,
close the connection.
   * @param ods the DataSource used to get the connection.
   */
  private static void doSQL( DataSource ods ) throws SQLException
    final String SQL = "select username from all users";
    OracleConnection conn = null;
    PreparedStatement ps = null;
```



```
ResultSet rs = null;
    try
    {
     conn = (OracleConnection) ods.getConnection();
      System.out.println( "Connection:" + conn );
      System.out.println( "Connection getImplicitCachingEnabled:" +
conn.getImplicitCachingEnabled() );
     System.out.println( "Connection getStatementCacheSize:" +
conn.getStatementCacheSize() );
     ps = conn.prepareStatement( SQL );
      System.out.println( "PreparedStatement:" + ps );
      rs = ps.executeQuery();
      while ( rs.next() )
      {
       String owner = rs.getString( 1 );
       System.out.println( owner );
      }
    }
    finally
    {
     if ( rs != null )
      {
       rs.close();
      }
      if ( ps != null )
      {
       ps.close();
     conn.close();
    }
  }
  }
 public static void main( String[] args )
  {
    try
    {
     OracleDataSource ods = new OracleDataSource();
     ods.setDriverType( "thin" );
     ods.setServerName( "localhost" );
     ods.setPortNumber( 5221 );
     ods.setServiceName( "orcl" );
     ods.setUser( "HR" );
     ods.setPassword( "hr" );
     ods.setConnectionCachingEnabled( true );
     ods.setImplicitCachingEnabled( true );
     Properties cacheProps = new Properties();
      cacheProps.put( "InitialLimit", "1" );
      cacheProps.put( "MinLimit", "1" );
      cacheProps.put( "MaxLimit", "5" );
      cacheProps.put( "MaxStatementsLimit", "50" );
      ods.setConnectionCacheProperties( cacheProps );
     System.out.println( "DataSource getImplicitCachingEnabled: " +
ods.getImplicitCachingEnabled() );
      for ( int i = 0; i < 5; i++ )
      {
        doSQL( ods );
      }
    catch ( Exception ex )
    {
     ex.printStackTrace();
    }
```



```
}
```

21.2.4 About Using Explicit Statement Caching

A prepared or callable statement can be explicitly cached when you enable explicit Statement caching. Explicit Statement caching includes the following steps:

- 1. Enable explicit Statement caching.
- 2. Allocate a statement using one of the standard methods.
- 3. Explicitly cache the statement by closing it with a key, using the closeWithKey method.
- 4. Retrieve the explicitly cached statement by calling the appropriate Oracle WithKey method, specifying the appropriate key.
- 5. Re-cache an open, explicitly cached statement by closing it again with the closeWithKey method. Each time a cached statement is closed, it is re-cached with its key.

Allocating a Statement for Explicit Caching

To allocate a statement for explicit Statement caching, use either the createStatement, prepareStatement, or prepareCall method as you would typically.

The following code allocates a new statement object called pstmt:

```
PreparedStatement pstmt =
    conn.prepareStatement ("UPDATE emp SET ename = ? WHERE rowid = ?");
```

Explicitly Caching a Statement

To explicitly cache an allocated statement, call the closeWithKey method of the statement object, specifying a key. The key is an arbitrary Java String that you provide. The closeWithKey method caches a statement as is. This means the data, state, and metadata are retained and not cleared.

The following code caches the pstmt statement with the key "mykey":

((OraclePreparedStatement)pstmt).closeWithKey ("mykey");

Retrieving an Explicitly Cached Statement

To recall an explicitly cached statement, call either the getStatementWithKey or getCallWithKey methods depending on the statement type.

If you retrieve a statement with a specified key, then the JDBC driver searches the cache for the statement, based on the specified key. If a match is found, then the matching statement is returned along with its state, data, and metadata. This information is as it was when the statement was last closed. If a match is not found, then the JDBC driver returns null.

The following code recalls pstmt from cache using the "mykey" key with the getStatementWithKey method. Recall that the pstmt statement object was cached with the "mykey" key.

pstmt = ((OracleConnection)conn).getStatementWithKey ("mykey");



If you call the creationState method on the pstmt statement object, then the method returns EXPLICIT.

Note:

When you retrieve an explicitly cached statement, ensure that you use the method that is appropriate for your statement type when specifying the key. For example, if you used the prepareStatement method to allocate a statement, then use the getStatementWithKey method to retrieve that statement from cache. The JDBC driver does *not* verify the type of statement it is returning.

21.2.4.1 Methods Used to Retrieve Explicitly Cached Statements

Table 21-3 describes the methods used to retrieve explicitly cached statements.

Method	Functionality for Explicit Statement Caching		
getStatementWithKey	Specifies the key needed to retrieve a prepared statement from cache		
getCallWithKey	Specifies the key needed to retrieve a callable statement from cache		

 Table 21-3
 Methods Used to Retrieve Explicitly Cached Statements

21.3 About Reusing Statements Objects

The JDBC 3.0 specification introduces the feature of statement pooling that enables an application to reuse a PreparedStatement object in the same way as it uses a Connection object. The PreparedStatement objects can be reused by multiple logical connections in a transparent manner.

This section covers the following topics:

- About Using a Pooled Statement
- About Closing a Pooled Statement

Note:

The Oracle JDBC Drivers use implicit statement caching to support statement pooling.

21.3.1 About Using a Pooled Statement

An application can find out whether a data source supports statement pooling by calling the isPoolable method from the Statement interface. If the return value is true, then the application knows that the PreparedStatement object is being pooled. The application can



also request a statement to be pooled or not pooled by using the setPoolable method from the Statement interface.

Reusing of pooled statement should be completely transparent to the application, that is, the application code should remain the same whether a PreparedStatement object participates in statement pooling or not. If an application closes a PreparedStatement object, it must still call Connection.prepareStatement method in order to reuse it.

Note:

An application has no direct control over how statements are pooled. A pool of statements is associated with a PooledConnection object, whose behavior is determined by the properties of the ConnectionPoolDataSource object that produced it.

21.3.2 About Closing a Pooled Statement

An application closes a pooled statement exactly the same way it closes a nonpooled statement. Once a statement is closed, whether is it pooled or nonpooled, it is no longer available for use by the application and an attempt to reuse it causes an exception to be thrown. The only difference visible is that an application cannot directly close a physical statement that is being pooled. This is done by the pool manager. The method PooledConnection.closeAll closes all of the statements open on a given physical connection, which releases the resources associated with those statements.

The following methods can close a pooled statement:

• close

This java.sql.Statement interface method is called by an application. If the statement is being pooled, then it closes the logical statement used by the application but does not close the physical statement being pooled.

• close

This java.sql.Connection interface method is called by an application. This method acts differently depending upon whether the connection using the statement is being pooled or not:

Nonpooled connection

This method closes the physical connection and all statements created by that connection. This is necessary because the garbage collection mechanism is unable to detect when externally managed resources can be released.

Pooled connection

This method closes the logical connection and the logical statements it returned, but leaves open the underlying PooledConnection object and any associated pooled statements

• PooledConnection.closeAll

This method is called by the connection pool manager to close all of the physical statements being pooled by the PooledConnection object



21.4 About Result Set Caching

Your applications sometime send repetitive queries to the database. To improve the response time of repetitive queries, results of queries, query fragments, and PL/SQL functions can be cached in memory. A result cache stores the results of queries shared across all sessions. When these queries are executed repeatedly, the results are retrieved directly from the cache memory.

Note:

If a result set is very large, then it may not be cached due to size restrictions.

You must annotate a query or query fragment with a result cache hint to indicate that results are to be stored in the query result cache.

The query result set can be cached in the following ways:

- Server-Side Result Set Cache
- Client-Side Result Set Cache

Note:

- The server-side and client result set caches are most useful for read-only or read-mostly data. They may reduce performance for queries with highly dynamic results.
- Both server-side and client result set caches use memory. So, caching very large result sets can cause performance problems.

21.4.1 Server-Side Result Set Cache

Support for server-side Result Set caching has been introduced for both JDBC Thin and JDBC Oracle Call Interface (OCI) drivers since Oracle Database 11*g* Release 1. The serverside result cache is used to cache the results of the current queries, query fragments, and PL/SQL functions in memory and then to use the cached results in future executions of the query, query fragment, or PL/SQL function. The cached results reside in the result cache memory portion of the SGA. A cached result is automatically invalidated whenever a database object used in its creation is successfully modified. The server-side caching can be of the following two types:

- SQL Query Result Cache
- PL/SQL Function Result Cache



See Also:

- Oracle Database Performance Tuning Guide for more information about SQL Query Result Cache
- Oracle Database PL/SQL Language Reference for more information about PL/SQL Function Result Cache

21.4.2 Client-Side Result Set Cache

Client-side result set cache feature enables client-side caching of SQL query result sets in client memory. In this way, the applications can use client memory to take advantage of the client-side result set cache to improve response times of repetitive queries.

This section covers the following topics:

- Enabling the Client-Side Result Set Cache
- Benefits of Client-Side Result Set Cache
- Usage Guidelines in JDBC

21.4.2.1 Enabling the Client-Side Result Set Cache

Oracle Database Release 18c supports client-side result set cache in the JDBC thin driver. You can use the new oracle.jdbc.enableQueryResultCache connection property for enabling this feature. The default value of this property is true, which means that this feature is enabled by default. You can disable this feature by setting the property to false.

Note:

- In Oracle Database 12c Release 2 (12.2), the enableQueryResultCache property is available as enableResultSetCache, and the default value is false. You can enable this feature by setting the enableResultSetCache property to true.
- The JDBC OCI driver supports client-side result set cache already.

See Also:

Oracle Call Interface Programmer's Guide



For using this feature, you must set the following database initialization parameters in the following way:

```
CLIENT_RESULT_CACHE_SIZE=100M
CLIENT_RESULT_CACHE_LAG=1000
```

This value of the CLIENT_RESULT_CACHE_SIZE parameter controls how much memory the thin driver can use for its cache.

A read-only or read-mostly table can then be annoted and its data can be cached on the driver. For example, RESULT CACHE (MODE FORCE).

You can also use a SQL hint $/*+RESULT_CACHE */$ for identifying queries that are eligible for caching.



21.4.2.2 Benefits of Client-Side Result Set Cache

The benefits of the client-side result set cache are the following:

- The client-side result set cache is completely transparent to the applications and its cache of result set data is kept consistent with any session or database changes that affect its result set.
- Table annotation makes client-side result set work transparently to the JDBC applications. Otherwise, you must use a hint to enable it. The cache hit avoids the execution of the query and roundtrip to the server to get the result sets. This can result in huge performance savings for server resources, for example, server CPU and server I/O.



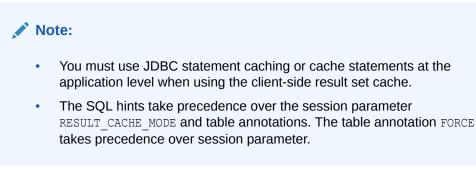
- The result cache on the client is per-process, so multiple client sessions can simultaneously use matching cached result sets.
- The result cache on the client minimizes the need for each application to have its own custom result set cache.
- The result cache on the client uses the client memory that is cheaper than server memory.

21.4.2.3 Usage Guidelines in JDBC

You can enable result set caching in the following three ways:

- RESULT_CACHE_MODE Parameter
- Table Annotations
- SQL Hints





Related Topics

Statement and Result Set Caching

21.4.2.3.1 RESULT_CACHE_MODE Parameter

You can use the RESULT_CACHE_MODE parameter to decide the result cache mode across tables in your queries. Use this clause with the ALTER SESSION and ALTER SYSTEM statements, or inside the server parameter file (init.ora) to determine result caching. You can set the RESULT_CACHE_MODE parameter to control whether the SQL query result cache is used for all queries, or only for the queries that are annotated with the result cache hint using SQL hints or table annotations.

21.4.2.3.2 Table Annotations

You can use table annotations to enable result caching without making changes to the code. The ALTER TABLE and CREATE TABLE statements enable you to annotate tables with result cache mode. The syntax is:

CREATE|ALTER TABLE [<schema>.] ... [RESULT_CACHE (MODE {FORCE|DEFAULT})]

Following example shows how to use table annotations with CREATE TABLE statements:

CREATE TABLE foo (a NUMBER, b VARCHAR2(20)) RESULT_CACHE (MODE FORCE);

Following example shows how to use table annotations with ALTER TABLE statements:

ALTER TABLE foo RESULT CACHE (MODE DEFAULT);

21.4.2.3.3 SQL Hints

You can use SQL hints to specify the queries to be cached by annotating the queries with a /*+ result_cache */ or /*+ no_result_cache */ hint. For example, look at the following code snippet:

```
String query = "select /*+ result_cache */ * from employees where employee_id
< : 1";
    ((oracle.jdbc.OracleConnection)conn).setImplicitCachingEnabled(true);
    ((oracle.jdbc.OracleConnection)conn).setStatementCacheSize(10);
    PreparedStatement pstmt;
    ResultSet rs;
    for (int j = 0 ; j < 10 ; j++ )
    {
        pstmt = conn.prepareStatement (query);
        pstmt.setInt(1,7500);
        rs = pstmt.executeQuery();
    }
}    for (int j = 0; j < 10; j++ )
</pre>
```

```
while (rs.next())
{    // see the values }
    rs.close;
    pstmt.close();
}
```

In the preceding example, the client result cache hint /*+ result_cache */ is annotated to the actual query, that is, select * from employees where employee_id < : 1. So, the first execution of the query goes to the database and the result set is cached for the remaining nine executions of the query. This improves the performance of your application significantly. This is primarily useful for read-only data.

Following are some more examples of SQL hints. All the following examples assume that the dept table is annotated for result caching by using the following command:

ALTER TABLE dept result_cache (MODE FORCE);

Examples

}

SELECT * FROM employees

The result set will not be cached.

SELECT * FROM departments

The result set will be cached.

- SELECT /*+ result_cache */ employee_id FROM employees The result set will be cached.
- SELECT /*+ no_result_cache */ department_id FROM departments
 The result set will not be cached.
- SELECT /*+ result_cache */ * FROM departments

The result set will be cached though query hint is not necessary.

 SELECT e.first_name FROM employees e, departments d WHERE e.department_id = d.department_id

The result set will not be cached because neither is a query hint available nor are all the tables annotated as FORCE.

Note:

For information about usage guidelines, Client cache consistency, Deployment Time settings, Client cache Statistics, Validation of client result cache, and OCI Client Result Cache and Server Result Cache, refer to the *Oracle Call Interface Programmer's Guide*.



22 Performance Extensions

This chapter describes the Oracle performance extensions to the Java Database Connectivity (JDBC) standard.

This chapter covers the following topics:

- Update Batching
- Additional Oracle Performance Extensions

Note:

Oracle update batching was deprecated in Oracle Database 12c Release 1 (12.1). Starting in Oracle Database 12c Release 2 (12.2), Oracle update batching is a no operation code (no-op). This means that if you implement Oracle update batching in your application, using the Oracle Database 12c Release 2 (12.2) JDBC driver, then the specified batch size is not set and results in a batch size of 1. With this batch setting, your application processes one row at a time. Oracle strongly recommends that you use the standard JDBC batching if you are using the Oracle Database 12c Release 2 (12.2) JDBC driver.

22.1 Update Batching

This section covers the following topics:

- Overview of Update Batching
- Standard Update Batching
- Premature Batch Flush

22.1.1 Overview of Update Batching

Update batching is especially useful with prepared statements, when you are repeating the same statement with different bind variables.

You can reduce the number of round-trips to the database and improve the application performance, by grouping multiple UPDATE, DELETE, or INSERT statements into a single batch, and then having the whole batch sent to the database and processed in one trip. This is referred to as 'update batching'.



Note:

- The JDBC 2.0 specification refers to 'update batching' as 'batch updates'.
- To adhere to the JDBC 2.0 standard, Oracle implementation of standard update batching supports callable statements without OUT parameters, generic statements, and prepared statements. You can migrate standard update batching into an Oracle JDBC application without difficulty. However, the Oracle implementation of standard update batching does not implement true batching for generic statements and callable statements and you will see performance improvement for only PreparedStatement objects.

22.1.2 Standard Update Batching

JDBC standard update batching depends on explicitly adding statements to the batch using an addBatch method and explicitly processing the batch using an executeBatch method.

Note:

Disable auto-commit mode when you use update batching. In case an error occurs while you are processing a batch, this provides you the option of committing or rolling back the operations that ran successfully prior to the error.

22.1.2.1 About Adding Operations to the Batch

When any statement object is first created, its statement batch is empty. Use the standard addBatch method to add an operation to the statement batch. This method is specified in the standard java.sql.Statement, PreparedStatement, and CallableStatement interfaces, which are implemented by the oracle.jdbc.OracleStatement, OraclePreparedStatement, and OracleCallableStatement interfaces, respectively.

Note:

Starting from Oracle Databse Release 23c, when you use the addBatch method, the JDBC driver starts a pipeline with the Database, which results in improved response time and throughput. See Support for Pipelined Database Operations.

For a Statement object, the addBatch method takes a Java String with a SQL operation as input. For example:

```
Statement stmt = conn.createStatement();
```



```
stmt.addBatch("INSERT INTO emp VALUES(1000, 'Joe Jones')");
stmt.addBatch("INSERT INTO dept VALUES(260, 'Sales')");
stmt.addBatch("INSERT INTO emp_dept VALUES(1000, 260)");
```

At this point, three operations are in the batch.

For prepared statements, update batching is used to batch multiple runs of the same statement with different sets of bind parameters. For a PreparedStatement or OraclePreparedStatement object, the addBatch method takes no input. It simply adds the operation to the batch using the bind parameters last set by the appropriate setXXX methods. This is also true for CallableStatement or OracleCallableStatement objects, but remember that in the Oracle implementation of standard update batching, you will probably see no performance improvement in batching callable statements.

For example:

At this point, two operations are in the batch.

Because a batch is associated with a single prepared statement object, you can batch only repeated runs of a single prepared statement, as in this example.

22.1.2.2 About Processing the Batch

To process the current batch of operations, use the executeBatch method of the statement object. This method is specified in the standard Statement interface, which is extended by the standard PreparedStatement and CallableStatement interfaces.

💉 Note:

If you add too many operations to a batch by calling the addBatch method several times and create a very large batch (for example, with more than or equal to 100,000 rows), then while calling the executeBatch method on the whole batch, you may face severe performance problems in terms of memory. To avoid this issue, the JDBC driver transparently breaks up the large batches into smaller internal batches and makes a roundtrip to the server for each internal batch. This makes your application slightly slower because of each round-trip overhead, but optimizes memory significantly. However, if each bound row is very large in size (for example, more than about 1MB each or so), then this process can impact the overall performance negatively because in such a case, the performance gained in terms of memory will be less than the performance lost in terms of time.



Following is an example that repeats the prepared statement addBatch calls shown previously and then processes the batch:

22.1.2.3 Row Count per Iteration for Array DMLs

Starting from Oracle Database 12c Release 1 (12.1), the executeBatch method has been improved so that it returns an int array of size that is the same as the number of records in the batch and each item in the return array is the number of database table rows affected by the corresponding record of the batch. For example, if the batch size is 5, then the executeBatch method returns an array of size 5. In case of an error in between execution of the batch, the executeBatch method cannot return a value, instead it throws a BatchUpdateException. In this case, the exception itself carries an int array of size n as its data, where n is the number of successful record executions. For example, if the batch is of size 5 and the error occurs at the 4th record, then the BatchUpdateException has an array of size 3 (3 records executed successfully) and each item in the array represents how many rows were affected by each of them.

22.1.2.4 About Committing the Changes in the Oracle Implementation of Standard Batching

After you process the batch, you must still commit the changes, presuming autocommit is disabled as recommended.

Calling commit, commits nonbatched operations and batched operations for statement batches that have been processed, but for the Oracle implementation of standard batching, has no effect on pending statement batches that have *not* been processed.

22.1.2.5 About Clearing the Batch

To clear the current batch of operations instead of processing it, use the clearBatch method of the statement object. This method is specified in the standard Statement interface, which is extended by the standard PreparedStatement and CallableStatement interfaces.

Keep the following things in mind:

- When a batch is processed, operations are performed in the order in which they were batched.
- After calling addBatch, you must call either executeBatch or clearBatch before a call to executeUpdate, otherwise there will be a SQL exception.



- A clearBatch or executeBatch call resets the statement batch to empty.
- The statement batch is not reset to empty if the connection receives a ROLLBACK request. You must explicitly call clearBatch to reset it.
- Invoking clearBatch method after a rollback works for all releases.
- An executeBatch call closes the current result set of the statement object, if one exists.
- Nothing is returned by the clearBatch method.

Following is an example that repeats the prepared statement addBatch calls shown previously but then clears the batch under certain circumstances:

```
PreparedStatement pstmt =
          conn.prepareStatement("INSERT INTO employees VALUES(?, ?)");
pstmt.setInt(1, 2000);
pstmt.setString(2, "Milo Mumford");
pstmt.addBatch();
pstmt.setInt(1, 3000);
pstmt.setString(2, "Sulu Simpson");
pstmt.addBatch();
if (...condition...)
{
   int[] updateCounts = pstmt.executeBatch();
   . . .
}
else
{
   pstmt.clearBatch();
   . . .
}
```

22.1.2.6 Update Counts in the Oracle Implementation of Standard Batching

If a statement batch is processed successfully, then the integer array, or update counts array, returned by the statement <code>executeBatch</code> call will always have one element for each operation in the batch. In the Oracle implementation of standard update batching, the values of the array elements are as follows:

- For a prepared statement batch, the array contains the actual update counts indicating the number of rows affected by each operation.
- For a generic statement batch, the array contains the actual update counts indicating the number of rows affected by each operation. The actual update counts can be provided only in the case of generic statements in the Oracle implementation of standard batching.
- For a callable statement batch, the array contains the actual update counts indicating the number of rows affected by each operation.

In your code, upon successful processing of a batch, you should be prepared to handle either -2, 1, or true update counts in the array elements. For a successful batch processing, the array contains either all -2, 1, or all positive integers.

Example 22-1 illustrates the use of standard update batching.



Example 22-1 Standard Update Batching

This example combines the sample fragments in the previous sections, accomplishing the following steps:

- 1. Disabling auto-commit mode, which you should always perform when using update batching
- 2. Creating a prepared statement object
- 3. Adding operations to the batch associated with the prepared statement object
- 4. Processing the batch
- 5. Committing the operations from the batch

You can process the update counts array to determine if the batch processed successfully.

22.1.2.7 Error Handling in the Oracle Implementation of Standard Batching

If any one of the batched operations fails to complete successfully or attempts to return a result set during an executeBatch call, then the processing stops and a java.sql.BatchUpdateException is generated.

After a batch exception, the update counts array can be retrieved using the getUpdateCounts method of the BatchUpdateException object. This returns an int array of update counts, just as the executeBatch method does. In the Oracle implementation of standard update batching, contents of the update counts array are as follows, after a batch is processed:

• For a prepared statement batch, in case of an error in between execution of the batch, the executeBatch method cannot return a value, instead it throws a BatchUpdateException. In this case, the exception itself carries an int array of size n as its data, where n is the number of successful record executions. For example, if the batch is of size 5 and the error occurs at the 4th record, then the BatchUpdateException has an array of size 3 (3 records executed successfully) and each item in the array represents how many rows were affected by each of them.



• For a generic statement batch or callable statement batch, the update counts array is only a partial array containing the actual update counts up to the point of the error. The actual update counts can be provided because Oracle JDBC cannot use true batching for generic and callable statements in the Oracle implementation of standard update batching.

For example, if there were 20 operations in the batch, the first 13 succeeded, and the 14th generated an exception, then the update counts array will have 13 elements, containing actual update counts of the successful operations.

You can either commit or roll back the successful operations in this situation, as you prefer.

In your code, upon failed processing of a batch, you should be prepared to handle either -3 or true update counts in the array elements when an exception occurs. For a failed batch processing, you will have either a full array of -3 or a partial array of positive integers.

22.1.2.8 About Intermixing Batched Statements and Nonbatched Statements

You cannot call <code>executeUpdate</code> for regular, nonbatched processing of an operation if the statement object has a pending batch of operations.

However, you can intermix batched operations and nonbatched operations in a single statement object if you process nonbatched operations either prior to adding any operations to the statement batch or after processing the batch. Essentially, you can call <code>executeUpdate</code> for a statement object only when its update batch is empty. If the batch is non-empty, then an exception will be generated.

For example, it is valid to have a sequence, such as the following:

```
. . .
PreparedStatement pstmt =
          conn.prepareStatement("INSERT INTO employees VALUES(?, ?)");
pstmt.setInt(1, 2000);
pstmt.setString(2, "Milo Mumford");
int scount = pstmt.executeUpdate(); // OK; no operations in pstmt batch
pstmt.setInt(1, 3000);
pstmt.setString(2, "Sulu Simpson");
pstmt.addBatch();
                                     // Now start a batch
pstmt.setInt(1, 4000);
pstmt.setString(2, "Stan Leland");
pstmt.addBatch();
int[] bcounts = pstmt.executeBatch();
pstmt.setInt(1, 5000);
pstmt.setString(2, "Amy Feiner");
int scount = pstmt.executeUpdate(); // OK; pstmt batch was executed
. . .
```

Intermixing nonbatched operations on one statement object and batched operations on another statement object within your code is permissible. Different statement objects are independent of each other with regard to update batching operations. A COMMIT request will



affect all nonbatched operations and all successful operations in processed batches, but will not affect any pending batches.

22.1.2.9 Limitations in the Oracle Implementation of Standard Batching

This section discusses the limitations and implementation details regarding the Oracle implementation of standard update batching.

In Oracle JDBC applications, update batching is intended for use with prepared statements that are being processed repeatedly with different sets of bind values.

The Oracle implementation of standard update batching does not implement true batching for generic statements and callable statements. Even though Oracle JDBC supports the use of standard batching for Statement and CallableStatement objects, you are unlikely to see performance improvement.

22.1.3 Premature Batch Flush

Premature batch flush happens due to a change in cached metadata. Cached metadata can be changed due to various reasons, such as the following:

- The initial bind was null and the following bind is not null.
- A scalar type is initially bound as string and then bound as scalar type or the reverse.

The premature batch flush count is summed to the return value of the next executeUpdate or sendBatch method.

The old functionality lost all these batch flush values which can be obtained now. To switch back to the old functionality, you can set the <code>AccumulateBatchResult</code> property to false, as follows:

```
java.util.Properties info = new java.util.Properties();
info.setProperty("user", "HR");
info.setProperty("passwd", "hr");
// other properties
...
// property: batch flush type
info.setProperty("AccumulateBatchResult", "false");
OracleDataSource ods = new OracleDataSource();
ods.setConnectionProperties(info);
ods.setURL("jdbc:oracle:oci:@"");
Connection conn = ods.getConnection();
```

Note:

The AccumulateBatchResult property is set to true by default.

Example 22-2 illustrates premature batch flushing.



Example 22-2 Premature Batch Flushing

```
((OraclePreparedStatement)pstmt).setExecuteBatch (2);
pstmt.setNull(1, OracleTypes.NUMBER);
pstmt.setString(2, "test11");
int count = pstmt.executeUpdate(); // returns 0
/*
* Premature batch flush happens here.
*/
pstmt.setInt(1, 22);
pstmt.setString(2, "test22");
int count = pstmt.executeUpdate(); // returns 0
pstmt.setInt(1, 33);
pstmt.setString(2, "test33");
/*
*
  returns 3 with the new batching scheme where as,
*
  returns 2 with the old batching scheme.
*/
int count = pstmt.executeUpdate();
```

22.2 Additional Oracle Performance Extensions

In addition to update batching, Oracle JDBC drivers support the extensions discussed in this chpter, which improve performance by reducing round-trips to the database:

Prefetching rows

This reduces round-trips to the database by fetching multiple rows of data each time data is fetched. The extra data is stored in client-side buffers for later access by the client. The number of rows to prefetch can be set as desired.

Specifying column types

This avoids an inefficiency in the standard JDBC protocol for performing and returning the results of queries.

Suppressing database metadata TABLE REMARKS columns

This avoids an expensive outer join operation.

Oracle provides several extensions to connection properties objects to support these performance extensions. These extensions enable you to set the <code>remarksReporting</code> flag and default values for row prefetching and update batching.

This section covers the following topics:

- Oracle Row-Prefetching Limitations
- About Defining Column Types
- About Reporting DatabaseMetaData TABLE_REMARKS

22.2.1 Oracle Row-Prefetching Limitations

There is no maximum prefetch setting. The default value is 10. Larger or smaller values may be appropriate depending on the number of rows and columns expected from the query. You can set the default connection row-prefetch value using a Properties object.



When a statement object is created, it receives the default row-prefetch setting from the associated connection. Subsequent changes to the default connection row-prefetch setting will have no effect on the statement row-prefetch setting.

If a column of a result set is of data type LONG, LONG RAW or LOBS returned through the data interface, that is, the streaming types, then JDBC changes the statement row-prefetch setting to 1, even if you never actually read a value of either of these types.

Setting the prefetch size can affect the performance of an application. Increasing the prefetch size will reduce the number of round-trips required to get all the data, but will increase memory usage. This will depend on the number and size of the columns in the query and the number of rows expected to be returned. It will also depend on the memory and CPU loading of the JDBC client machine. The optimum for a standalone client application will be different from a heavily loaded application server. The speed and latency of the network connection should also be considered.

Note:

Starting from Oracle Database 11g Release 1, the Thin driver can fetch the first prefetch_size number of rows from the server in the very first round-trip. This saves one round-trip in SELECT statements.

If you are migrating an application from earlier releases of Oracle JDBC drivers to 10g Release 1 (10.1) or later releases of Oracle JDBC drivers, then you should revisit the optimizations that you had done earlier, because the memory usage and performance characteristics may have changed substantially.

A common situation that you may encounter is, say, you have a query that selects a unique key. The query will return only zero or one row. Setting the prefetch size to 1 will decrease memory and CPU usage and cannot increase round-trips. However, you must be careful to avoid the error of requesting an extra fetch by writing while (rs.next()) instead of if (rs.next()).

If you are using the JDBC Thin driver, then use the useFetchSizeWithLongColumn connection property, because it will perform PARSE, EXECUTE, and FETCH in a single round-trip.

Tuning of the prefetch size should be done along with tuning of memory management in your JVM under realistic loads of the actual application.



Note:

- Do not mix the JDBC 2.0 fetch size application programming interface (API) and the Oracle row-prefetching API in your application. You can use one or the other, but not both.
- Be aware that setting the Oracle fetch size value can affect not only queries, but also explicitly refetching rows in a result set through the result set refreshRow method, which is relevant for scroll-sensitive/read-only, scroll-sensitive/ updatable, and scroll-insensitive/updatable result sets, and the window size of a scroll-sensitive result set, affecting how often automatic refetches are performed. However, the Oracle fetch size value will be overridden by any setting of the fetch size.

22.2.2 About Defining Column Types

Note:

Starting from Oracle Database 12c Release 1 (12.1), the defineColumnType method is deprecated.

The implementation of defineColumnType changed significantly since Oracle Database 10g. Previously, defineColumnType was used both as a performance optimization and to force data type conversion. In previous releases, all of the drivers benefited from calls to defineColumnType. Starting from Oracle Database 10g, the JDBC Thin driver no longer needs the information provided. The JDBC Thin driver achieves maximum performance without calls to defineColumnType. The JDBC Oracle Call Interface (OCI) and server-side internal drivers still get better performance when the application uses defineColumnType.

If your code is used with both the JDBC Thin and OCI drivers, you can disable the defineColumnType method when using the Thin driver by setting the connection property disableDefineColumnType to true. Doing this makes defineColumnType have no effect. Do not set this connection property to true when using the JDBC OCI or server-side internal drivers.

You can also use defineColumnType to control how much memory the client-side allocates or to limit the size of variable-length data.

Follow these general steps to define column types for a query:

- If necessary, cast your statement object to OracleStatement, OraclePreparedStatement, or OracleCallableStatement, as applicable.
- 2. If necessary, use the clearDefines method of your Statement object to clear any previous column definitions for this Statement object.
- 3. On each column, call the defineColumnType method of your Statement object, passing it these parameters:
 - Column index (integer)
 - Type code (integer)



Use the static constants of the java.sql.Types class or oracle.jdbc.OracleTypes class, such as Types.INTEGER, Types.FLOAT, Types.VARCHAR, OracleTypes.VARCHAR, and OracleTypes.ROWID. Type codes for standard types are identical in these two classes.

Type name (string)

For structured objects, object references, and arrays, you must also specify the type name. For example, Employee, EmployeeRef, Or EmployeeArray.

Maximum field size (integer)

Optionally specify a maximum data length for this column.

You cannot specify a maximum field size parameter if you are defining the column type for a structured object, object reference, or array. If you try to include this parameter, it will be ignored.

Form of use (short)

Optionally specify a form of use for the column. This can be OraclePreparedStatement.FORM_CHAR to use the database character set or OraclePreparedStatement.FORM_NCHAR to use the national character set. If this parameter is omitted, the default is FORM_CHAR.

For example, assuming stmt is an Oracle statement, use:

stmt.defineColumnType(column_index, typeCode);

If the column is VARCHAR or equivalent and you know the length limit:

stmt.defineColumnType(column_index, typeCode, max_size);

For an NVARCHAR column where the original maximum length is desired and conversion to the database character set is requested:

For structured object, object reference, and array columns:

stmt.defineColumnType(column_index, typeCode, typeName);

Set a maximum field size if you do not want to receive the full default length of the data. Calling the setMaxFieldSize method of the standard JDBC Statement class sets a restriction on the amount of data returned. Specifically, the size of the data returned will be the minimum of the following:

- The maximum field size set in defineColumnType
- The maximum field size set in setMaxFieldSize
- The natural maximum size of the data type

After you complete these steps, use the <code>executeQuery</code> method of the statement to perform the query.



Note:

It is no longer necessary to specify a data type for each column of the expected result set.

The following example illustrates the use of this feature. It assumes you have imported the oracle.jdbc.* interfaces.

Example 22-3 Defining Column Types

```
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:oracle:thin:@localhost:5221:orcl");
ods.setUser("HR");
ods.setPassword("hr");
Connection conn = ods.getConnection();
Statement stmt = conn.createStatement();
// Allocate only 2 chars for this column (truncation will happen)
((OracleStatement)stmt).defineColumnType(1, Types.VARCHAR, 2);
ResultSet rset = stmt.executeQuery("select ename from emp");
while(rset.next() )
   System.out.println(rset.getString(1));
stmt.close();
```

As this example shows, you must cast the Statement object, stmt, to OracleStatement in the invocation of the defineColumnType method. The createStatement method of the connection returns an object of type java.sql.Statement, which does not have the defineColumnType and clearDefines methods. These methods are provided only in the OracleStatement implementation.

The define-extensions use JDBC types to specify the desired types. The allowed define types for columns depend on the internal Oracle type of the column.

All columns can be defined to their natural JDBC types. In most cases, they can be defined to the Types.CHAR or Types.VARCHAR type code.

The following table lists the valid column definition arguments that you can use in the defineColumnType method.

If the column has Oracle SQL type:	You can use defineColumnType to define it as:		
NUMBER, VARNUM	BIGINT, TINYINT, SMALLINT, INTEGER, FLOAT, REAL, DOUBLE, NUMERIC, DECIMAL, CHAR, VARCHAR		
CHAR, VARCHAR2	CHAR, VARCHAR		
LONG	CHAR, VARCHAR, LONGVARCHAR		
LONGRAW	LONGVARBINARY, VARBINARY, BINARY		
RAW	VARBINARY, BINARY		
DATE	DATE, TIME, TIMESTAMP, CHAR, VARCHAR		
ROWID	ROWID		

Table 22-1 Valid Column Type Specifications



If the column has Oracle SQL type:	You can use defineColumnType to define it as:
BLOB	VARBINARY, BINARY
CLOB	LONG, CHAR, VARCHAR

Table 22-1	(Cont.)	Valid Column	Type S	pecifications
------------	---------	--------------	--------	---------------

It is always valid to use defineColumnType with the original data type of the column.

22.2.3 About Reporting DatabaseMetaData TABLE_REMARKS

The getColumns, getProcedureColumns, getProcedures, and getTables methods of the database metadata classes are slow if they must report TABLE_REMARKS columns, because this necessitates an expensive outer join. For this reason, the JDBC driver does *not* report TABLE_REMARKS columns by default.

You can enable TABLE_REMARKS reporting by passing a true argument to the setRemarksReporting method of an OracleConnection object.

Equivalently, instead of calling setRemarksReporting, you can set the remarksReporting Java property if you use a Java Properties object in establishing the connection.

If you are using a standard java.sql.Connection object, you must cast it to OracleConnection to use setRemarksReporting.

The following code snippet illustrates how to enable TABLE REMARKS reporting:

((oracle.jdbc.OracleConnection)conn).setRemarksReporting(true);

Here, conn is the name of your standard Connection object, the following statement enables TABLE REMARKS reporting:

Considerations for getColumns

By default, the getColumns method does not retrieve information about the columns if a synonym is specified. To enable the retrieval of information if a synonym is specified, you must call the setIncludeSynonyms method on the connection as follows:

((oracle.jdbc.OracleConnection)conn).setIncludeSynonyms(true)

This will cause all subsequent getColumns method calls on the connection to include synonyms. This is similar to setRemarksReporting. Alternatively, you can set the includeSynonyms connection property. This is similar to the remarksReporting connection property.

However, bear in mind that if includeSynonyms is true, then the name of the object returned in the table_name column will be the synonym name, if a synonym exists. This is true even if you pass the table name to getColumns.

Considerations for getProcedures and getProcedureColumns Methods

According to JDBC versions 1.1 and 1.2, the methods getProcedures and getProcedureColumns treat the catalog, schemaPattern, columnNamePattern, and



procedureNamePattern parameters in the same way. In the Oracle definition of these methods, the parameters are treated differently:

catalog

Oracle does not have multiple catalogs, but it does have packages. Consequently, the catalog parameter is treated as the package name. This applies both on input, which is the catalog parameter, and the output, which is the catalog column in the returned ResultSet. On input, the construct " ", which is an empty string, retrieves procedures and arguments without a package, that is, standalone objects. A null value means to drop from the selection criteria, that is, return information about both standalone and packaged objects. That is, it has the same effect as passing in the percent sign (%). Otherwise, the catalog parameter should be a package name pattern, with SQL wild cards, if desired.

• schemaPattern

All objects within Oracle database must have a schema, so it does not make sense to return information for those objects without one. Thus, the construct " ", which is an empty string, is interpreted on input to mean the objects in the current schema, that is, the one to which you are currently connected. To be consistent with the behavior of the catalog parameter, null is interpreted to drop the schema from the selection criteria. That is, it has the same effect as passing in %. It can also be used as a pattern with SQL wild cards.

• procedureNamePattern **and** columnNamePattern

The empty string (" ") does not make sense for either parameter, because all procedures and arguments must have names. Thus, the construct " " will raise an exception. To be consistent with the behavior of other parameters, null has the same effect as passing in percent sign ().

23 JDBC Reactive Extensions

The Reactive Extensions are a set of methods that extend the JDBC standard to offer asynchronous database access.

The Reactive Extensions use non-blocking mechanisms for creating connection objects, executing SQL statements, fetching rows, committing transactions, rolling back transactions, closing Connection objects, and reading and writing BFILEs, BLOBs, and CLOBs.

This chapter includes the following topics:

- Overview of JDBC Reactive Extensions
- About Building an Application with Reactive Extensions
- Threading Model of Asynchronous Methods
- About the Flow API
- Using the FlowAdapters Class
- Streaming Row Data with the Reactor Library
- Streaming Row Data with the RxJava Library
- Streaming Row Data with the Akka Streams Library
- Limitations of JDBC Reactive Extensions

23.1 Overview of JDBC Reactive Extensions

The Reactive Extensions implement the Publisher and Subscriber types defined by the java.util.concurrent.Flow interfaces, which is the standard JDK representation of a reactive stream.

The Reactive Extensions use a single Java NIO Selector for nonblocking Database operations.

Requirements for Using JDBC Reactive Extensions

For using the JDBC Reactive Extensions, you must use the following:

- JDBC Thin Driver 21c or later for building connections
- JDK 11
- ojdbc11.jar

Architecture of JDBC Reactive Extensions

The following diagram illustrates the architecture of applications using JDBC Reactive Extensions:



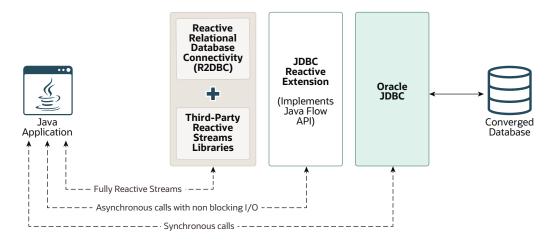


Figure 23-1 Architecture of Applications Using JDBC Reactive Extensions

23.2 About Building an Application with Reactive Extensions

This section describes the steps that you must follow to build an application using the Reactive Extensions.

Building an application using the Reactive Extensions involves the same steps as building an application using the standard methods. But, in case of Reactive Extensions, you use the new asynchronous methods. This section describes how to use the various asynchronous methods in the following sections:

- Opening a Connection Using Asynchronous Methods
- Execution of SQL Statements with Asynchronous Methods
- About Fetching Row Data with Asynchronous Methods
- Reading LOB Data Using Asynchronous Methods
- Writing LOB Data Using Asynchronous Methods
- Committing a Transaction Using Asynchronous Methods
- Closing a Connection Using Asynchronous Methods

23.2.1 Opening a Connection Using Asynchronous Methods

The OracleConnectionBuilder interface provides methods for opening a connection asynchronously.

The OracleConnectionBuilder.buildConnectionPublisherOracle method returns a Flow.Publisher<OracleConnection> type. The Publisher emits a single Connection to a Subscriber. Once the Subscriber signals demand, the Publisher asynchronously opens a new Connection. The Published Connection is identical to a Connection that you can build using the ConnectionBuilder.build method.

The following example demonstrates how to asynchronously open a connection.

/**

- * Asynchronously opens a new connection
- * @param dataSource Datasource configured with a URL, User, and



```
Password
 * @return A Publisher that emits a single connection
 * @throws SQLException If a database access error occurs before the
 * connection can be opened
 */
Flow.Publisher<OracleConnection> openConnection(DataSource dataSource)
 throws SQLException {
  return dataSource.unwrap(OracleDataSource.class)
    .createConnectionBuilder()
    .buildConnectionPublisherOracle();
}
```

23.2.2 Execution of SQL Statements with Asynchronous Methods

This section describes how to execute SQL statements with asynchronous methods.

The OraclePreparedStatement interface exposes methods for asynchronous SQL execution. Each asynchronous method performs a function that is analogous to the corresponding synchronous method of SQL execution. This relationship is expressed in the following table:

Synchronous Method	Asynchronous Method	
boolean execute	Flow.Publisher <boolean> executeAsyncOracle</boolean>	
long executeLargeUpdate	Flow.Publisher <long> executeUpdateAsyncOracle</long>	
<pre>long[] executeLargeBatch</pre>	Flow.Publisher <long> executeBatchAsyncOracle</long>	
ResultSet executeQuery	Flow.Publisher <oracleresultset> executeQueryAsyncOracle</oracleresultset>	

Table 23-1 Method Comparison

The following sections provide more information about the asynchronous methods:

- Standard SQL Statement Execution with the executeAsyncOracle Method
- DML Statement Execution with the executeUpdateAsyncOracle method
- Batch DML Statement Execution with the executeBatchAsyncOracle Method
- SQL Query Execution with the executeQueryAsyncOracle Method

23.2.2.1 Standard SQL Statement Execution with the executeAsyncOracle Method

This section describes the executeAsyncOracle method, which is equivalent to the standard execute method.

Any type of SQL statement can be executed by calling the

OraclePreparedStatement.executeAsyncOracle method. This call returns a Flow.Publisher<Boolean> type. The Publisher emits a single Boolean and supports multiple Subscribers. If the Boolean value is TRUE, then it means that the SQL statement has resulted in row data, which is accessible from the OraclePreparedStatement.getResultSet method.



If it is FALSE, then it means that the SQL statement has returned an update count. The Boolean result is semantically equivalent to the boolean returned by the execute method.

```
/**
   * Asynchronously creates a new table by executing a DDL SQL
statement
   * @param connection Connection to a database where the table is
created
   * @return A Publisher that emits the result of executing DDL SQL
   * @throws SQLException If a database access error occurs before the
DDL
   * SQL can be executed
   */
 Flow.Publisher<Boolean> createTable(Connection connection)
    throws SQLException {
    PreparedStatement createTableStatement =
      connection.prepareStatement(
        "CREATE TABLE employee names (" +
          "id NUMBER PRIMARY KEY, " +
          "first name VARCHAR(50), " +
          "last name VARCHAR2(50))");
    Flow.Publisher<Boolean> createTablePublisher =
      createTableStatement.unwrap(OraclePreparedStatement.class)
        .executeAsyncOracle();
    createTablePublisher.subscribe(
      // This subscriber will close the PreparedStatement
      new Flow.Subscriber<Boolean>() {
        public void onSubscribe(Flow.Subscription subscription) {
          subscription.request(1L);
        }
        public void onNext(Boolean item) { }
        public void onError(Throwable throwable) { closeStatement(); }
        public void onComplete() { closeStatement(); }
        void closeStatement() {
          try { createTableStatement.close(); }
          catch (SQLException closeException) { log(closeException); }
        }
      });
    return createTablePublisher;
```

}



23.2.2.2 DML Statement Execution with the executeUpdateAsyncOracle method

This section describes the <code>executeUpdateAsyncOracle</code> method, which is equivalent to the standard <code>executeLargeUpdate</code> method.

You can use the <code>OraclePreparedStatement.executeUpdateAsyncOracle</code> method to execute single (non-batch) DML statements. This call returns a <code>Flow.Publisher<Long></code> type. The returned publisher emits a single <code>Long</code> value. This <code>Long</code> value indicates the number of rows updated or to be inserted by the DML statement. This <code>Long</code> value result is semantically equivalent to the <code>long</code> value returned by the standard <code>executeLargeUpdate</code> method.

```
/**
 * Asynchronously updates table data by executing a DML SQL statement
 * @param connection Connection to a database where the table data resides
 * @return A Publisher that emits the number of rows updated
 * @throws SQLException If a database access error occurs before the DML
 * SQL can be executed
 */
Flow.Publisher<Long> updateData(Connection connection)
  throws SQLException {
  PreparedStatement updateStatement = connection.prepareStatement(
    "UPDATE employee names SET " +
      "first name = UPPER(first name), " +
      "last name = UPPER(last name)");
  Flow.Publisher<Long> updatePublisher =
    updateStatement.unwrap(OraclePreparedStatement.class)
      .executeUpdateAsyncOracle();
  updatePublisher.subscribe(
    // This subscriber will close the PreparedStatement
    new Flow.Subscriber<Long>() {
      public void onSubscribe(Flow.Subscription subscription) {
        subscription.request(1L);
      }
     public void onNext(Long item) { }
      public void onError(Throwable throwable) { closeStatement(); }
      public void onComplete() { closeStatement(); }
      void closeStatement() {
        try { updateStatement.close(); }
        catch (SQLException closeException) { log(closeException); }
      }
    });
  return updatePublisher;
}
```



23.2.2.3 Batch DML Statement Execution with the executeBatchAsyncOracle Method

This section describes the executeBatchAsyncOracle method, which is equivalent to the standard executeLargeBatch method.

You can use the <code>OraclePreparedStatement.executeBatchAsyncOracle</code> method to execute batch DML statements. This call returns a <code>Flow.Publisher<Long></code> type. The returned publisher emits a <code>Long</code> value for each statement in the batch. The <code>Long</code> values indicate the number of rows updated by each DML statement. These <code>Long</code> value results are semantically equivalent to the <code>long[]</code> value returned by the standard executeLargeBatch method.

```
/**
   * Asynchronously loads table data by executing a batch of DML SQL
statements.
   * @param connection Connection to a database where the table data
resides.
   * @return A Publisher which emits the number of rows updated.
   * @throws SQLException If a database access error occurs before the
DML
   * SOL can be executed.
   */
 Flow.Publisher<Long> createData(
    Connection connection, Iterable<Employee> employeeData)
    throws SQLException {
    PreparedStatement batchStatement = connection.prepareStatement(
      "INSERT INTO employee names (id, first name, last name) " +
        "VALUES (?, ?, ?)");
    for (Employee employee : employeeData) {
     batchStatement.setLong(1, employee.id());
     batchStatement.setString(2, employee.firstName());
     batchStatement.setString(3, employee.lastName());
     batchStatement.addBatch();
    }
    Flow.Publisher<Long> batchPublisher =
     batchStatement.unwrap(OraclePreparedStatement.class)
        .executeBatchAsyncOracle();
   batchPublisher.subscribe(
      // This subscriber will close the PreparedStatement
      new Flow.Subscriber<Long>() {
        public void onSubscribe(Flow.Subscription subscription) {
          subscription.request(Long.MAX VALUE);
        }
        public void onNext(Long item) { }
        public void onError(Throwable throwable) { closeStatement(); }
        public void onComplete() { closeStatement(); }
        void closeStatement() {
          try { batchStatement.close(); }
```

```
catch (SQLException closeException) { log(closeException); }
});
return batchPublisher;
}
```

23.2.2.4 SQL Query Execution with the executeQueryAsyncOracle Method

This section describes the executeQueryAsyncOracle Method, which is equivalent to the standard executeQuery method.

```
You can execute SQL query statements with the

OraclePreparedStatement.executeQueryAsyncOracle method. This call returns a

Flow.Publisher<OracleResultSet> type. The returned publisher emits a single

OracleResultSet value. The OracleResultSet value provides access to row data that is

resulted from the SQL query. This OracleResultSet is semantically equivalent to the

ResultSet returned by the standard executeQuery method.
```

```
/**
 * Asynchronously reads table data by executing a SELECT SQL statement
 * @param connection Connection to a database where the table resides
 * @return A Publisher that emits the number of rows updated
 * @throws SQLException If a database access error occurs before the SELECT
 * SOL can be executed
 */
Flow.Publisher<OracleResultSet> readData(Connection connection)
  throws SQLException {
  PreparedStatement queryStatement = connection.prepareStatement(
    "SELECT id, first name, last name FROM employee names");
  Flow.Publisher<OracleResultSet> queryPublisher =
    queryStatement.unwrap(OraclePreparedStatement.class)
      .executeQueryAsyncOracle();
  // Close the PreparedStatement after the result set is consumed.
  queryStatement.closeOnCompletion();
  return queryPublisher;
}
```

23.2.3 About Fetching Row Data with Asynchronous Methods

This section describes how to fetch row data with asynchronous methods.

The OracleResultSet interface exposes the publisherOracle(Function<OracleRow, T>) method for asynchronous row data fetching. The argument to this method is a mapping function for row data. The mapping function is applied to each row of the ResultSet. The method returns a Flow.Publisher<T> type, where T is the output type of the mapping function. The input type of the mapping function is OracleRow. An OracleRow represents a single row of the ResultSet, and exposes methods to access the column values of that row.



The following example demonstrates how you can fetch row data with asynchronous methods:

```
/**
   * Asynchronously fetches table data by from a ResultSet.
   * @param resultSet ResultSet which fetches table data.
   * @return A Publisher which emits the fetched data as Employee
objects.
   * @throws SQLException If a database access error occurs before
table data is
   * fetched.
   */
  Flow.Publisher<Employee> fetchData(ResultSet resultSet)
    throws SQLException {
    // Before closing the ResultSet with publisherOracle(..), first
obtain a
    // reference to the ResultSet's Statement. The Statement needs to
be closed
    // after all data has been fetched.
    Statement resultSetStatement = resultSet.getStatement();
    Flow.Publisher<Employee> employeePublisher =
      resultSet.unwrap(OracleResultSet.class)
        .publisherOracle(oracleRow -> {
          try {
            return new Employee(
              oracleRow.getObject("id", Long.class),
              oracleRow.getObject("first name", String.class),
              oracleRow.getObject("last name", String.class));
          }
          catch (SQLException getObjectException) {
            // Unchecked exceptions thrown by a row mapping function
will be
            // emitted to each Subscriber's onError method.
            throw new RuntimeException (getObjectException);
          }
        });
    employeePublisher.subscribe(
      // This subscriber will close the ResultSet's Statement
      new Flow.Subscriber<Employee>() {
        public void onSubscribe(Flow.Subscription subscription) {
          subscription.request(Long.MAX VALUE);
        }
        public void onNext(Employee item) { }
        public void onError(Throwable throwable) { closeStatement(); }
        public void onComplete() { closeStatement(); }
        void closeStatement() {
          try { resultSetStatement.close(); }
          catch (SQLException closeException) { log(closeException); }
        }
      });
```

```
return employeePublisher;
}
```

Instances of OracleRow, which the mapping function receives as input, becomes invalid after the function returns. Restricting the access of OracleRow to the scope of the mapping function enables the driver to efficiently manage the memory that is used to store row data. If you need a persistent copy of an OracleRow, then you can use the OracleRow.clone method to create a new instance of OracleRow, which is backed by a copy of the original OracleRow data. The OracleRow returned by the clone method remains valid outside the scope of the mapping function and retains its data even after the database connection is closed.

The row mapping function must return a non-null value or throw an unchecked exception. If the mapping function throws an unchecked exception, then it is delivered to row data subscribers as an onError signal. The row data Publisher supports multiple Subscribers. Row data emission to multiple Subscribers follow certain policies as stated below:

- A Subscriber will not receive an onNext signal for row data emitted before the Subscriber received an onSubscribe signal.
- A Subscriber will not receive an onNext signal until demand has been signaled by all other Subscribers.

The following table demonstrates the event flow while working with multiple subscribers:

Time	Event	Cause
0	SubscriberA receives an onSubscribe signal	A call to the row data Publisher's subscribe (Subscriber) method requested a Subscription for SubscriberA
1	SubscriberA requests 1 row	SubscriberA signaled demand on its Subscription
2	SubscriberA receives data for the first row of the ResultSets	The row data Publisher fetched a row of data that was requested by SubscriberA
3	SubscriberB receives an onSubscribe signal	A call to the row data Publisher's subscribe (Subscriber) method requested a Subscription for SubscriberB
4	SubscriberA requests 1 row	SubscriberA signaled demand on its Subscription
5	SubscriberB requests 1 row	SubscriberB signaled demand on its Subscription
6	Both SubscriberA and SubscriberB receive data for the second row of the ResultSet	The row data Publisher fetched a row of data that was requested by both Subscribers
7	SubscriberA requests 1 row	SubscriberA signaled demand on its Subscription
8	No row data is emitted	The row data Publisher does not emit the next row until <i>ALL</i> subscribers have requested it.
9	SubscriberB requests 1 row	SubscriberB signaled demand on its Subscription

Table 23-2 Emission to Multiple Subscribers



Time	Event	Cause
10	Both SubscriberA and SubscriberB receive data for the third row of the ResultSet	The row data Publisher fetched a row of data that was requested by both Subscribers

Table 23-2	(Cont.) Emission to Multiple Subscribers
------------	--

Note:

SubscriberB never received data for the first row. This is because SubscriberB subscribed after the first row was emitted. Also, no data was emitted at 8 seconds. This is because all Subscribers need to request the next row before it is emitted. At 8 seconds, SubscriberA had requested for the next row, but SubscriberB had not placed a request till then.

23.2.4 Reading LOB Data Using Asynchronous Methods

The OracleBlob, OracleBFile, OracleClob, and OracleNClob interfaces expose a publisherOracle(long) method for asynchronous reading of LOB data.

The argument to the publisherOracle(long) method is a position of the LOB from where the data is read. The OracleBlob.publisherOracle(long) and OracleBFile.publisherOracle(long) methods return a Publisher
segments of binary data that have been read from the LOB. The OracleClob.publisherOracle(long) and OracleNClob.publisherOracle(long) methods return a Publisher
OracleClob.publisherOracle(long) type. This Publisher emits segments of character data that have been read from the LOB.

The following example demonstrates how to asynchronously read binary data from a LOB.

```
/**
   * Asynchronously reads binary data from a BLOB
   * @param connection Connection to a database where the BLOB resides
   * @param employeeId ID associated to the BLOB
   * @return A Publisher that emits binary data of a BLOB
   * @throws SQLException If a database access error occurs before the
   * BLOB can be read
   */
  Flow.Publisher<byte[]> readLOB(Connection connection, long
employeeId)
    throws SQLException {
    PreparedStatement lobQueryStatement = connection.prepareStatement(
      "SELECT photo bytes FROM employee photos WHERE id = ?");
    lobQueryStatement.setLong(1, employeeId);
    ResultSet resultSet = lobQueryStatement.executeQuery();
    if (!resultSet.next())
      throw new SQLException ("No photo found for employee ID " +
employeeId);
```

```
OracleBlob photoBlob =
    (OracleBlob) resultSet.unwrap (OracleResultSet.class).getBlob(1);
 Flow.Publisher<byte[]> photoPublisher = photoBlob.publisherOracle(1);
 photoPublisher.subscribe(
    // This subscriber will close the PreparedStatement and BLOB
    new Flow.Subscriber<byte[]>() {
     public void onSubscribe(Flow.Subscription subscription) {
        subscription.request(Long.MAX VALUE);
     public void onNext(byte[] item) { }
     public void onError(Throwable throwable) { freeResources(); }
     public void onComplete() { freeResources(); }
     void freeResources() {
        try { lobQueryStatement.close(); }
        catch (SQLException closeException) { log(closeException); }
        try { photoBlob.free(); }
        catch (SQLException freeException) { log(freeException); }
    });
  return photoPublisher;
}
```

You cannot configure the size of data segments emitted by the LOB publishers. The driver chooses a segment size that is optimized as per the DB_BLOCK_SIZE parameter of the database.

23.2.5 Writing LOB Data Using Asynchronous Methods

The OracleBlob, OracleClob, and OracleNClob interfaces expose a subscriberOracle(long) method for asynchronous writing of LOB data.

The argument to the subscriberOracle(long) method is a position of the LOB where the data is written. The OracleBlob.subscriberOracle(long) method returns a Subscriber
Subscriber
Subscriber
OracleClob.subscriberOracle(long) method and the OracleNClob.subscriberOracle(long) method return a Subscriber<String> type. These Subscribers receive segments of character data that are written to the LOB.

The following examples demonstrate how to asynchronously write binary data to a LOB.

```
/**
 * Asynchronously writes binary data to a BLOB
 * @param connection Connection to a database where the BLOB resides
 * @param bytesPublisher Publisher that emits binary data
 * @return A CompletionStage that completes with a reference to the BLOB,
 * after all binary data is written.
 * @throws SQLException If a database access error occurs before the table
data is
 * fetched
 */
CompletionStage<Blob> writeLOB(
 Connection connection, Flow.Publisher<byte[]> bytesPublisher)
 throws SQLException {
```



```
OracleBlob oracleBlob =
      (OracleBlob)
connection.unwrap(OracleConnection.class).createBlob();
    // This future is completed after all bytes have been written to
the BLOB
    CompletableFuture<Blob> writeFuture = new CompletableFuture<>();
    Flow.Subscriber<byte[]> blobSubscriber =
      oracleBlob.subscriberOracle(1L,
        // This Subscriber will receive a terminal signal when all
byte[]'s
        // have been written to the BLOB.
        new Flow.Subscriber<Long>() {
          long totalWriteLength = 0;
          @Override
          public void onSubscribe(Flow.Subscription subscription) {
            subscription.request(Long.MAX VALUE);
          }
          @Override
          public void onNext(Long writeLength) {
            totalWriteLength += writeLength;
            log(totalWriteLength + " bytes written.");
          }
          @Override
          public void onError(Throwable throwable) {
            writeFuture.completeExceptionally(throwable);
          }
          @Override
          public void onComplete() {
            writeFuture.complete(oracleBlob);
          }
        });
    bytesPublisher.subscribe(blobSubscriber);
    return writeFuture;
  }
```

The OracleBlob, OracleClob, and OracleNClob interfaces also expose a subscriberOracle(long, Subscriber<Long>) method that performs the same function as the single-argument form of the subscriberOracle(long) method. However, the single-argument form also accepts a Subscriber<Long> type. The Subscriber<Long> type notifies a Subscriber to receive the result of write operations against the database. Each time an asynchronous write operation completes, the Subscriber<Long> type receives an onNext signal with the number of bytes or number of characters written by the operation. If an asynchronous write operation fails, the Subscriber<Long> type receives an onError signal. After the final write operation completes, the Subscriber<Long> receives an onComplete signal.

After the CompletionStage<Blob> returned by the writeLOB method completes, the resulting Blob object can be passed to the insertLOB method to have the BLOB data stored in a table.



The following examples demonstrate how to insert the data.

```
/**
 ^{\star} Asynchronously inserts BLOB data into a table by executing a DML SQL
 * statement
 * @param connection Connection to a database where the table data resides
 * @param employeeId ID related to the BLOB data
 * @param photoBlob Reference to BLOB data
 * @return A Publisher that emits the number of rows inserted (always 1)
 * @throws SQLException If a database access error occurs before the DML
 * SOL can be executed
 */
Flow.Publisher<Long> insertLOB(
  Connection connection, long employeeId, Blob photoBlob)
  throws SQLException {
  PreparedStatement lobInsertStatement = connection.prepareStatement(
    "INSERT INTO employee photos(id, photo bytes) VALUES (?,?)");
  lobInsertStatement.setLong(1, employeeId);
  lobInsertStatement.setBlob(2, photoBlob);
  Flow.Publisher<Long> insertPublisher =
    lobInsertStatement.unwrap(OraclePreparedStatement.class)
      .executeUpdateAsyncOracle();
  insertPublisher.subscribe(new Flow.Subscriber<Long>() {
    @Override
    public void onSubscribe(Flow.Subscription subscription) {
      subscription.request(1L);
    }
    @Override
    public void onNext(Long item) { }
    @Override
    public void onError(Throwable throwable) { releaseResources(); }
    @Override
    public void onComplete() { releaseResources(); }
   void releaseResources() {
      try { lobInsertStatement.close(); }
      catch (SQLException closeException) { log(closeException); }
      try { photoBlob.free(); }
      catch (SQLException freeException) { log(freeException); }
  });
  return insertPublisher;
```

}

23.2.6 Committing a Transaction Using Asynchronous Methods

The OracleConnection interface exposes the commitAsyncOracle and rollbackAsyncOracle methods for asynchronous transaction completion.

Both the commitAsyncOracle and rollbackAsyncOracle methods return a Flow.Publisher<Void> type. The Publishers do not emit any item, as signified by their



<Void> type. The Publishers emit a single onComplete or onError signal to indicate whether the commit or rollback operation was completed successfully or not.

The following example demonstrates how to asynchronously commit a transaction.

```
/**
 * Asynchronously commits a transaction
 * @param connection Connection to a database with an active
transaction
 * @return A Publisher that emits a terminal signal when the
transaction
 * has been committed
 * @throws SQLException If a database access error occurs before the
 * transaction can be committed
 */
public Flow.Publisher<Void> commitTransaction(Connection connection)
 throws SQLException {
 return connection.unwrap(OracleConnection.class)
 .commitAsyncOracle();
}
```

The commitAsyncOracle and rollbackAsyncOracle methods perform the same function as the Connection.commit and Connection.rollback methods.

23.2.7 Closing a Connection Using Asynchronous Methods

The OracleConnection interface exposes the closeAsyncOracle method for closing an asynchronous connection.

The closeAsyncOracle method returns a Flow.Publisher<Void> type. The Publisher does not emit any item, as signified by its <Void> type. The Publisher emits a single onComplete or onError signal to indicate whether the connection was closed successfully or not.

The following example demonstrates how to asynchronously close a connection.

```
/**
 * Asynchronously closes a connection
 * @param connection Connection to be closed
 * @return A Publisher that emits a terminal signal when the
connection
 * has been closed
 * @throws SQLException If a database access error occurs before the
 * connection can be closed
 */
Flow.Publisher<Void> closeConnection(Connection connection)
 throws SQLException {
 return connection.unwrap(OracleConnection.class)
 .closeAsyncOracle();
}
```

The closeAsyncOracle method performs the same function as the Connection.close method.



23.3 Threading Model of Asynchronous Methods

This section describes the threading model of asynchronous methods.

When an asynchronous method is called, it performs as much work as possible on the calling thread, without blocking on a network read. An asynchronous method call returns immediately after a request is written to the network, without waiting for a response. If the write buffer of the operating system is not large enough to store a complete request, then the calling thread may become blocked until this buffer is flushed.

Once the write job is complete, the network channel is registered for I/O readiness polling. A single thread polls the network channels of all Oracle JDBC Connections in the same JVM. The I/O polling thread is named oracle.net.nt.TcpMultiplexer, and is configured as a daemon thread. This polling thread performs a blocking operation using a Selector.

When I/O readiness is detected for a network channel, the polling thread arranges for a worker thread to handle the event. The worker thread reads from the network and then notifies a Publisher that an operation is complete. Upon notification, the Publisher arranges worker threads that emit a signal to each of its Subscribers.

The java.util.concurrent.Executor interface manages the worker threads. The default Executor is the java.util.concurrent.ForkJoinPool.commonPool method. You can call the OracleConnectionBuilder.executorOracle(Executor) method to specify an alternative Executor.

23.4 About the Flow API

The java.util.concurrent.Flow types define the minimal set of operations that you can use to create a reactive stream.

You can write application code directly against the Flow API. However, you must implement the low-level signal processing in accordance with the reactive-streams specification as specified in the following link

https://github.com/reactive-streams/reactive-streams-jvm/blob/master/README.md

The JDBC Reactive Extensions APIs handle the low-level mechanics of the Flow API to the application using <code>java.util.concurrent.Flow.Publisher</code> and

java.util.concurrent.Flow.Subscriber. Popular libraries such as Reactor, RxJava, and Akka-Streams interface with the org.reactivestreams.Publisher and

org.reactivestreams.Subscriber types defined by the reactive-streams-jvm project as specified in the following link:

https://github.com/reactive-streams/reactive-streams-jvm/tree/master/api/src/main/java/org/ reactivestreams

Although the org.reactivestreams.Publisher type and the

org.reactivestreams.Subscriber type, and their java.util.concurrent.Flow counterparts declare the same interface, the Java compiler must still consider them to be distinct types. You can convert between the Flow type and the org.reactivestreams type using the org.reactivestreams.FlowAdapters class as specified in the following link

https://github.com/reactive-streams/reactive-streams-jvm/tree/master/api/src/main/java9/org/ reactivestreams



Oracle recommends that you use the reactive streams libraries when interfacing with the Flow types exposed by the JDBC driver.

23.5 Using the FlowAdapters Class

The libraries covered in this section interfaces with the org.reactivestreams types described in the About the Flow API section.

The following code snippet shows how the FlowAdapters class can convert the Flow.Publisher types into the org.reactivestreams types.

```
Example 23-1 Conversion to org.reactivestreams Types
```

```
public static org.reactivestreams.Publisher<ResultSet> publishQuery(
  PreparedStatement queryStatement) {
  try {
    Flow.Publisher<OracleResultSet> gueryPublisher =
      queryStatement.unwrap(OraclePreparedStatement.class)
        .executeQueryAsyncOracle();
    return FlowAdapters.toPublisher(queryPublisher);
  }
  catch (SQLException sqlException) {
    return createErrorPublisher(sqlException);
  }
}
public static <T> org.reactivestreams.Publisher<T> publishRows(
  ResultSet resultSet, Function<OracleRow, T> rowMappingFunction) {
  try {
    Flow.Publisher<T> rowPublisher =
      resultSet.unwrap(OracleResultSet.class)
        .publisherOracle(rowMappingFunction);
    return FlowAdapters.toPublisher(rowPublisher);
  }
  catch (SQLException sqlException) {
    return createErrorPublisher(sqlException);
  }
}
```

23.6 Streaming Row Data with the Reactor Library

The Reactor library defines a *Flux* type that represents a stream of many items and a *Mono* type that represents a stream of just one item.

The following example shows how to create a *Flux* of row data using the Reactive Extensions.

return Flux.using(



// Prepare a SQL statement.

() -> connection.prepareStatement("SELECT * FROM emp"),

// Execute the PreparedStatement.

preparedStatement ->

// Create a Mono which emits one ResultSet.

Mono.from(publishQuery(preparedStatement))

// Flat map the ResultSet to a Flux which emits many Rows. Each

row

// is mapped to an Employee object.

.flatMapMany(resultSet ->

publishRows(resultSet, row -> mapRowToEmployee(row))),

```
prepareStatement -> {
   try {
     prepareStatement.close();
   }
   catch (SQLException sqlException) {
     throw new RuntimeException(sqlException);
   }
});
```

The using factory method creates a *Flux* that depends on a resource that must be released explicitly. In this case, the resource is a <code>PreparedStatement</code> instance that is released by calling the <code>close</code> method.

The first lambda argument creates the PreparedStatement instance. This lambda is executed before the *Flux* begins to emit items.

The second lambda argument uses the PreparedStatement instance to create a stream of row data that is mapped to an **Employee** object. First, the PreparedStatement instance is executed by the call to the publishQuery method. As the query execution Publisher emits a



single ResultSet, it is adapted into a *Mono*. Once the query execution completes, the *Mono* emits the ResultSet into the lambda specified by the flatMapMany method. This lambda calls the publishRows method with a function to map OracleRow objects into the Employee objects. This results in the flatMapMany method call returning a Flux of Employee objects, where each Employee is mapped from a row of the ResultSet object.

The third lambda argument closes the PreparedStatement instance. This lambda is executed after the *Flux* emits its last item.

23.7 Streaming Row Data with the RxJava Library

The RxJava library defines a *Flowable* type that represents a stream of many items, and a *Single* type that represents a stream of just one item.

The following example shows how to create a *Flowable* of row data using the Reactive Extensions.

```
private Publisher<Employee> queryAllEmployees(Connection
connection) {
      return Flowable.using(
        // Prepare a SQL statement
        () -> connection.prepareStatement("SELECT * FROM emp"),
        // Execute the PreparedStatement
        queryStatement ->
          // Create a Single which emits one ResultSet
          Single.fromPublisher(publishQuery(queryStatement))
            // Flat map the ResultSet to a Flowable which emits many
rows, where
            // each row is mapped to an Employee object
            .flatMapPublisher(resultSet ->
              publishRows(resultSet, oracleRow ->
mapRowToEmployee(oracleRow))),
        // Close the PreparedStatement after emitting the last
Employee object
        PreparedStatement::close
```

);



The using factory method creates a *Flowable* that depends on a resource that must be released explicitly. In this case, the resource is a <code>PreparedStatement</code> instance that is released by calling the close method.

The first lambda argument creates the PreparedStatement instance. This lambda is executed before the *Flowable* begins to emit items.

The second lambda argument uses the PreparedStatement instance to create a stream of row data that is mapped to an **Employee** object. First, the PreparedStatement instance is executed by the publishQuery method call. As the query execution Publisher emits a single ResultSet object, it is adapted into a *Single*. Once the query execution completes, the *Single* emits the ResultSet object into the lambda specified by the flatMapPublisher method. This lambda calls the publishRows method with a function to map OracleRow objects into **Employee** objects. This results in the flatMapPublisher method call returning a *Flowable* of **Employee** objects, where each **Employee** is mapped from a row of the ResultSet.

The third method handle argument closes the PreparedStatement instance. This lambda is executed after the *Flowable* emits its last item.

23.8 Streaming Row Data with the Akka Streams Library

}

The Akka Streams library defines a Source type that represents a stream of items.

The following example shows how to create a *Source* of row data using the Reactive Extensions.

```
private Source<Employee, NotUsed> queryAllEmployees(Connection
connection) {
    final PreparedStatement queryStatement;
    try {
        queryStatement = connection.prepareStatement("SELECT * FROM emp");
    }
    catch (SQLException prepareStatementFailure) {
        return Source.failed(prepareStatementFailure);
    }
    // Create a Source which emits one ResultSet
        return Source.fromPublisher(publishQuery(queryStatement))
        // Flat map the ResultSet to a Source which emits many Rows, where
    each
```

ORACLE

```
// Row is mapped to an Employee object
.flatMapConcat(resultSet -> {
    Publisher<Employee> employeePublisher =
        publishRows(resultSet, oracleRow ->
mapRowToEmployee(oracleRow));
        return Source.fromPublisher(employeePublisher);
    })
    // This Sink closes the PreparedStatement when the Source
terminates
    .alsoTo(Sink.onComplete(result -> queryStatement.close()));
  }
```

A PreparedStatement instance is used to create a stream of row data that is mapped to an **Employee** object. First, the PreparedStatement instance is executed by the publishQuery method call. The query execution Publisher emits a single ResultSet object. This Publisher is adapted into a *Source*. Once the query completes, the *Source* emits the ResultSet object into the lambda specified by the flatMapConcat method.

This lambda calls the publishRows method with a function to map OracleRow objects into Employee objects. This results in the flatMapConcat method call returning a *Source* of Employee objects, where each Employee is mapped from a row of the ResultSet object.

The PreparedStatement instance is a resource that needs to be explicitly closed. This is handled by the alsoTo method call that specifies a *Sink* that closes the PreparedStatement instance when the *Source* emits an onComplete or onError signal.

23.9 Limitations of JDBC Reactive Extensions

This section describes the limitations of JDBC Reactive Extensions.

JDBC Reactive Extensions have the following limitations:

- You must access the asynchronous methods through the java.sql.Wrapper.unwrap method, instead of accessing those through a type cast. This ensures correct behavior of the asynchronous methods when you use them with proxy Oracle JDBC classes, for example, when you use asynchronous methods with connection pools.
- Reading large responses from the network may require blocking I/O bound operations. Blocking read operations can occur if the driver reads a response that is larger than the TCP Receive buffer size.
- Asynchronous SQL execution supports neither scrollable ResultSet types nor sensitive ResultSet types.



24

Support for Java library for Reactive Streams Ingestion

Starting with Oracle Database Release 21c, you have a Java library that provides support for Reactive Streams Ingestion (RSI), which enables efficient streaming of data into Oracle Database. The new Java library enables Java applications to continuously receive and ingest data from a large group of clients.

Using the direct path load method of Oracle Database for inserting data, the new Java library makes the ingestion process nonblocking and extremely fast. It uses an extension of the existing UCP APIs, which enables the ingestion process to furnish several high-availability and scalability features of the database, like support for table partitions, Oracle RAC connection affinity, and sharding.

24.1 Overview of the Java Library for Reactive Streams Ingestion

The Java library for Reactive Streams Ingestion (RSI) enables efficient data streaming into Oracle Database in a nonblocking way.

This library is particularly useful when a large number of clients use the database to persist information in the form of table rows, and do not want to be blocked waiting for a synchronous response from the database. So, in case of use cases like the following, when the application must ingest streaming data into the database at a very high speed, and persist it in the form of rows in an Oracle Database table, you can use this library:

- Internet of Things (IoT) sensors
- Time series data for stock trade
- Call detail records (CDRs)
- Geospatial activities
- Video web sites
- Social media posts

For using this library, you must add the following JAR files to your CLASSPATH:

- rsi.jar
- ojdbc11.jar
- ucp.jar
- ons.jar



24.2 Features of the Java Library for Reactive Streams Ingestion

The Java library for Reactive Streams Ingestion (RSI) uses the direct path load method of Oracle Database for inserting data. It also uses Oracle Universal Connection Pool (UCP) to furnish several high-availability and scalability features of the Database, such as support for table partitions, Oracle RAC connection affinity, and sharding. So, this library provides the benefits of the following features:

- Reactive Streams Ingestion
- Direct Path Load
- Universal Connection Pool

24.2.1 Reactive Streams Ingestion

This is the core feature of the Java library for Reactive Streams Ingestion, which provides the APIs to handle the logic.

The library implements the Java.util.concurrent.Flow Subscriber interface. You must implement the Java.util.concurrent.Flow Publisher interface for invoking the methods of the Subscriber interface. The Subscriber interface has the following methods:

onSubscribe

Invoke this method only once for establishing the initial relationship between the Subscriber interface and the Publisher interface.

• onNext

Invoke this method for creating each new row in the implementation of the Subscriber interface that the library provides.

• onError

Invoke this method in case of any error that might occur during the ingestion process.

• onComplete Invoke this method when the ingestion job completes.

The Subscriber interface calls the request (n) method or the cancel method of the Java.util.concurrent.Flow Subscriber interface to indicate whether it can accept more data or it should stop ingesting data.

See Also: The Java.util.concurrent.Flow Subscriber Interface

24.2.2 Direct Path Load

The Java library for Reactive Streams Ingestion uses the direct path load method of Oracle Database for nonblocking data ingestion. This method eliminates the SQL layer



overhead significantly as it formats Oracle data blocks and writes the data blocks directly to the database files.

See Also: The Direct Path Load Method

During the direct path load method call, the database appends the inserted data after the existing data in the table. This method writes data directly into data files, bypassing the buffer cache. It does not perform reuse of free space in the table, and ignores the referential integrity constraints. So, a direct path load method can perform significantly better than conventional insert operations.

24.2.3 Universal Connection Pool

The Java library for Reactive Streams Ingestion (RSI) uses Universal Connection Pool (UCP) for various connection pooling and management activities like sharding topology knowledge, Fast Application Notification (FAN) awareness for an Oracle Real Application Cluster (Oracle RAC) database, and so on.

The RSI library uses the UCP sharding APIs to establish a proper connection for the specified sharding key. You can map each record in RSI to a unique chunk ID and then use the unique chunk ID to group these records together. When the RSI library has enough records to send to the database, then it borrows a chunk specific connection from UCP and uses that connection to insert the record into the sharded database.

24.3 About Reactive Streams Ingestion (RSI) Modes

The current release introduces the new DataLoad mode, which is useful when you use the Java library for Reactive Streams Ingestion (RSI) to execute a large INSERT batch to the database.

Starting from Oracle Database 23c Release, you can use the Java library for Reactive Streams Ingestion (RSI) in the following modes, depending on your business use case:

- The Streaming mode: Use this default mode, when you want to use RSI in a server where the number of rows to be inserted is not finite, but you do not need to insert a large number of rows at one go.
- **The DataLoad Mode:** Use the DataLoad mode, when there is a known large list of records to be inserted to the database at one go.

The key differences between the DataLoad mode and the Streaming mode are:

- In the DataLoad mode, the changes are not committed until the RSI instance is closed. In the default Streaming mode, the changes are committed regularly. This can negatively impact the throughput, when you execute a large INSERT batch to the database.
- In the DataLoad mode, each worker thread has its own JDBC connection. So, there is no
 effort to reduce the number of JDBC connections needed to execute the insertion task.
 This behavior is different from the default Streaming mode, where the worker threads
 share a pool of JDBC connections.



24.3.1 Enabling the DataLoad Mode

The Streaming mode is enabled by default with the Java library for Reactive Streams Ingestion. To enable the DataLoad mode, you must use the <code>useDataLoadMode</code> method.

Enable the DataLoad mode as shown in the following code snippet:

```
ReactiveStreamsIngestion.Builder rsiBuilder =
ReactiveStreamsIngestion.builder()
    .useDataLoadMode()
    .username("<user name>")
    .password("<password>")
    .url("jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)
(HOST=myhost.com) (PORT=5521))
(CONNECT DATA=(SERVICE NAME=myservice.com)))")
    .table("customers")
    .columns (new String[] { "id", "name", "region" });
// Use try-with-resource statement to ensure that RSI instance is
closed at the
// end of the statement.
try (ReactiveStreamsIngestion rsi = rsiBuilder.build()) {
   // Publish Records.
}
```

24.4 Code Samples: Java Library for Reactive Streams Ingestion

This section contains code samples showing how to use the Reactive Streams Ingestion library.

- PushPublisher
- Flow.Publisher Dynamic Implementations
- Flow.Publisher Third-Party implementations

24.4.1 PushPublisher

This is the simplest way to use the Reactive Streams Ingestion (RSI) library, where the RSI library implements the java.util.concurrent.Flow.Subscriber interface and the Java code in your application implements the java.util.concurrent.Flow.Publisher interface.

The following example demonstrates this implementation, where you create a Publisher and the RSI library subscribes to that Publisher:

Example 24-1 PushPublisher

```
package oracle.rsi.demos;
import java.sql.SQLException;
import java.time.Duration;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
```



```
import oracle.rsi.ReactiveStreamsIngestion;
import oracle.rsi.PushPublisher;
public class SimplePushPublisher {
  public static void main(String[] args) throws SQLException {
    ExecutorService workerThreadPool = Executors.newFixedThreadPool(2);
    ReactiveStreamsIngestion rsi = ReactiveStreamsIngestion
        .builder()
        .url(
            "jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)
(HOST=example.com) (PORT=5521)) (CONNECT DATA=(SERVICE NAME=myservice.com)))")
        .username(<user name>)
        .password(<password>)
        .schema (<schema name>)
        .executor(workerThreadPool)
        .bufferRows(10)
        .bufferInterval (Duration.ofSeconds (20))
        .table("customers")
        .columns(new String[] { "id", "name", "region" })
        .build();
    PushPublisher<Object[]> pushPublisher =
ReactiveStreamsIngestion.pushPublisher();
    pushPublisher.subscribe(rsi.subscriber());
    //Ingests byte arrays using the accept method
    pushPublisher.accept(new Object[] { 1, "John Doe", "North" });
    pushPublisher.accept(new Object[] { 2, "Jane Doe", "North" });
    pushPublisher.accept(new Object[] { 3, "John Smith", "South" });
    try {
      pushPublisher.close();
    } catch (Exception e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
    rsi.close();
    workerThreadPool.shutdown();
  }
}
```

24.4.2 Flow.Publisher Dynamic Implementations

The following example shows how to use the RSI library when your application implements the Flow.Publisher interface and subscribes to the library.



Example 24-2 Flow.Publisher Dynamic Implementations

```
package oracle.rsi.demos;
import java.sql.SQLException;
import java.time.Duration;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Flow.Publisher;
import java.util.concurrent.Flow.Subscriber;
import java.util.concurrent.Flow.Subscription;
import java.util.function.Consumer;
import oracle.rsi.ReactiveStreamsIngestion;
public class SimpleFlowPublisher {
  public static void main(String[] args) throws SQLException {
    ExecutorService workerThreadPool = Executors.newFixedThreadPool(2);
    ReactiveStreamsIngestion rsi = ReactiveStreamsIngestion
        .builder()
        .url(
            "jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)
(HOST=example.com) (PORT=5521))
(CONNECT DATA=(SERVICE NAME=myservice.com)))")
        .username(<user name>)
        .password(<password>)
        .schema (<schema name>)
        .executor(workerThreadPool)
        .bufferRows(1)
        .bufferInterval(Duration.ofMinutes(60))
        .table("customers")
        .columns(new String[] { "id", "name", "region" })
        .build();
    SimpleObjectPublisher<Object[]> publisher = new
SimpleObjectPublisher<Object[]>();
    publisher.subscribe(rsi.subscriber());
    SimpleObjectPublisher<Object[]> anotherPublisher = new
SimpleObjectPublisher<Object[]>();
    anotherPublisher.subscribe(rsi.subscriber());
    publisher.accept(new Object[] { 1, "John Doe", "North" });
    publisher.accept(new Object[] { 2, "Jane Doe", "North" });
    publisher.accept(new Object[] { 3, "John Smith", "South" });
    anotherPublisher.accept(new Object[] { 4, "John Doe", "North" });
    anotherPublisher.accept(new Object[] { 5, "Jane Doe", "North" });
    anotherPublisher.accept(new Object[] { 6, "John Smith", "South" });
    rsi.close();
    workerThreadPool.shutdown();
```

```
}
}
class SimpleObjectPublisher<T> implements Publisher<T>, Consumer<T> {
  Subscriber<? super T> subscriber;
  Subscription subscription = new SimpleObjectSubscription();
 //Data streaming starts
  @Override
 public void subscribe(Subscriber<? super T> subscriber) {
    this.subscriber = subscriber;
    this.subscriber.onSubscribe(subscription);
  }
 @Override
 public void accept(T t) {
    subscriber.onNext(t);
  }
}
 // You must provide this subscription
class SimpleObjectSubscription implements Subscription {
  @Override
 public void request(long n) {
    System.out.println("Library requesting: " + n + " records");
  }
 @Override
 public void cancel() {
    // TODO Auto-generated method stub
  }
}
```

24.4.3 Flow.Publisher Third-Party implementations

The following example shows how to use the RSI library when a third-party implements the Flow.Publisher interface.

In the following example, the standard JDK SubmissionPublisher interface is the third-party interface.

Example 24-3 Flow.Publisher Third-Party Implementations

```
package oracle.rsi.demos;
import java.sql.SQLException;
import java.time.Duration;
```



```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.SubmissionPublisher;
import oracle.rsi.ReactiveStreamsIngestion;
public class SimpleSubmissionPublisher {
  public static void main(String[] args) throws SQLException {
    ExecutorService workerThreadPool = Executors.newFixedThreadPool(2);
    ReactiveStreamsIngestion rsi = ReactiveStreamsIngestion
        .builder()
        .url(
            "jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp))
(HOST=example.com) (PORT=5521))
(CONNECT DATA=(SERVICE NAME=myservice.com)))")
        .username(<user name>)
        .password(<password>)
        .schema (<schema name>)
        .executor(workerThreadPool)
        .bufferRows(10)
        .bufferInterval(Duration.ofSeconds(20))
        .table("customers")
        .columns(new String[] { "id", "name", "region" })
        .build();
    SubmissionPublisher<Object[]> publisher = new
SubmissionPublisher<>();
    publisher.subscribe(rsi.subscriber());
    publisher.submit(new Object[] { 1, "John Doe", "North" });
    publisher.submit(new Object[] { 2, "Jane Doe", "North" });
    publisher.submit(new Object[] { 3, "John Smith", "South" });
    while (publisher.estimateMaximumLag() > 0);
    try {
      publisher.close();
    } catch (Exception e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
    rsi.close();
    workerThreadPool.shutdown();
}
```



24.5 Limitations of Java library for Reactive Streams Ingestion

Reactive streams ingestion of data streams may not ingest all the data into the database in case of unexpected crashes or errors because the library may lose some of the records while trying to store them in the database.

Apart from the possibility of data loss due to database crashes, this library has the following limitations:

- It does not support database triggers.
- It does not check referential integrity.
- It does not support clustered tables.
- It does not support loading of remote objects.
- It does not support loading of VARRAY columns.
- It does not support LONG data type with streams.
- It expects the LONG data type to be the last column in the database table.
- It expects all partitioning columns to appear before any column with LOB data type.



25 Support for Pipelined Database Operations

Java applications can now asynchronously submit several SQL requests to the server without waiting for the return of the preceding calls.

This chapter discusses more about pipelining in the following topics:

- Overview of Pipelining
- JDBC Support for Pipelining
- Pipelining with Reactive Extensions
- Pipelining with Java library for Reactive Streams Ingestion

25.1 Overview of Pipelining

Pipelining is a form of network communication in which an application can send multiple requests to a server, without having to wait for a response.

The fundamental idea of pipelining is to keep the server busy and enable an application to use the interleaving requests appropriately. The application can keep sending requests, while the server builds up a queue and processes those requests one by one. Then, the server sends the responses back to the client in the same order in which it received the requests.

As the pipeline does not read responses very often, using the data from a previous SQL response as in-bind data to the subsequent request creates a dependency and breaks the pipeline. So, the application must ensure that the requests are independent because it is an essential requirement for the pipelining functionality.

Pipelining offers the following benefits to your applications:

- Improved response time and throughput
- Improved scalability due to reduced context switching

25.2 JDBC Support for Pipelining

In the previous releases, the JDBC driver did not allow a new database call to start until the current call had been completed. Starting with Oracle Database Release 23c, JDBC Thin drivers now support pipelined database operations.

As a pipeline is executed by sending multiple requests without waiting for a response, this feature translates to an asynchronous programming model, in which the execution of multiple SQL statements can begin even before a thread consumes the results of those statements.

You can implement pipelining in your JDBC applications, using the following asynchronous programming features of Oracle Database:



Note:

When calls are made to a reactive (asynchronous) API or a standard JDBC batching API, pipelining is enabled by default.

- Pipelining with Reactive Extensions
- Pipelining with Java library for Reactive Streams Ingestion

25.3 Pipelining with Reactive Extensions

Starting from Oracle Database Relese 23c, you can use the Reactive Extensions APIs to carry out pipelined database operations.

The following code provides a simplified example of how you can use the Reactive Extensions APIs to carry out a pipeline:

In the following example, a call to the unwrap(OraclePreparedStatement.class) method is used to access the executeUpdateAsyncOracle and executeQueryAsyncOracle methods. These methods return a Flow.Publisher that implements the Reactive Streams specification. As the result of each SQL statement is received, the corresponding Publisher signals that result to a Subscriber.

```
void pipelineExample(OracleConnection connection) throws SQLException {
 // Prepare statements to execute
 PreparedStatement delete = connection.prepareStatement(
    "DELETE FROM example WHERE id = 0");
 PreparedStatement insert = connection.prepareStatement(
    "INSERT INTO example (id, value) VALUES (1, 'x')");
  PreparedStatement select = connection.prepareStatement(
    "SELECT id, value FROM example WHERE id = 2");
 // Execute statements in a pipeline
 Flow.Publisher<Long> deletePublisher =
   delete.unwrap(OraclePreparedStatement.class)
      .executeUpdateAsyncOracle();
 Flow.Publisher<Long> insertPublisher =
    insert.unwrap(OraclePreparedStatement.class)
      .executeUpdateAsyncOracle();
 Flow.Publisher<OracleResultSet> selectPublisher =
    select.unwrap(OraclePreparedStatement.class)
      .executeQueryAsyncOracle();
. . .
}
```



25.4 Pipelining with Java library for Reactive Streams Ingestion

Starting from Oracle Database Relese 23c, you can use the Java library for Reactive Streams Ingestion to carry out pipelined database operations.

The following code provides a simplified example of how you can use the Java library for Reactive Streams Ingestion to carry out a pipeline:

In the following example, instances of Mono and Flux publishers are created by calling their from methods. The argument to the method is an org.reactivestreams.Publisher, which is adapted from a Flow.Publisher, by calling

org.reactivestreams.FlowAdapters.toPublisher. A subscribe method is called to asynchronously consume results with a callback function. To consume a result of multiple rows, a call to Mono.flatMapMany converts a stream of a single ResultSet into a stream of multiple row values.

```
import oracle.jdbc.OracleConnection;
import oracle.jdbc.OraclePreparedStatement;
import oracle.jdbc.OracleResultSet;
import org.reactivestreams.FlowAdapters;
import org.reactivestreams.Publisher;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.concurrent.CompletionException;
import java.util.concurrent.Flow;
public class ReactiveExample {
  void reactivePipelineExample(OracleConnection connection) throws
SQLException {
    // Push a DELETE into the pipeline
    Mono.using(
        () -> connection.prepareStatement("DELETE FROM example WHERE id =
1"),
        preparedStatement -> Mono.from(publishUpdate(preparedStatement)),
        preparedStatement -> close(preparedStatement))
      .subscribe(deleteCount ->
        System.out.println(deleteCount + " rows deleted"));
    // Push an INSERT operation into the pipeline
    Mono.using(
        () -> connection.prepareStatement(
          "INSERT INTO example (id, value) VALUES (1, 'x')"),
        preparedStatement -> Mono.from(publishUpdate(preparedStatement)),
        preparedStatement -> close(preparedStatement))
      .subscribe(insertCount ->
        System.out.println(insertCount + " rows inserted"));
    // Push a SELECT into the pipeline
```

```
Flux.using(
        () -> connection.prepareStatement(
          "SELECT id, value FROM example ORDER BY id"),
        preparedStatement ->
          Mono.from(publishQuery(preparedStatement))
            .flatMapMany(resultSet -> publishRows(resultSet)),
        preparedStatement -> close(preparedStatement))
      .subscribe(rowString ->
        System.out.println(rowString));
  Publisher<Long> publishUpdate(PreparedStatement preparedStatement) {
    try {
      Flow.Publisher<Long> updatePublisher =
        preparedStatement.unwrap(OraclePreparedStatement.class)
          .executeUpdateAsyncOracle();
      return FlowAdapters.toPublisher(updatePublisher);
    }
    catch (SQLException sqlException) {
      return Mono.error(sqlException);
    }
  }
  Publisher<OracleResultSet> publishQuery(PreparedStatement
preparedStatement) {
    try {
      Flow.Publisher<OracleResultSet> queryPublisher =
        preparedStatement.unwrap(OraclePreparedStatement.class)
          .executeQueryAsyncOracle();
      return FlowAdapters.toPublisher(queryPublisher);
    }
    catch (SQLException sqlException) {
      return Mono.error(sqlException);
    }
  }
  Publisher<String> publishRows(ResultSet resultSet) {
    try {
      Flow.Publisher<String> rowPublisher =
        resultSet.unwrap(OracleResultSet.class)
          .publisherOracle(row -> {
            try {
              return String.format("id: %d, value: %s\n",
                row.getObject("id", Long.class),
                row.getObject("value", String.class));
            }
            catch (SQLException sqlException) {
              throw new CompletionException(sqlException);
            }
          });
      return FlowAdapters.toPublisher(rowPublisher);
    catch (SQLException sqlException) {
```

```
return Flux.error(sqlException);
}
void close(PreparedStatement preparedStatement) {
   try {
     preparedStatement.close();
   }
   catch (SQLException sqlException) {
     throw new RuntimeException(sqlException);
   }
}
```

26 OCI Connection Pooling

The Java Database Connectivity (JDBC) Oracle Call Interface (OCI) driver connection pooling functionality is part of the JDBC client. This functionality is provided by the OracleOCIConnectionPool class.

A JDBC application can have multiple pools at the same time. Multiple pools can correspond to multiple application servers or pools to different data sources. The connection pooling provided by the JDBC OCI driver enables applications to have multiple logical connections, all using a small set of physical connections. Each call on a logical connection gets routed on to the physical connection that is available at the time of call.

This chapter contains the following sections:

- Background of OCI Driver Connection Pooling
- Comparison Between OCI Driver Connection Pooling and Shared Servers
- About Defining an OCI Connection Pool
- About Connecting to an OCI Connection Pool
- Sample Code for OCI Connection Pooling
- Statement Handling and Caching
- JNDI and the OCI Connection Pool

Note:

Use OCI connection pooling if you need session multiplexing. Otherwise, Oracle recommends using Universal Connection Pool.

26.1 Background of OCI Driver Connection Pooling

The Oracle JDBC OCI driver provides several transaction monitor capabilities, such as the fine-grained management of Oracle sessions and connections. It is possible for a high-end application server or transaction monitor to multiplex several sessions over fewer physical connections on a call-level basis, thereby achieving a high degree of scalability by pooling of connections and back-end Oracle server processes.

The connection pooling provided by the OracleOCIConnectionPool interface simplifies the session/connection separation interface hiding the management of the physical connection pool. The Oracle sessions are the OracleOCIConnection objects obtained from OracleOCIConnectionPool. The connection pool itself is usually configured with a much smaller shared pool of physical connections, translating to a back-end server pool containing an identical number of dedicated server processes. Note that many more Oracle sessions can be multiplexed over this pool of fewer shared connections and back-end Oracle processes.



26.2 Comparison Between OCI Driver Connection Pooling and Shared Servers

In some ways, what OCI driver connection pooling offers on the middle tier is similar to what shared server processes offer on the back end. OCI driver connection pooling makes a dedicated server instance behaves as a shared instance by managing the session multiplexing logic on the middle tier. Therefore, the pooling of dedicated server processes and incoming connections into the dedicated server processes is controlled by the OCI connection pool on the middle tier.

The main difference between OCI connection pooling and shared servers is that in the case of shared servers, the connection from the client is typically to a dispatcher in the database instance. The dispatcher is responsible for directing the client request to an appropriate shared server. On the other hand, the physical connection from the OCI connection pool is established directly from the middle tier to the Oracle dedicated server process in the back-end server pool.

Note that OCI connection pool is mainly beneficial only if the middle tier is multithreaded. Each thread could maintain a session to the database. The actual connections to the database are maintained by <code>OracleOCIConnectionPool</code>, and these connections, including the pool of dedicated database server processes, are shared among all the threads in the middle tier.

26.3 About Defining an OCI Connection Pool

This section describes the following concepts:

- Overview of Creating an OCI Connection Pool
- Importing the oracle.jdbc.pool and oracle.jdbc.oci Packages
- Creating an OCI Connection Pool
- Setting the OCI Connection Pool Parameters
- Checking the OCI Connection Pool Status

26.3.1 Overview of Creating an OCI Connection Pool

An OCI connection pool is created at the beginning of the application. Creating connections from a pool is quite similar to creating connections using the <code>OracleDataSource class</code>.

The oracle.jdbc.pool.OracleOCIConnectionPool class, which extends the OracleDataSource class, is used to create OCI connection pools. From an OracleOCIConnectionPool instance, you can obtain logical connection objects. These connection objects are of the OracleOCIConnection class type. This class implements the OracleConnection interface. The Statement objects you create from the OracleOCIConnection instance have the same fields and methods as OracleStatement objects you create from OracleConnection instances.

The following code shows header information for the OracleOCIConnectionPool class:

/* * @param us ConnectionPool user-id.



```
* @param p ConnectionPool password
   * @param name logical name of the pool. This needs to be one in the
                      tnsnames.ora configuration file.
    @param config (optional) Properties of the pool, if the default does not
               suffice. Default connection configuration is min =1, max=1,
               incr=0
                  Please refer setPoolConfig for property names.
                Since this is optional, pass null if the default configuration
                 suffices.
   * @return
   * Notes: Choose a userid and password that can act as proxy for the users
           in the getProxyConnection() method.
           If config is null, then the following default values will take
           effect.
           CONNPOOL MIN LIMIT = 1
           CONNPOOL MAX LIMIT = 1
           CONNPOOL INCREMENT = 0
*/
public synchronized OracleOCIConnectionPool
  (String
           user,
                      String password, String name, Properties config)
  throws SQLException
/*
 * This will use the user-id, password and connection pool name values set
  LATER using the methods setUser, setPassword, setConnectionPoolName.
 * @return
 * Notes:
    No OracleOCIConnection objects can be created on
    this class unless the methods setUser, setPassword, setPoolConfig
     are invoked.
    When invoking the setUser, setPassword later, choose a userid and
    password that can act as proxy for the users
    in the getProxyConnection() method.
 */
 public synchronized OracleOCIConnectionPool ()
```

26.3.2 Importing the oracle.jdbc.pool and oracle.jdbc.oci Packages

Before you create an OCI connection pool, import the following to have Oracle OCI connection pooling functionality:

```
import oracle.jdbc.pool.*;
import oracle.jdbc.oci.*;
```

throws SQLException

26.3.3 Creating an OCI Connection Pool

The following code show how you create an instance of the OracleOCIConnectionPool class called cpool:



poolConfig is a set of properties that specify the connection pool. If poolConfig is null, then the default values are used. For example, consider the following:

- poolConfig.put (OracleOCIConnectionPool.CONNPOOL MIN LIMIT, "4");
- poolConfig.put (OracleOCIConnectionPool.CONNPOOL MAX LIMIT, "10");
- poolConfig.put (OracleOCIConnectionPool.CONNPOOL INCREMENT, "2");

As an alternative to the constructor call, you can create an instance of the OracleOCIConnectionPool class using individual methods to specify the user, password, and connection string.

26.3.4 Setting the OCI Connection Pool Parameters

The connection pool configuration is determined by the following OracleOCIConnectionPool class attributes:

CONNPOOL MIN LIMIT

Specifies the minimum number of physical connections that can be maintained by the pool.

CONNPOOL_MAX_LIMIT

Specifies the maximum number of physical connections that can be maintained by the pool.

• CONNPOOL INCREMENT

Specifies the incremental number of physical connections to be opened when all the existing ones are busy and a call needs one more connection; the increment is done only when the total number of open physical connections is less than the maximum number that can be opened in that pool.

• CONNPOOL TIMEOUT

Specifies how much time must pass before an idle physical connection is disconnected; this does not affect a logical connection.

• CONNPOOL NOWAIT

Specifies, if enabled, that an error is returned if a call needs a physical connection while the maximum number of connections in the pool are busy. If disabled, a call waits until a connection is available. Once this attribute is set to true, it cannot be reset to false.

You can configure all of these attributes dynamically. Therefore, an application has the flexibility of reading the current load, that is number of open connections and number

of busy connections, and adjusting these attributes appropriately, using the setPoolConfig method.

```
Note:
The default values for the CONNPOOL_MIN_LIMIT, CONNPOOL_MAX_LIMIT, and
CONNPOOL_INCREMENT parameters are 1, 1, and 0, respectively.
```

The setPoolConfig method is used to configure OCI connection pool properties. The following is a typical example of how the OracleOCIConnectionPool class attributes can be set:

```
java.util.Properties p = new java.util.Properties();
p.put (OracleOCIConnectionPool.CONNPOOL_MIN_LIMIT, "1");
p.put (OracleOCIConnectionPool.CONNPOOL_MAX_LIMIT, "5");
p.put (OracleOCIConnectionPool.CONNPOOL_INCREMENT, "2");
p.put (OracleOCIConnectionPool.CONNPOOL_TIMEOUT, "10");
p.put (OracleOCIConnectionPool.CONNPOOL_NOWAIT, "true");
cpool.setPoolConfig(p);
...
```

Observe the following rules when setting these attributes:

- CONNPOOL MIN LIMIT, CONNPOOL MAX LIMIT, and CONNPOOL INCREMENT are mandatory.
- CONNPOOL MIN LIMIT must be a value greater than zero.
- CONNPOOL_MAX_LIMIT must be a value greater than or equal to CONNPOOL_MIN_LIMIT plus CONNPOOL INCREMENT.
- CONNPOOL INCREMENT must be a value greater than or equal to zero.
- CONNPOOL TIMEOUT must be a value greater than zero.
- CONNPOOL_NOWAIT must be true or false.

See Also: Oracle Call Interface Programmer's Guide

26.3.5 Checking the OCI Connection Pool Status

To check the status of the connection pool, use the following methods from the OracleOCIConnectionPool class:

• int getMinLimit()

. . .

Retrieves the minimum number of physical connections that can be maintained by the pool.

int getMaxLimit()



Retrieves the maximum number of physical connections that can be maintained by the pool.

int getConnectionIncrement()

Retrieves the incremental number of physical connections to be opened when all the existing ones are busy and a call needs a connection.

int getTimeout()

Retrieves the specified time (in seconds) that a physical connection in a pool can remain idle before it is disconnected; the age of a connection is based on the Least Recently Used (LRU) algorithm.

String getNoWait()

Retrieves if the NOWAIT property is enabled. It returns a string of "true" or "false".

int getPoolSize()

Retrieves the number of physical connections that are open. This should be used only as an estimate and for statistical analysis.

int getActiveSize()

Retrieves the number of physical connections that are open and busy. This should be used only as an estimate and for statistical analysis.

• boolean isPoolCreated()

Retrieves if the pool has been created. The pool is actually created when OracleOCIConnection(user, password, url, poolConfig) is called or when setUser, setPassword, and setURL has been done after calling OracleOCIConnection().

26.4 About Connecting to an OCI Connection Pool

The OracleOCIConnectionPool class, through a getConnection method call, creates an instance of the OracleOCIConnection class. This instance represents a connection.

Because the OracleOCIConnection class extends OracleConnection class, it has the functionality of this class too. Close the OracleOCIConnection objects once the user session is over, otherwise, they are closed when the pool instance is closed.

There are two ways of calling getConnection:

OracleConnection getConnection()

If you do not supply the user name and password, then the default user name and password used for the creation of the connection pool are used while creating the connection objects.

OracleConnection getConnection(String user, String password)

If you this method, you will get a logical connection identified with the specified user name and password, which can be different from that used for pool creation.

The following code shows the signatures of the overloaded getConnection method:

/*



```
* For getting a connection to the database.
*
* @param us Connection user-id
* @param p Connection password
* @return connection object
*/
public synchronized OracleConnection getConnection(String us, String p)
throws SQLException
```

As an enhancement to OracleConnection, the following new method is added into OracleOCIConnection as a way to change the password for the user:

void passwordChange (String user, String oldPassword, String newPassword)

26.5 Sample Code for OCI Connection Pooling

The following code illustrates the use of OCI connection pooling in a sample application:

```
import java.sql.DriverManager;
import java.sgl.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Properties;
import oracle.jdbc.OracleDriver;
import oracle.jdbc.pool.OracleOCIConnectionPool;
public class conPoolAppl extends Thread
{
  public static final String query = "SELECT object name FROM all objects WHERE rownum
< 300";
  static public void main(String args[]) throws SQLException
  {
    int maxCount = 10;
    Connection []conn = new Connection[ maxCount];
    try
    {
      String s = null; //System.getProperty ("JDBC URL");
      String url = "jdbc:oracle:oci8:@localhost";
      OracleOCIConnectionPool cpool = new OracleOCIConnectionPool("HR", "hr", url,
null);
      // Print out the default configuration for the OracleOCIConnectionPool
      System.out.println ("-- The default configuration for the
OracleOCIConnectionPool --");
      displayPoolConfig(cpool);
      //Set up the initial pool configuration
      Properties p1 = new Properties();
      pl.put (OracleOCIConnectionPool.CONNPOOL MIN LIMIT, Integer.toString(1));
      pl.put (OracleOCIConnectionPool.CONNPOOL MAX LIMIT, Integer.toString( maxCount));
      p1.put (OracleOCIConnectionPool.CONNPOOL INCREMENT, Integer.toString(1));
      // Enable the initial configuration
      cpool.setPoolConfig(p1);
      Thread []t = new Thread[_maxCount];
      for (int i = 0; i < maxCount; ++i)</pre>
        conn[i] = cpool.getConnection("HR", "hr");
```

```
if ( conn[i] == null )
      {
        System.out.println("Unable to create connection.");
        return;
      }
      t[i] = new conPoolAppl (i, conn[i]);
      t[i].start ();
      //displayPoolConfig(cpool);
    }
    ((conPoolAppl)t[0]).startAllThreads ();
    try
    {
     Thread.sleep (200);
    }
   catch (Exception ea) {}
   displayPoolConfig(cpool);
   for (int i = 0; i < _maxCount; ++i)</pre>
     t[i].join ();
  }
  catch(Exception ex)
  {
   System.out.println("Error: " + ex);
   ex.printStackTrace ();
   return;
  }
  finally
  {
   for (int i = 0; i < _maxCount; ++i)</pre>
      if (conn[i] != null)
      conn[i].close ();
  }
} //end of main
private Connection m conn;
private static boolean m startThread = false;
private int m threadId;
public conPoolAppl (int i, Connection conn)
{
 m threadId = i;
 m conn = conn;
}
public void startAllThreads ()
{
 m startThread = true;
}
public void run ()
 while (!m_startThread) Thread.yield ();
  try
  {
   doQuery (m_conn);
  }
  catch (SQLException ea)
  {
   System.out.println ("*** Thread id: " + m_threadId);
   ea.printStackTrace ();
```



```
}
 } // end of run
 private static void doQuery (Connection conn) throws SQLException
 {
   PreparedStatement pstmt = null;
   ResultSet rs = null;
    try
    {
     pstmt = conn.prepareStatement (query);
     rs = pstmt.executeQuery ();
     while (rs.next ())
      {
        //System.out.println ("Object name: " +rs.getString (1));
     }
    }
    catch (Exception ea)
    {
     System.out.println ("Error during execution: " +ea);
     ea.printStackTrace ();
    }
    finally
    {
     if (rs != null)
       rs.close ();
     if (pstmt != null)
       pstmt.close ();
     if (conn != null)
       conn.close ();
   }
 } // end of doQuery (Connection)
 // Display the current status of the OracleOCIConnectionPool
 private static void displayPoolConfig (OracleOCIConnectionPool cpool) throws
SQLException
 {
    System.out.println (" Min poolsize Limit: " + cpool.getMinLimit());
   System.out.println (" Max poolsize Limit: " + cpool.getMaxLimit());
   System.out.println (" Connection Increment: " + cpool.getConnectionIncrement());
   System.out.println (" NoWait: " + cpool.getNoWait());
   System.out.println (" Timeout: " + cpool.getTimeout());
 */
   System.out.println (" PoolSize: " + cpool.getPoolSize());
    System.out.println (" ActiveSize: " + cpool.getActiveSize());
 }
} // end of class conPoolAppl
```

26.6 Statement Handling and Caching

Statement caching is supported with <code>OracleOCIConnectionPool</code>. The caching improves performance by not having to open, parse, and close cursors. When <code>OracleOCIConnection.prepareStatement ("a_SQL_query")</code> is processed, the statement cache is searched for a statement that matches the SQL query. If a match is found, then you can reuse the <code>Statement</code> object instead of incurring the cost of creating another <code>Statement</code> object. The cache size can be dynamically increased or decreased. The default cache size is zero.

Note:

The OracleStatement object created from OracleOCIConnection has the same behavior as one that is created from OracleConnection.

26.7 JNDI and the OCI Connection Pool

The Java Naming and Directory Interface (JNDI) feature makes the properties of a Java object persist, therefore these properties can be used to construct a new instance of the object, such as cloning the object. The benefit is that the old object can be freed, and at a later time a new object with exactly the same properties can be created. The InitialContext.bind method makes the properties persist, either on file or in a database, while the InitialContext.lookup method retrieves the properties from the persistent store and creates a new object with these properties.

OracleOCIConnectionPool objects can be bound and looked up using the JNDI feature. No new interface calls in OracleOCIConnectionPool are necessary.



27 Database Resident Connection Pooling

Database Resident Connection Pool (DRCP) is a connection pool in the server, which many clients can share. You should use DRCP when the number of active connections, at a given point of time, is reasonably less than the number of open connections. DRCP is particularly useful for applications with a large number of middle-tier servers. The Database Resident Connection Pooling feature increases Database server scalability and resolves the resource wastage issues of middle-tier connection pooling.

This chapter contains the following sections:

- Overview of Database Resident Connection Pooling
- Enabling Database Resident Connection Pooling
- Pooled Server Processes Across Multiple Connection Pools
- Multi-Pool Support in DRCP
- Tagging Support in Database Resident Connection Pooling
- APIs for Using Database Resident Connection Pooling

27.1 Overview of Database Resident Connection Pooling

In middle-tier connection pools, every connection pool maintains a minimum number of open connections to the server. Each open connection represents resources that are in use at the server. However, your application does not use all the open connections at a given point of time, which means that there are unused resources that consume server resources unnecessarily. In a multiple middle-tier scenario, every middle tier maintains an individual connection pool, without sharing connections with other middle-tier connection pools, and retains the open connections for long, even if some of those are inactive. For an application with a large number of middle-tier connection pools, the number of inactive connections to the Database server is significantly large in such case, which wastes a lot of Database resources.

For example, if in your application, the minimum pool size is 200 for every middle-tier connection pool, the connection pool has 200 open connections to the server, and the Database server has 200 server processes associated with these connections. If you have 30 similar middle-tier connection pools in your application, then the server has 6000 (200 * 30) corresponding server processes running at any given point of time. Typically, on an average only 5% of the connections, and in turn, server processes are in use simultaneously. So, out of the 6,000 server processes, only 300 server processes are active at any given time. This leads to 5,700 unused processes on the server, resulting in wasted resources on the server.

The Database Resident Connection Pool implementation creates a pool on the Database server side, which is shared across multiple client pools. This significantly lowers memory consumption on the Database server because of reduced number of server processes running simultaneously and increases its scalability.



See Also:

- Oracle Database Concepts
- Oracle Database Administrator's Guide

27.2 Enabling Database Resident Connection Pooling

This section describes how to enable Database Resident Connection Pooling (DRCP) on the server side and the client side:

- Enabling DRCP on the Server Side
- Enabling DRCP on the Client Side

27.2.1 Enabling DRCP on the Server Side

You must be a database administrator (DBA) and must log on as SYSDBA to start and end a Database Resident Connection Pool (DRCP). This section discusses the following related concepts:

- Starting a Database Resident Connection Pool
- Configuring a Database Resident Connection Pool
- Ending a Database Resident Connection Pool
- Setting the Statement Cache Size

Note:

You can leverage the DRCP features only with a connection pool on the client because JDBC does not have a default pool on its own.

Starting a Database Resident Connection Pool

Run the start_pool method in the DBMS_CONNECTION_POOL package with the default settings to start the default Oracle Database resident connection pool, namely, SYS DEFAULT CONNECTION POOL. For example:

sqlplus /nolog connect / as sysdba execute dbms_connection_pool.start_pool();

Configuring a Database Resident Connection Pool

Use the procedures in the DBMS_CONNECTION_POOL package to configure the default Oracle Database resident connection pool. By default, this connection pool uses default parameter values.



🖍 See Also:

- DBMS_CONNECTION_POOL
- Configuring Database Resident Connection Pooling

Ending a Database Resident Connection Pool

Run the stop_pool method in the DBMS_CONNECTION_POOL package to end the pool. For example:

```
sqlplus /nolog
connect / as sysdba
execute dbms connection pool.stop pool();
```

Setting the Statement Cache Size

If you use DRCP, then the server caches statement information on the server side. So, you must specify the statement cache size on the server side in the following way, where 50 is the preferred size:

execute DBMS_CONNECTION_POOL.CONFIGURE_POOL (session_cached_cursors=>50);

Related Topics

About Statement Caching

27.2.2 Enabling DRCP on the Client Side

Perform the following steps to enable DRCP on the client side:

Note: The example in this section uses Universal Connection Pool (UCP) as the clientside connection pool. For any other connection pool, you must use oracle.jdbc.pool.OracleConnectionPoolDataSource as the connection factory. Pass a non-null and non-empty String value to the oracle.jdbc.DRCPConnectionClass connection property Append (SERVER=POOLED) to the CONNECT_DATA configuration parameter in the long connection string You can also specify (*service_name>*=POOLED) in the short connection string in the following way: jdbc:oracle:thin:@<host>:<port>/<service_name>[:POOLED] For example:

jdbc:oracle:thin:@localhost:5221/orcl:POOLED

The following code snippet shows how to enable DRCP on the client side:



Note:

In UCP, if you do not provide a connection class, then the connection pool name is used as the connection class name by default.

Example 27-1 Enabling DRCP on Client Side Using Universal Connection Pool

```
String url = "jdbc:oracle:thin:@localhost:5521/orcl:POOLED";
PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();
pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
// Set DataSource Property
pds.setUser("HR");
pds.setPassword("hr");
System.out.println ("Connecting to " + url);
pds.setURL(url);
pds.setURL(url);
pds.setConnectionPoolName("HR-Pool1");
pds.setMinPoolSize(2);
pds.setInitialPoolSize(3);
pds.setInitialPoolSize(2);
Properties prop = new Properties();
prop.put("oracle.jdbc.DRCPConnectionClass", "HR-Pool1");
pds.setConnectionProperties(prop);
```

27.3 Pooled Server Processes Across Multiple Connection Pools

You can set the same Database Resident Connection Pool (DRCP) connection class name for all the pooled server processes on the server and share those across multiple connection pools on the server. For setting the DRCP connection class name, use the oracle.jdbc.DRCPConnectionClass connection property.

27.4 Multi-Pool Support in DRCP

You can now develop applications that use multiple, named DRCP pools. This helps in avoiding situations, where connections from a few services can occupy all the pooled servers of DRCP, while the rest of the connections from the other services may need to wait for the availability of the pooled servers.

For making DRCP mark the connection against the appropriate pool, specify (POOL_NAME=<pool_name>) in the connection string, along with (SERVER=POOLED), as shown in the following example:

```
(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=<host>)(PORT=<port>))
(CONNECT DATA=(SERVER=POOLED)(POOL NAME=<pool name>)))
```

It is not mandatory to create a DRCP pool for all the services, a few services may have multi-pool DRCP and the rest of the services may share the per-PDB or CDB-wide DRCP. If you are a PDB administrator, then by connecting to a PDB, you can also perform administrative operations like creating and destroying the DRCP.

The following procedures have been added to the DBMS_CONNECTION_POOL package to add and remove pooled servers from the multi-pool DRCP:



ADD_POOL Procedure

This procedure adds a new pool to the multi-pool DRCP. For example:

```
exec dbms connection pool.add pool('mypool')
```

REMOVE_POOL Procedure

This procedure removes a new pool from the multi-pool DRCP. For example:

exec dbms connection pool.remove pool('mypool')

See Also:

- Oracle Database Development Guide
- Oracle Database PL/SQL Packages and Types Reference

27.5 Tagging Support in Database Resident Connection Pooling

Database Resident Connection Pooling (DRCP) provides an option to associate a server process with a particular tag name. You can apply a tag to a given connection and retrieve that tagged connection later. Connection tagging enhances session pooling because you can retrieve specific sessions easily.

DRCP also provides support for multiple tagging, which is disabled by default. Set the oracle.jdbc.UseDRCPMultipletag connection property to TRUE for enabling this feature in your DRCP application.

Once you enable the multiple tagging feature, you can use the same APIs for setting single or multiple DRCP tags. The only difference between both the cases is the separator. You must separate the key and the value of a DRCP tag by an equal (=) character, whereas, you must separate multiple tags from one another by a semi-colon (;) character.

Remember the following points while working with DRCP tags:

- You cannot specify the key or the value of a tag as null or empty.
- When you specify multiple tags, then the leftmost tag has the highest priority and the rightmost tag has the lowest priority.
- While retrieving a tagged connection, if a complete match is not found (all tags are not matched), then it searches for a partial match.

See Also:

Oracle Call Interface Programmer's Guide for more information about session pooling and connection tagging



27.6 PL/SQL Callback for Session State Fix Up

Starting from Oracle Database 12c Release 2 (12.2.0.1), a PL/SQL based fix-up callback for the session state can be provided on the server. This application-provided callback transforms a session checked out from the pool to the desired state requested by the application. This callback works with or without Database Resident Connection Pooling (DRCP).

Note:

The PL/SQL based fix-up callback is only applicable for multiple tagging.

Using this callback can improve the performance of your application because the fixup logic is run for the session state on the server. So, this feature eliminates application round-trips to the database for the fix-up logic. An appropriate installation user, who must be granted execute permissions on the related package, should register the fix-up callback during application installation.

Example 27-2 Example of PL/SQL Fix-Up Callback

Following is an example implementation of the PL/SQL fix up callback to fix up the session properties SCHEMA and CURRENCY:

```
CREATE OR REPLACE PACKAGE mycb pack AS
PROCEDURE mycallback (
desired_props IN VARCHAR2,
actual props IN VARCHAR2
);
END;
/
CREATE OR REPLACE PACKAGE BODY mycb pack AS
PROCEDURE mycallback (
desired props IN VARCHAR2,
actual props IN VARCHAR2
) IS
property VARCHAR2(64);
key VARCHAR2(64);
value VARCHAR2(64);
pos number;
pos2 number;
pos3 number;
idx1 number;
BEGIN
idx1:=1;
pos:=1;
pos2:=1;
pos3:=1;
```



```
property := 'tmp';
-- To check if desired properties are part of actual properties
while (pos > 0 and length(desired props)>pos)
loop
pos := instr (desired props, ';', 1, idx1);
if (pos=0)
then
property := substr (desired props, pos2);
else
property := substr (desired props, pos2, pos-pos2);
end if ;
pos2 := pos+1;
pos3 := instr (property, '=', 1, 1);
key := substr (property, 1, pos3-1);
value := substr (property, pos3+1);
if (key = 'CURRENCY') then
EXECUTE IMMEDIATE 'ALTER SESSION SET NLS CURRENCY=''' || value || '''';
elsif (key = 'SCHEMA') then
EXECUTE IMMEDIATE 'ALTER SESSION SET CURRENT SCHEMA=' || value;
end if;
idx1 := idx1+1;
end loop;
END; -- mycallback
END mycb pack;
```

See Also: Oracle Database JDBC Java API Reference

27.7 APIs for Using Database Resident Connection Pooling

If you want to take advantage of Database Resident Connection Pooling (DRCP) with higher granular control for your custom connection pool implementations, then you must use the following APIs declared in the oracle.jdbc.OracleConnection interfaces:

- attachServerConnection
- detachServerConnection
- isDRCPEnabled
- isDRCPMultitagEnabled
- getDRCPReturnTag
- needToPurgeStatementCache
- getDRCPState



See Also: Oracle Database JDBC Java API Reference

28 JDBC Support for Database Sharding

This section describes Oracle JDBC support for the Database Sharding feature.

- Overview of Database Sharding for JDBC Users
- About Building the Sharding Key
- APIs for Database Sharding Support
- JDBC Sharding Example
- Overview of the Sharding Data Source

28.1 Overview of Database Sharding for JDBC Users

Modern web applications face new scalability challenges with huge volumes of data. A commonly accepted solution to this problem is sharding. *Sharding* is a data tier architecture, where data is horizontally partitioned across independent databases. Each database in such a configuration is called a *shard*. All shards together make up a single logical database, which is referred to as a *sharded database* (*SDB*). Sharding is a *shared-nothing* database architecture because shards do not share physical resources such as CPU, memory, or storage devices.

Sharding uses Global Data Services (GDS), where GDS routes a client request to an appropriate database based on parameters such as availability, load, network latency, and replication lag. A GDS pool is a set of replicated databases that offer the same global service. The databases in a GDS pool can be located in multiple data centers across different regions. A sharded GDS pool contains all shards of a sharded database and their replicas, and appears as a single sharded database to database clients.

Starting from Oracle Database 12c Release 2 (12.2.0.1), Oracle JDBC supports database sharding. The JDBC driver recognizes the specified sharding key and super sharding key and connects to the relevant shard that contains the data. Once the connection is established to a shard, then any database operations, such as DMLs, SQL queries and so on, are supported and executed in the usual way. The following section describes the sharding terminologies used in this guide:

See Also:

Oracle Database Administrator's Guide

Sharding, Shard, and Sharded Database

Sharding is a data tier architecture where data is horizontally partitioned across independent databases. Each database in such configuration is called a *shard*. All shards together make up a single logical database which is referred to as a *sharded database (SDB)*.



Sharding Key, Composite Sharding Key, and Super Sharding Key

A sharding key is a partitioning key used in single-level sharding by range, list, or consistent hash. All sharding keys together are referred to as the composite sharding keys. A super-sharding key is the partitioning key used in composite sharding for the top-level sharding by range or list. Both the sharding key and the super sharding key can contain one or more columns that determine the shard where each row is stored. A sharding key can be of type VARCHAR2, CHAR, DATE, NUMBER, TIMESTAMP and so on.

For JDBC users, it is recommended that sharding keys and super sharding keys must be passed while obtaining connections from the database. However, Sharding Keys can be provided in the connection string as a separate attribute under CONNECT_DATA. Passing sharding key in the connection string restricts the connections only to one shard. So, it is not recommended to use this approach. Following code snippet shows how you can provide Sharding Keys as a separate attribute under CONNECT_DATA in the connection string:

```
(DESCRIPTION=(...) (CONNECT_DATA=(SERVICE_NAME=ORCL (SHARDING_KEY=...)
(SUPER SHARDING KEY=...)))
```

Note:

You must provide the sharding key compliant to the NLS formatting that is specified in the database.

Multi Shard Queries

Multi Shard Queries enable routing and processing of queries and transactions that access data stored on multiple shards. Multi Shard Queries are executed without a sharding key. Multi Shard Operations are used for simple aggregation of data and reporting across shards.

Shard Catalog

Shard Catalog is a special database that is used for storing sharded database and supporting multi shard queries. It also helps in centralized management of a sharded database.

Shard Director

A shard director is a specific implementation of a global service manager (GSM) that acts as a regional listener for clients that connect to an SDB and maintains a current topology map of the SDB. Based on the sharding key passed during a connection request, it routes the connections to the appropriate shard.

Shard Topology

Shard Topology is the sharding key range mappings stored in a particular shard. Universal Connection Pool (UCP) can cache shard topology, which enables it to bypass shard director while establishing connections to shards. So, applications that you built using UCP get fast path for shards.



See Also: Oracle Universal Connection Pool Developer's Guide

Chunk

A chunk is a single partition from each table of a table family. It is a unit of data migration between shards.

Chunk Split

Chunk Split is a process that is required when chunks become too big or only part of a chunk needs to be migrated to another shard.

Chunk Migration

Chunk migration is the process of moving a chunk from one shard to another, when data or workload skew occurs without any change in the number of shards. It is initiated by DBA to eliminate hot spots.

Resharding

Resharding is the process of redistributing data between shards triggered by a change in the number of shards. Chunks are moved between shards for even distribution of chunks across shards. However, content of chunks does not change, that is, no rehashing takes place during Resharding.

28.2 About Building the Sharding Key

The shard aware applications must identify and build the sharding key and the super sharding key, which are required to establish a connection to the sharded database. For achieving this, the shard aware applications must use the <code>OracleShardingKey</code> and the <code>OracleShardingKeyBuilder</code> interfaces.

The OracleShardingKeyBuilder uses the following builder method for supporting compound keys with different data types:

subkey(Object subkey, java.sql.SQLTYPE subkeyDataType)

There are multiple invocations of the subkey method on the builder for building a compound sharding key, where each subkey can be of different data types. The data type can be defined using the oracle.jdbc.OracleType enum or java.sql.JDBCType.

Example 28-1 Building a Sharding Key

The following example shows how to build a sharding key:

```
import java.sql.Connection;
import java.sql.Date;
import java.sql.SQLException;
import java.sql.Statement;
import oracle.jdbc.OracleShardingKey;
```



```
import oracle.jdbc.OracleType;
import oracle.ucp.jdbc.PoolDataSource;
import oracle.ucp.jdbc.PoolDataSourceFactory;
  public class ShardExample
  {
    public static void main(String[] args) throws SQLException
    {
      String url =
"jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(HOST=myhost)(PORT=3216)
(PROTOCOL=tcp)) (CONNECT DATA=(SERVICE NAME=myservice) (REGION=east)))";
      String user="testuser1";
      String pwd = "password";
      PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();
      pds.setURL(url);
      pds.setUser(user);
      pds.setPassword(pwd);
pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
      pds.setInitialPoolSize(5);
      pds.setMinPoolSize(5);
      pds.setMaxPoolSize(20);
      // build the sharding key object
      Date shardingKeyVal = new java.sql.Date(OL);
      OracleShardingKey sdkey = pds.createShardingKeyBuilder()
                                    .subkey(shardingKeyVal,
OracleType.DATE)
                                    .build();
      Connection conn = pds.createConnectionBuilder()
                             .shardingKey(sdkey)
                             .build();
      Statement stmt = conn.createStatement();
      stmt.execute("... SQL statement here ...");
      stmt.close();
      conn.close();
    }
  }
```

The following code snippet shows how to build a compound sharding key that consists of String and Date data types:



Note:

- There is a fixed set of data types that are valid and supported. If any unsupported data types are used as keys, then exceptions are thrown. The following list specifies the supported data types:
 - OracleType.VARCHAR2/JDBCType.VARCHAR
 - OracleType.CHAR/JDBCType.CHAR
 - OracleType.NVARCHAR/JDBCType.NVARCHAR
 - OracleType.NCHAR/JDBCType.NCHAR
 - OracleType.NUMBER/JDBCType.NUMERIC
 - OracleType.FLOAT/ JDBCType.FLOAT
 - OracleType.DATE/ JDBCType.DATE
 - OracleType.TIMESTAMP/JDBCType.TIMESTAMP
 - OracleType.TIMESTAMP_WITH_LOCAL_TIME_ZONE
 - OracleType.RAW
- You must provide a sharding key that is compliant to the NLS formatting specified in the database.

28.3 APIs for Database Sharding Support

Oracle Database 12c Release 2 (12.2.0.1) introduced a set of APIs for implementing database sharding. The following sections discuss these APIs in details:

- The OracleShardingKey Interface
- The OracleShardingKeyBuilder Interface
- The OracleConnectionBuilder Interface
- Other New Classes and Methods for Database Sharding Support

28.3.1 The OracleShardingKey Interface

This interface indicates that the current object represents an Oracle sharding key that is to be used with Oracle sharded database.

Syntax

public interface OracleShardingKey extends Comparable <OracleShardingKey>



28.3.2 The OracleShardingKeyBuilder Interface

OracleShardingKeyBuilder provides the interface to build the compound sharding key with subkeys of various supported data types. This interface uses the new JDK 8 builder pattern for building a sharding key.

Syntax

public interface OracleShardingKeyBuilder

Example 28-2 Creating the Sharding Key

OracleDataSource ods = new OracleDataSource(); ... //set datasource properties.. ... OracleShardingKey shardingKey = ods.createShardingKeyBuilder() .subkey("Customer_Name_XYZ", JDBCType.VARCHAR) .subkey(94002, JDBCType.NUMERIC) .build();

28.3.3 The OracleConnectionBuilder Interface

The OracleConnectionBuilder is used for building connection objects with additional parameters, other than user name and password. For creating a connection, the builder methods need to be called for each parameter that needs to be part of the connection request, followed by a build() method. The order in which the builder methods are called is not important. However, if the same builder attribute is applied more than once, then only the most recent value is considered while building the connection. The build() method of the builder can be called only once on a builder object.

Syntax

public interface OracleConnectionBuilder

Example 28-3 Creating the Connection Builder



28.3.4 Other New Classes and Methods for Database Sharding Support

This section describes the rest of the new classes and methods introduced for implementing database sharding support.

New Methods in OracleDataSource Class

The createConnectionBuilder and createShardingKeyBulider methods have been introduced in OracleDataSource class for database sharding support.

OracleConnectionBuilder createConnectionBuilder() throws SQLException; OracleShardingKeyBuilder createShardingKeyBuilder()

New Methods in OracleXADataSource Class

The createConnectionBuilder method has been introduced in OracleXADataSource class for database sharding support.

OracleConnectionBuilder createConnectionBuilder() throws SQLException;

New Methods in OracleConnection Class

The setShardingKeyIfValid and setShardingKey methods have been introduced in OracleConnection class for database sharding support.

boolean setShardingKeyIfValid(OracleShardingKey shardingKey, OracleShardingKey superShardingKey, int timeout) throws SQLException;

void setShardingKey(OracleShardingKey shardingKey, OracleShardingKey superShardingKey) throws SQLException;

New Methods in OracleXAConnection Class

The setShardingKeyIfValid and setShardingKey methods have been introduced in OracleConnection class for database sharding support.

boolean setShardingKeyIfValid(OracleShardingKey shardingKey, OracleShardingKey superShardingKey, int timeout) throws SQLException;

void setShardingKey(OracleShardingKey shardingKey, OracleShardingKey superShardingKey) throws SQLException;

28.4 JDBC Sharding Example

The following code snippet shows how to use JDBC sharding APIs:



Example 28-4 JDBC Sharding Example

```
OracleDataSource ods = new OracleDataSource();
      ods.setURL("jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(HOST=myhost))
(PORT=1521) (PROTOCOL=tcp))
(CONNECT DATA=(SERVICE NAME=myorcldbservicename)))");
      ods.setUser("hr");
      ods.setPassword("hr");
     // Employee name is the sharding Key in this example.
     // Build the Sharding Key using employee name as shown below.
      OracleShardingKey employeeNameShardKey =
ods.createShardingKeyBuilder()
                                                   .subkey("Mary",
JDBCType.VARCHAR) // First Name
                                                   .subkey("Claire",
JDBCType.VARCHAR) // Last Name
                                                   .build();
      OracleShardingKey locationSuperShardKey =
ods.createShardingKeyBuilder() // Building a super sharding key using
location as the key
                                                    .subkey("US",
JDBCType.VARCHAR)
                                                    .build();
      OracleConnection connection = ods.createConnectionBuilder()
                                        .shardingKey(employeeNameShardKe
y)
                                        .superShardingKey(locationSuperS
hardKey)
                                        .build();
```

28.5 Overview of the Sharding Data Source

Starting with Oracle Database Release 21c, the JDBC data source can establish Java connectivity to a sharded database without the need to furnish a sharding key. So, you do not need to identify and build the sharding key and the super sharding key to establish a connection.

The new sharding data source scales out to sharded databases transparently as it does not involve any change to the application code. When the sharding key can be derived from SQL or PL/SQL, the JDBC driver can identify it without the need for the application to send the sharding key. For using this data source, you must set the connection property oracle.jdbc.useShardingDriverConnection to true in the following way:

```
Properties prop = new Properties();
prop.setProperty("oracle.jdbc.useShardingDriverConnection", "true");
```

The default value of this connection property is false.



28.5.1 Benefits of the Sharding Data Source

Following are the benefits of the new sharding data source:

- You do not need to use the sharding APIs to pass the sharding key because the sharding data source derives the sharding key from the SQL statement.
- You do not need to configure the Universal Connection Pool (UCP) because the sharding data source uses the auto tune feature of UCP.
- You do not need to check-in or check-out a physical connection for every new sharding key because the sharding data source does it automatically.
- You do not need to separate cross-shard statements from single-shard statements and create separate connection pools for them because the sharding data source maintains those connections pools.
- The sharding data source enables the prepared statement caching and routes the connection to the direct shard based on the key used in the SQL statement.
- The sharding data source simplifies applications and optimizes application performance without any code change.

28.5.2 Example: How to Use the Sharding Data Source

The following examples show how to use the sharding data source.

```
Example 28-5 Using the Sharding Data Source
```

```
public class ShardingDriverSample {
  public static void main(String[] args) throws SQLException
{
    ShardingDriverSample sample = new ShardingDriverSample();
    sample.bindQuerySample();
  }
  private void bindQuerySample() throws SQLException {
    OracleConnection conn = getGsmConnection();
    executeQueryWithBindAndReadRows (conn, "SELECT * FROM MY SHARD TABLE
where ID = ?", 10);
    System.out.println("Direct shard execution percentage:" +
conn.getPercentageQueryExecutionOnDirectShard());
  private OracleConnection getGsmConnection() throws SQLException {
    OracleDataSource ds = new OracleDataSource();
    ds.setURL(qsmURL);
    Properties prop = new Properties();
    prop.setProperty("user", <userName>);
    prop.setProperty("password", <password>);
    // this property enables sharding datasource feature
    prop.setProperty("oracle.jdbc.useShardingDriverConnection", "true");
    ds.setConnectionProperties(prop);
    return ds.getConnection();
  }
```

private void executeQueryWithBindAndReadRows(Connection dbConnection,



```
String sql, int noOfTime)
    throws SQLException {
    while (noOfTime-- > 0) {
        PreparedStatement statement = dbConnection.prepareStatement(sql);
        statement.setInt(1, 15);
        ResultSet rs = statement.executeQuery();
        ...
    }
    }
}
```

28.5.3 Limitations of the Sharding Data Source

This section describes the limitations of the sharding data source.

- The sharding data source supports only the JDBC Thin driver. It does not support the JDBC OCI driver or the KPRB driver.
- The sharding data source does not support some Oracle JDBC extension APIs such as Direct Path Load, JDBC Dynamic Monitoring Service (DMS) metrics, and so on.
- The sharding data source supports PL/SQL execution only through the catalog database.
- When AUTO COMMIT is set to OFF, then the execution always happens on the catalog database.
- If the data source property singleShardTransactionSupport is set to TRUE, then the sharding data source supports local transactions against a single shard, when AUTO COMMIT is set to OFF.

The following code snippet shows how to set the singleShardTransactionSupport property:

```
private OracleDataSource createShardingDataSource() throws
SQLException {
```

OracleDataSource ds = new OracleDataSource();

Properties prop = new Properties();

prop.setProperty("user", <userName>);

prop.setProperty("password", <password>);

 $//\ {\rm Set}$ this connection property to enable sharding data source feature

prop.setProperty("oracle.jdbc.useShardingDriverConnection",
 "true");

ds.setConnectionProperties(prop);

ds.setURL(gsmURL);



```
\ensuremath{//} Set this data source property to enable single shard transaction support
```

```
ds.setSingleShardTransactionSupport(true);
return ds;
```

}

29 Oracle Advanced Queuing

Oracle Advanced Queuing (AQ) provides database-integrated message queuing functionality. It is built on top of Oracle Streams and optimizes the functions of Oracle Database so that messages can be stored persistently, propagated between queues on different computers and databases, and transmitted using Oracle Net Services, HTTP, and HTTPS. Because Oracle AQ is implemented in database tables, all operational benefits of high availability, scalability, and reliability are also applicable to queue data. This chapter provides information about the Java interface to Oracle AQ.

Note:

- Oracle Advanced Queuing (AQ) is a feature of the Oracle JDBC Thin driver and is not supported by JDBC OCI driver.
- In Oracle Database 12c Release 1 (12.1), support for XMLType queues has been added. Till Oracle Database 11g Release 1, supported queue types were RAW, ADT, and ANYDATA queue types.

See Also:

Oracle Database Advanced Queuing User's Guide

This chapters covers the following topics:

- Functionality and Framework of Oracle Advanced Queuing
- Making Changes to the Database
- AQ Asynchronous Event Notification
- About Creating Messages
- Enqueuing Messages
- Dequeuing Messages
- Examples: Enqueuing and Dequeuing

29.1 Functionality and Framework of Oracle Advanced Queuing

The Oracle JDBC package oracle.jdbc.aq provides a fast Java interface to AQ. This package contains the following:

- Classes
 - AQDequeueOptions



Specifies the options available for the dequeue operation

AQEnqueueOptions

Specifies the options available for the enqueue operation

AQFactory

Is a factory class for AQ

AQNotificationEvent

Is created whenever a new message is enqueued in a queue for which you have registered your interest

- Interfaces
 - AQAgent

Used to represent and identify a user of the queue or a producer or consumer of the message

AQMessage

Represents a message that is enqueued or dequeued

- AQMessageProperties

Contains message properties such as Correlation, Sender, Delay and Expiration, Recipients, and Priority and Ordering

- AQNotificationListener

Is a listener interface for receiving AQ notification events

AQNotificationRegistration

Represents your interest in being notified when a new message is enqueued in a particular queue

These classes and interfaces enable you to access an existing queue, create messages, and enqueue and dequeue messages.

Note:

Oracle JDBC drivers do *not* provide any API to create a queue. Queues must be created through the DBMS AQADM PL/SQL package.

See Also:

For more information about the APIs, refer to Oracle Database JDBC Java API Reference.

29.2 Making Changes to the Database

The code snippets used in this chapter assume that user HR is connecting to the database. Therefore, in the database, you must grant the following privileges to HR:



```
GRANT EXECUTE ON DBMS_AQ to HR;
GRANT EXECUTE ON DBMS_AQADM to HR;
GRANT AQ_ADMINISTRATOR_ROLE TO HR;
GRANT ADMINISTER DATABASE TRIGGER TO HR;
```

Before you start enqueuing and dequeuing messages, you must have queues in the Database. For this, you must perform the following:

1. Create a queue table in the following way:

```
BEGIN
DBMS_AQADM.CREATE_QUEUE_TABLE(
QUEUE_TABLE => 'HR.RAW_SINGLE_QUEUE_TABLE',
QUEUE_PAYLOAD_TYPE => 'RAW',
COMPATIBLE => '10.0');
END:
```

```
2. Create a queue in the following way:
```

```
BEGIN
DBMS_AQADM.CREATE_QUEUE(
QUEUE_NAME =>'HR.RAW_SINGLE_QUEUE',
QUEUE_TABLE =>'HR.RAW_SINGLE_QUEUE_TABLE',
END;
```

3. Start the queue in the following way:

```
BEGIN
DBMS_AQADM.START_QUEUE(
'HR.RAW_SINGLE_QUEUE',
END:
```

It is a good practice to stop the queue and remove the queue tables from the database. You can perform this in the following way:

1. Stop the queue in the following way:

```
BEGIN
DBMS_AQADM.STOP_QUEUE(
HR.RAW_SINGLE_QUEUE',
END;
```

2. Remove the queue tables from the database in the following way:

```
BEGIN
DBMS_AQADM.DROP_QUEUE_TABLE(
               QUEUE_TABLE => 'HR.RAW_SINGLE_QUEUE_TABLE',
               FORCE => TRUE
END;
```

29.3 AQ Asynchronous Event Notification

A JDBC application can do the following:

Register to the AQ namespace and receive notification when an enqueue occurs. This
can be performed in the following way:

```
public AQNotificationRegistration registerForAQEvents(
    OracleConnection conn,
    String queueName) throws SQLException
{
    Properties globalOptions = new Properties();
    String[] queueNameArr = new String[1];
```



```
queueNameArr[0] = queueName;
Properties[] opt = new Properties[1];
opt[0] = new Properties();
opt[0].setProperty(OracleConnection.NTF_AQ_PAYLOAD,"true");
AQNotificationRegistration[] regArr =
conn.registerAQNotification(queueNameArr,opt,globalOptions);
AQNotificationRegistration reg = regArr[0];
return reg;
}
```

 Register subscriptions to database events and receive notifications when the events are triggered

Registered clients are notified asynchronously when events are triggered or on an explicit AQ enqueue (or a new message is enqueued in a queue for which you have registered your interest). Clients do not need to be connected to a database.

The following code snippet shows how to subscribe to database events and receive notifications when the events are triggered:

```
class DemoAQRawQueueListener implements AQNotificationListener
{
 OracleConnection conn;
 String gueueName;
 String typeName;
 int eventsCount = 0;
 public DemoAQRawQueueListener(String queueName, String typeName)
   throws SQLException
  {
  queueName = _queueName;
   typeName = _typeName;
  conn = (OracleConnection)DriverManager.getConnection
     (DemoAQRawQueue.URL, DemoAQRawQueue.USERNAME, DemoAQRawQueue.PASSWORD);
  public void onAQNotification(AQNotificationEvent e)
  {
    try
    {
     AQDequeueOptions deqopt = new AQDequeueOptions();
     degopt.setRetrieveMessageId(true);
     if(e.getConsumerName() != null)
       degopt.setConsumerName(e.getConsumerName());
     if((e.getMessageProperties()).getDeliveryMode()
        == AQMessageProperties.DeliveryMode.BUFFERED)
     {
      deqopt.setDeliveryMode(AQDequeueOptions.DEQUEUE BUFFERED);
       deqopt.setVisibility(AQDequeueOptions.DEQUEUE IMMEDIATE);
     AQMessage msg = conn.dequeue(queueName, deqopt, typeName);
     byte[] msqId = msq.getMessageId();
     if(msqId != null)
       String mesgIdStr = DemoAQRawQueue.byteBufferToHexString(msgId,20);
       System.out.println("ID of message dequeued = "+mesgIdStr);
     System.out.println(msg.getMessageProperties().toString());
     byte[] payload = msg.getPayload();
     if(typeName.equals("RAW"))
       String payloadStr = new String(payload, 0, 10);
```

```
System.out.println("payload.length="+payload.length+", value="+payloadStr);
}
catch(SQLException sqlex)
{
System.out.println(sqlex.getMessage());
}
eventsCount++;
}
public int getEventsCount()
{
return eventsCount;
}
public void closeConnection() throws SQLException
{
conn.close();
}
```

Register to the listener in the following way:

```
AQNotificationRegistration reg = registerForAQEvents(conn,queueName+":BLUE");
DemoAQRawQueueListener demo_li = new DemoAQRawQueueListener(queueName,queueType);
reg.addListener(demo_li);
```

29.4 About Creating Messages

This section describes the following concepts:

- Creating Messages
- AQ Message Properties
- AQ Message Payload

29.4.1 Creating Messages

Before you enqueue a message, you must create the message. An instance of a class implementing the AQMessage interface represents an AQ message. An AQ message contains properties (metadata) and a payload (data). Perform the following to create an AQ message:

1. Create an instance of AQMessageProperties in the following way:

AQMessageProperties msgprop = AQFactory.createAQMessageProperties();

2. Set the property attributes in the following way:

```
msgprop.setCorrelation("mycorrelation");
msgprop.setExceptionQueue("MY_EXCEPTION_QUEUE");
msgprop.setExpiration(0);
msgprop.setPriority(1);
```

3. Create the AQ message using the AQMessageProperties object in the following way:

AQMessage mesg = AQFactory.createAQMessage(msgprop);

4. Set the payload in the following way:

```
byte[] rawPayload = "Example_Payload".getBytes();
mesg.setPayload(new oracle.sql.RAW(rawPayload));
```



29.4.2 AQ Message Properties

The properties of the AQ message are represented by an instance of the AQMessageProperties interface. You can set or get the following message properties:

- Dequeue Attempts Count: Specifies the number of attempts that have been made to dequeue the message. This property cannot be set.
- Correlation: Is an identifier supplied by the producer of the message at the time of enqueuing the message.
- Delay: Is the number of seconds for which the message is in the WAITING state. After the specified delay, the message is in the READY state and available for dequeuing. Dequeuing a message by using the message ID (msgid) overrides the delay specification.

Note:

Delay is not supported with buffered messaging.

- Delivery Mode: Specifies whether the message is a buffered message or a persistent message. This property cannot be set.
- Enqueue Time: Specifies the time at which the message was enqueued. This value is determined by the system and cannot be set by the user.
- Exception Queue: Specifies the name of the queue into which the message is moved if it cannot be processed successfully. Messages are moved in two cases:
 - The number of unsuccessful dequeue attempts has exceeded max retries.
 - The message has expired.
- Expiration: Is the number of seconds during which the message is available for dequeuing, starting from when the message reaches the READY state. If the message is not dequeued before it expires, then it is moved to the exception queue in the EXPIRED state.
- Message State: Specifies the state of the message at the time of dequeuing the message. This property cannot be set.
- Previous Queue Message ID: Is the ID of the message in the last queue that generated the current message. When a message is propagated from one queue to another, this attribute identifies the ID of the queue from which it was last propagated. This property cannot be set.
- Priority: Specifies the priority of the message. It can be any integer including negative integers; the smaller the value, the higher the priority.
- Recipient list: Is a list of AQAgent objects that represent the recipients. The default recipients are the queue subscribers. This parameter is valid only for multipleconsumer queues.
- Sender: Is an identifier specified by the producer at the time of enqueuing the message. It is an instance of AQAgent.
- Transaction group: Specifies the transaction group of the message for transactiongrouped queues. It is set after a successful call to the dequeueArray method.



29.4.3 AQ Message Payload

. . .

Depending on the type of the queue, the payload of the AQ message can be specified using the setPayload method of the AQMessage interface. The following code snippet illustrates how to set the payload:

```
byte[] rawPayload = "Example_Payload".getBytes();
mesg.setPayload(new oracle.sql.RAW(rawPayload));
...
```

You can retrieve the payload of an AQ message using the getPayload method or the appropriate getXXXPayload method in the following way:

```
byte[] payload = mesg.getPayload();
```

These methods are defined in the AQMessage interface.

29.5 Example: Creating a Message and Setting a Payload

This section provides an example that illustrates how to create a message and set a payload.

Example 29-1 Creating a Message and Setting a Payload

This example shows how to Create an instance of AQMessageProperties, set the property attributes, create the AQ message, and set the payload.

```
AQMessageProperties msgprop = AQFactory.createAQMessageProperties();
msgprop.setCorrelation("mycorrelation");
msgprop.setExceptionQueue("MY_EXCEPTION_QUEUE");
AQAgent ag = AQFactory.createAQAgent();
ag.setName("MY_SENDER_AGENT_NAME");
ag.setAddress("MY_SENDER_AGENT_ADDRESS");
msgprop.setSender(ag);
// handle multi consumer case:
if(recipients != null)
msgprop.setRecipientList(recipients);
System.out.println(msgprop.toString());
AQMessage mesg = AQFactory.createAQMessage(msgprop);
byte[] rawPayload = "Example_Payload".getBytes();
mesg.setPayload(new oracle.sql.RAW(rawPayload));
```

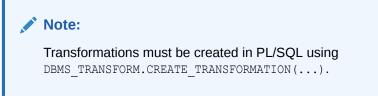
29.6 Enqueuing Messages

After you create a message and set the message properties and payload, you can enqueue the message using the enqueue method of the OracleConnection interface. Before you enqueue the message, you can specify some enqueue options. The AQEnqueueOptions class enables you to specify the following enqueue options:

- Delivery mode: Specifies the delivery mode. Delivery mode can be set to either persistent (ENQUEUE_PERSISTENT) or buffered (ENQUEUE_BUFFERED).
- Retrieve Message ID: Specifies whether or not the message ID has to be retrieved from the server when the message has been enqueued. By default, the message ID is not retrieved.



 Transformation: Specifies a transformation that will be applied before enqueuing the message. The return type of the transformation function must match the type of the queue.



 Visibility: Specifies the transactional behavior of the enqueue request. The default value for this option is ENQUEUE_ON_COMMIT. It indicates that the enqueue operation is part of the current transaction. ENQUEUE_IMMEDIATE indicates that the enqueue operation is an autonomous transaction, which commits at the end of the operation. For buffered messaging, you must use ENQUEUE IMMEDIATE.

The following code snippet illustrates how to set the enqueue options and enqueue the message:

```
AQEnqueueOptions opt = new AQEnqueueOptions();opt.setRetrieveMessageId(true);
conn.enqueue(queueName, opt, mesg);
```

29.7 Dequeuing Messages

. . .

Enqueued messages can be dequeued using the dequeue method of the OracleConnection interface. Before you dequeue a message you must set the dequeue options. The AQDequeueOptions class enables you to specify the following dequeue options:

- Condition: Specifies a conditional expression based on the message properties, the message data properties, and PL/SQL functions. A dequeue condition is specified as a Boolean expression using syntax similar to the WHERE clause of a SQL query.
- Consumer name: If specified, only the messages matching the consumer name are accessed.

Note:

If the queue is a single-consumer queue, do not set this option.

- Correlation: Specifies a correlation criterion (or search criterion) for the dequeue operation.
- Delivery Filter: Specifies the type of message to be dequeued. You dequeue buffered messages only (DEQUEUE_BUFFERED) or persistent messages only (DEQUEUE_PERSISTENT), which is the default, or both (DEQUEUE_PERSISTENT_OR_BUFFERED).
- Dequeue Message ID: Specifies the message identifier of the message to be dequeued. This can be used to dequeue a unique message whose ID is known.



- Dequeue mode: Specifies the locking behavior associated with the dequeue operation. It can take one of the following values:
 - DequeueMode.BROWSE: Message is dequeued without acquiring any lock.
 - DequeueMode.LOCKED: Message is dequeued with a write lock that lasts for the duration of the transaction.
 - DequeueMode.REMOVE: (default) Message is dequeued and deleted. The message can be retained in the queue based on the retention properties.
 - DequeueMode.REMOVE NO DATA: Message is marked as updated or deleted.
- Maximum Buffer Length: Specifies the maximum number of bytes that will be allocated when dequeuing a message from a RAW queue. The default maximum is DEFAULT_MAX_PAYLOAD_LENGTH but it can be changed to any other nonzero value. If the buffer is not large enough to contain the entire message, then the exceeding bytes will be silently ignored.
- Navigation: Specifies the position of the message that will be retrieved. It can take one of the following values:
 - NavigationOption.FIRST_MESSAGE: The first available message matching the search criteria is dequeued.
 - NavigationOption.NEXT_MESSAGE: (default) The next available message matching the search criteria is dequeued. If the previous message belongs to a message group, then the next available message matching the search criteria in the message group is dequeued.
 - NavigationOption.NEXT_TRANSACTION: Messages in the current transaction group are skipped, and the first message of the next transaction group is dequeued. This setting can be used *only* if message grouping is enabled for the queue.
- Retrieve Message ID: Specifies whether or not the message identifier of the dequeued message needs to be retrieved. By default, it is not retrieved.
- Transformation: Specifies a transformation that will be applied after dequeuing the message. The source type of the transformation must match the type of the queue.

Note:

Transformations must be created in PL/SQL using DBMS TRANSFORM.CREATE TRANSFORMATION(...).

- Visibility: Specifies whether or not the message is dequeued as part of the current transaction. It can take one of the following values:
 - VisibilityOption.ON_COMMIT: (default) The dequeue operation is part of the current transaction.
 - VisibilityOption.IMMEDIATE: The dequeue operation is an autonomous transaction that commits at the end of the operation.



Note:

The Visibility option is ignored in the DequeueMode.BROWSE dequeue mode. If the delivery filter is DEQUEUE_BUFFERED or DEQUEUE_PERSISTENT_OR_BUFFERED, then this option must be set to VisibilityOption.IMMEDIATE.

Wait: Specifies the wait time for the dequeue operation, if none of the messages matches the search criteria. The default value is DEQUEUE_WAIT_FOREVER indicating that the operation waits forever. If set to DEQUEUE_NO_WAIT, then the operation does not wait. If a number is specified, then the dequeue operation waits for the specified number of seconds.

Note:

If you use DEQUEUE_WAIT_FOREVER, then the dequeue operation will not return until a message that matches the search criterion is available in the queue. However, you can interrupt the dequeue operation by calling the cancel method on the OracleConnection object.

The following code snippet illustrates how to set the dequeue options and dequeue the message:

```
...
AQDequeueOptions deqopt = new AQDequeueOptions();
deqopt.setRetrieveMessageId(true);
deqopt.setConsumerName(consumerName);
AQMessage msg = conn.dequeue(queueName,deqopt,queueType);
```

29.8 Examples: Enqueuing and Dequeuing

This section provides a few examples that illustrate how to enqueue and dequeue messages.

Example 29-2 illustrates how to enqueue a message, and Example 29-3 illustrates how to dequeue a message.

Example 29-2 Enqueuing a Single Message

This example illustrates how to obtain access to a queue, create a message, and enqueue it.

```
AQMessageProperties msgprop = AQFactory.createAQMessageProperties();
msgprop.setPriority(1);
msgprop.setExceptionQueue("EXCEPTION_QUEUE");
msgprop.setExpiration(0);
AQAgent agent = AQFactory.createAQAgent();
agent.setName("AGENTNAME");
agent.setAddress("AGENTADDRESS");
msgprop.setSender(agent);
AQMessage mesg = AQFactory.createAQMessage(msgprop);
mesg.setPayload(buffer); // where buffer is a byte array (for a RAW queue)
AQEnqueueOptions options = new AQEnqueueOptions();
conn.enqueue("HR.MY_QUEUE", options, mesg);
```



Example 29-3 Dequeuing a Single Message

This example illustrates how to obtain access to a queue, set the dequeue options, and dequeue the message.

AQDequeueOptions options = new AQDequeueOptions(); options.setDeliveryFilter(AQDequeueOptions.DeliveryFilter.BUFFERED); AQMessage mesg = conn.dequeue("HR.MY_QUEUE", options, "RAW");



30 Continuous Query Notification

This chapter describes how the Continuous Query Notification feature works.

This chapter describes the following topics:

- Overview of Continuous Query Notification
- Overview of Client Initiated Continuous Query Notification
- Creating a Registration
- Associating a Query with a Registration
- Notifying Database Change Events
- Deleting a Registration

30.1 Overview of Continuous Query Notification

Generally, a middle-tier data cache duplicates some data from the back-end database server, so that it can avoid redundant queries to the database. However, this is efficient only when the data rarely changes in the database. The data cache has to be updated or invalidated when the data changes in the database.

Starting from 11g Release 1, Oracle JDBC drivers provide support for the Continuous Query Notification feature of Oracle Database. Using this functionality, multitier systems can take advantage of the Continuous Query Notification feature to maintain a data cache as up-to-date as possible, by receiving invalidation events from the JDBC drivers.

The JDBC drivers can register SQL queries with the database and receive notifications in response to the following:

- DML or DDL changes on the objects associated with the queries
- DML or DDL changes that affect the result set

The notifications are published when the DML or DDL transaction commits (changes made in a local transaction do not generate any event until they are committed).

To use Oracle JDBC driver support for Continuous Query Notification, perform the following:

- 1. Registration: You first need to create a registration.
- 2. Query association: After you have created a registration, you can associate SQL queries with it. These queries are part of the registration.
- Notification: Notifications are created in response to changes in tables or result set. Oracle database communicates these notifications to the JDBC drivers through a dedicated network connection and JDBC drivers convert these notifications to Java events.

Also, you need to grant the CHANGE NOTIFICATION privilege to the user. For example, if you connect to the database using the HR user name, then you need to run the following command in the database:



grant change notification to HR;

30.2 Overview of Client Initiated Continuous Query Notification

Starting from Oracle Database Release 19c, the JDBC Thin driver supports the Client Initiated Continuous Query Notification feature. In this case, the client application initiates a connection to the database server for receiving notifications.

A client application first initiates a new database connection before creating a Continuous Query Notification registration. When the application creates a registration, the JDBC driver internally starts a new thread and creates a new connection with the database server. The database server then uses this new connection to send change notifications to the client.

By default, this feature is disabled for an on-premise database. You must set the OracleConnection.DCN CLIENT INIT CONNECTION to true for enabling this feature.

See Also:

Continuous Query Notification Registration Options

30.3 Creating a Registration

Creating a CQN registration is a one-time process and is done outside the currently used transaction. The API for creating a registration in the server is executed in its own transaction and is committed immediately.

You need a JDBC connection to create a registration. However, the registration is not attached to the connection. You can close the connection after creating a registration, and the registration survives. In an Oracle RAC environment, a registration is a persistent entity that exists on all nodes. The registration exists in the Database. So, even if a node goes down, the registration continues to exist and is notified when the tables change.

There are two ways to create a registration:

- The JDBC-style of registration: Use the JDBC driver to create a registration on the server. The JDBC driver launches a new thread that listens to notifications from the server (through a dedicated channel) and converts these notification messages into Java events. The driver then notifies all the listeners registered with this registration.
- The PL/SQL-style of registration: If you want a PL/SQL stored procedure to handle the notifications, then create a PL/SQL-style registration. As in the JDBC-style of registration, the JDBC drivers enable you to attach statements (queries) to this registration. However the JDBC drivers do not get notifications from the server because the notifications are handled by the PL/SQL stored procedure.



Note:

This approach is useful only for nonmultithreaded languages, such as PHP.

There is no way to remove one particular object (table) from an existing registration. A workaround would be to either create a new registration without this object or ignore the events that are related to this object.

You can use the registerDatabaseChangeNotification method of the oracle.jdbc.OracleConnection interface to create a JDBC-style of registration. You can set certain registration options through the options parameter of this method. The "Continuous Query Notification Registration Options" table in the following section lists some of the registration options that can be set. To set these options, use the java.util.Properties object. These options are defined in the oracle.jdbc.OracleConnection interface. The registration options have a direct impact on the notification events that the JDBC drivers will create. The example (at the end of this chapter) illustrates how to use the Continuous Query Notification feature.

The registerDatabaseChangeNotification method creates a new database change registration in the database server with the given options. It returns a DatabaseChangeRegistration object, which can then be used to associate a statement with this registration. It also opens a listener socket that will be used by the database to send notifications.

Note:

If a listener socket (created by a different registration) exists, then this socket is used by the new database change registration as well.

30.3.1 Continuous Query Notification Registration Options

The following table lists the Continuous Query Notification Registration Options:

Table 30-1 Continuous Query Notification Registration Options

Option	Description	
DCN_IGNORE_DELETEOP	If set to true, DELETE operations will not generate any database change event.	
DCN_IGNORE_INSERTOP	If set to true, INSERT operations will not generate any database change event.	
DCN_IGNORE_UPDATEOP	If set to true, UPDATE operations will not generate any database change event.	
DCN_NOTIFY_CHANGELAG	Specifies the number of transactions by which the client is willing to lag behind.	
	Note: If this option is set to any value other than 0, then ROWID level granularity of information will not be available in the events, even if the DCN_NOTIFY_ROWIDS option is set to true.	
DCN_NOTIFY_ROWIDS	Database change events will include row-level details, such as operation type and ROWID.	



Option	Description	
DCN_QUERY_CHANGE_NOTIFICATION	Activates query change notification instead of object change notification.	
	Note: This option is available only when running against an 11.0 database.	
DCN_CLIENT_INIT_CONNECTION	Specifies Client Initiated Continuous Query Notification, where the client application initiates a database connection, which the server uses to send change notifications to the client.	
NTF_LOCAL_HOST	Specifies the IP address of the computer that will receive the notifications from the server.	
NTF_LOCAL_TCP_PORT	Specifies the TCP port that the driver should use for the listener socket.	
NTF_QOS_PURGE_ON_NTFN	Specifies if the registration should be expunged on the first notification event.	
NTF_QOS_RELIABLE	Specifies whether or not to make the notifications persistent, which comes at a performance cost.	
NTF_TIMEOUT	Specifies the time in seconds after which the registration will be automatically expunged by the database.	

Table 30-1 (Cont.) Continuous Query Notification Registration Options

If there exists a registration, then you can also use the

getDatabaseChangeRegistration method to map the existing registration with a new DatabaseChangeRegistration object. This method is particularly useful if you have created a registration using PL/SQL and want to associate a statement with it.

30.4 Associating a Query with a Registration

After you have created a registration or mapped to an existing registration, you can associate a query with it. Like creating a registration, associating a query with a registration is a one-time process and is done outside of the currently used registration. The query will be associated even if the local transaction is rolled back.

You can associate a query with registration using the setDatabaseChangeRegistration method defined in the OracleStatement class. This method takes a DatabaseChangeRegistration object as parameter. The following code snippet illustrates how to associate a query with a registration:

```
// conn is an OracleConnection object.
// prop is a Properties object containing the registration options.
DatabaseChangeRegistration dcr = conn.registerDatabaseChangeNotifictaion(prop);
...
Statement stmt = conn.createStatement();
// associating the query with the registration
((OracleStatement)stmt).setDatabaseChangeRegistration(dcr);
// any query that will be executed with the 'stmt' object will be associated with
// the registration 'dcr' until 'stmt' is closed or
// '((OracleStatement)stmt).setDatabaseChangeRegistration(null);' is executed.
...
```



30.5 Notifying Database Change Events

To receive Continuous Query Notifications, attach a listener to the registration. When a database change event occurs, the database server notifies the JDBC driver. The driver then constructs a new Java event, identifies the registration to be notified, and notifies the listeners attached to the registration. The event contains the object ID of the database object that has changed and the type of operation that caused the change. Depending on the registration options, the event may also contain row-level detail information. The listener code can then use the event to make decisions about the data cache.

Note:

The listener code must not slow down the JDBC notification mechanism. If the code is time-consuming, for example, if it refreshes the data cache by querying the database, then it needs to be executed within its own thread.

You can attach a listener to a registration using the addListener method. The following code snippet illustrates how to attach a listener to a registration:

```
// conn is an OracleConnection object.
// prop is a Properties object containing the registration options.
DatabaseChangeRegistration dcr = conn.registerDatabaseChangeNotifictaion(prop);
...
// Attach the listener to the registration.
// Note: DCNListener is a custom listener and not a predefined or standard
// lsiener
DCNListener list = new DCNListener();
dcr.addListener(list);
...
```

30.6 Deleting a Registration

You need to explicitly unregister a registration to delete it from the server and release the resources in the driver. You can unregister a registration using a connection different from one that was used for creating it. To unregister a registration, you can use the unregisterDatabaseChangeNotification method defined in oracle.jdbc.OracleConnection.

You must pass the DatabaseChangeRegistration object as a parameter to this method. This method deletes the registration from the server and the driver and closes the listener socket.

If the registration was created outside of JDBC, say using PL/SQL, then you must pass the registration ID instead of the DatabaseChangeRegistration object. The method will delete the registration from the server, however, it does not free any resources in the driver.

Example 30-1 illustrates how to use the Continuous Query Notification feature. In this example, the HR user is connecting to the database. Therefore in the database you need to grant the following privilege to the user:

```
grant change notification to HR;
```



This code will also work with Oracle Database 10g Release 2 (10.2). This code uses table registration. That is, when you register a SELECT query, what you register is the name of the tables involved and not the query itself. In other words, you might select one single row of a table and if another row is updated, you will be notified although the result of your query has not changed.

In this example, if you leave the registration open instead of closing it, then the Continuous Query Notification thread continues to run. Now if you run a DML query that changes the HR.DEPARTMENTS table and commit it, say from SQL*Plus, then the Java program prints the notification.

Example 30-1 Continuous Query Notification

```
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;
import oracle.jdbc.OracleConnection;
import oracle.jdbc.OracleDriver;
import oracle.jdbc.OracleStatement;
import oracle.jdbc.dcn.DatabaseChangeEvent;
import oracle.jdbc.dcn.DatabaseChangeListener;
import oracle.jdbc.dcn.DatabaseChangeRegistration;
public class DBChangeNotification
{
  static final String USERNAME= "HR";
  static final String PASSWORD= "hr";
  static String URL;
  public static void main(String[] argv)
    if(argv.length < 1)
    {
      System.out.println("Error: You need to provide the URL in the first
argument.");
      System.out.println(" For example: > java -classpath .:ojdbc6.jar
DBChangeNotification \"jdbc:oracle:thin:
@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=yourhost.yourdomain.com)(PORT=5221))
(CONNECT DATA=
(SERVICE NAME=orcl)))\"");
      System.exit(1);
    }
    URL = argv[0];
    DBChangeNotification demo = new DBChangeNotification();
    try
    {
      demo.run();
    }
    catch(SQLException mainSQLException )
    {
      mainSQLException.printStackTrace();
    }
  }
  void run() throws SQLException
  {
    OracleConnection conn = connect();
    // first step: create a registration on the server:
```



```
Properties prop = new Properties();
    // if connected through the VPN, you need to provide the TCP address of the client.
    // For example:
    // prop.setProperty(OracleConnection.NTF LOCAL HOST, "14.14.13.12");
    // Ask the server to send the ROWIDs as part of the DCN events (small performance
    // cost):
   prop.setProperty(OracleConnection.DCN NOTIFY ROWIDS, "true");
11
//Set the DCN QUERY CHANGE NOTIFICATION option for query registration with finer
granularity.
prop.setProperty(OracleConnection.DCN QUERY CHANGE NOTIFICATION, "true");
    // The following operation does a roundtrip to the database to create a new
    // registration for DCN. It sends the client address (ip address and port) that
    // the server will use to connect to the client and send the notification
    // when necessary. Note that for now the registration is empty (we haven't
registered
    // any table). This also opens a new thread in the drivers. This thread will be
    // dedicated to DCN (accept connection to the server and dispatch the events to
    // the listeners).
    DatabaseChangeRegistration dcr = conn.registerDatabaseChangeNotification(prop);
    try
    {
      // add the listenerr:
      DCNDemoListener list = new DCNDemoListener(this);
      dcr.addListener(list);
      // second step: add objects in the registration:
      Statement stmt = conn.createStatement();
      // associate the statement with the registration:
      ((OracleStatement)stmt).setDatabaseChangeRegistration(dcr);
      ResultSet rs = stmt.executeQuery("select * from dept where deptno='45'");
      while (rs.next())
      { }
      String[] tableNames = dcr.getTables();
      for(int i=0;i<tableNames.length;i++)</pre>
        System.out.println(tableNames[i]+" is part of the registration.");
      rs.close();
      stmt.close();
    }
    catch(SQLException ex)
      // if an exception occurs, we need to close the registration in order
      // to interrupt the thread otherwise it will be hanging around.
     if(conn != null)
        conn.unregisterDatabaseChangeNotification(dcr);
      throw ex;
    1
    finally
    {
      try
        // Note that we close the connection!
       conn.close();
      catch(Exception innerex) { innerex.printStackTrace(); }
    1
```



```
synchronized( this )
      // The following code modifies the dept table and commits:
     try
      {
        OracleConnection conn2 = connect();
       conn2.setAutoCommit(false);
        Statement stmt2 = conn2.createStatement();
        stmt2.executeUpdate("insert into dept (deptno,dname) values ('45','cool
dept')",
Statement.RETURN GENERATED KEYS);
        ResultSet autoGeneratedKey = stmt2.getGeneratedKeys();
        if(autoGeneratedKey.next())
         System.out.println("inserted one row with
ROWID="+autoGeneratedKey.getString(1));
       stmt2.executeUpdate("insert into dept (deptno,dname) values ('50','fun
dept')",
Statement.RETURN GENERATED KEYS);
       autoGeneratedKey = stmt2.getGeneratedKeys();
       if(autoGeneratedKey.next())
         System.out.println("inserted one row with
ROWID="+autoGeneratedKey.getString(1));
       stmt2.close();
       conn2.commit();
       conn2.close();
     }
     catch(SQLException ex) { ex.printStackTrace(); }
     // wait until we get the event
     try{ this.wait();} catch( InterruptedException ie ) {}
    1
    // At the end: close the registration (comment out these 3 lines in order
    // to leave the registration open).
   OracleConnection conn3 = connect();
   conn3.unregisterDatabaseChangeNotification(dcr);
    conn3.close();
  }
  /**
   * Creates a connection the database.
  */
  OracleConnection connect() throws SQLException
  {
   OracleDriver dr = new OracleDriver();
   Properties prop = new Properties();
   prop.setProperty("user",DBChangeNotification.USERNAME);
   prop.setProperty("password", DBChangeNotification.PASSWORD);
   return (OracleConnection)dr.connect(DBChangeNotification.URL,prop);
 }
}
/**
 * DCN listener: it prints out the event details in stdout.
 */
class DCNDemoListener implements DatabaseChangeListener
 DBChangeNotification demo;
  DCNDemoListener (DBChangeNotification dem)
  {
   demo = dem;
  }
```



```
public void onDatabaseChangeNotification(DatabaseChangeEvent e)
{
   Thread t = Thread.currentThread();
   System.out.println("DCNDemoListener: got an event ("+this+" running on thread
"+t+")");
   System.out.println(e.toString());
   synchronized( demo ){ demo.notify();}
  }
}
```

Part VI High Availability

This section provides information about the high availability features of Oracle Database 12c Release 2 (12.2.0.1).

Part VI contains the following chapters:

- Transaction Guard for Java
- Application Continuity for Java
- Oracle JDBC Support for FAN Events
- Transparent Application Failover
- Single Client Access Name



31 Transaction Guard for Java

The Transaction Guard feature of Oracle Database provides a generic infrastructure for atmost-once execution during planned and unplanned outages and duplicate submissions. This chapter discusses Transaction Guard for Java in the following sections:

- Overview of Transaction Guard for Java
- Transaction Guard Support for XA Transactions
- Transaction Guard for Java APIs
- Complete Example:Using Transaction Guard APIs
- About Using Server-Side Transaction Guard APIs

31.1 Overview of Transaction Guard for Java

For most of the current applications, determining the outcome of the last commit operation in a guaranteed and scalable manner, following a communication failure to the server, is an unsolved problem. In many cases, the applications prompt the end users to follow certain steps to avoid resubmitting duplicate requests. For example, some applications warn users not to click the Submit button twice because if it is not followed, then users may unintentionally purchase the same items twice and submit multiple payments for the same invoice.

To solve this problem, Transaction Guard for Java provides transaction idempotence, that is, every transaction has at-most-once execution that prevents applications from submitting duplicate transactions. You can tag every transaction with a Logical Transaction Identifier (LTXID), which the application can use after the occurrence of a failure to verify whether the transaction had committed before the failure or not. For example, if the commit calls do not return, then using the LTXID, the application can find out whether the calls succeeded or not.

The Application Continuity for Java feature uses Transaction Guard for Java internally, which enables transparent session recovery and replay of SQL statements (queries and DMLs) since the beginning of the in-flight transaction. Application Continuity enables recovery of work after the occurrence of a planned or unplanned outage and Transaction Guard for Java ensures transaction idempotence. When an outage occurs, the recovery restores the state exactly as it was before the failure occurred.

Related Topics

Application Continuity for Java

The outages of the underlying software, hardware, communications, and storage layers can cause application execution to fail. In the worst cases, the middle-tier servers may need to be restarted to deal with a logon storm, which is a sudden increase in the number of client connection requests.

31.2 Transaction Guard Support for XA Transactions



Starting from Oracle Database 12c Release 2 (12.2.0.1), Transaction Guard provides support for XA transactions for one-phase commit optimization, read-only optimization, and promotable XA. Transaction Guard with XA provides safe replay following recoverable outages for XA transactions. With the addition of XA support, Transaction Managers can now provide replay with idempotence enforced more easily using Transaction Guard.

Note:

For using Transaction Guard with XA, during session check-out from the connection pool, you must verify that the database version is Oracle 12c Release 2 (12.2.0.1) and Transaction Guard is enabled.

A new server protocol provides a guaranteed commit outcome when the commit is one-phase managed by the database, and switches to a disabled mode while the Transaction Manager coordinates a transaction for that session. The new protocol sets a status flag in the LTXID that validates and invalidates access to the LTXID, based on the current transaction owner.

The protocol is intelligent in its handling that XA can encompass many sessions and many branches for the one XA transaction. As a further challenge, once a branch is suspended, a session is available for different transactions, while the original transaction remains active. There is no requirement to prepare or commit XA transactions on the same session or RAC instance that created the original branches. Transaction Guard for XA uses the following two new methods for handling commit outcome at the database for one-phase XA transactions, while the Transaction Manager continues to handle commit-outcome for two-phase transactions:

- Using the first method, the driver marks the LTXID provisional until a recoverable error, or any other named condition, occurs on that session. When a recoverable error (or any other condition) occurs, the LTXID at the client is marked final. The guaranteed commit outcome is provided only when the LTXID is final at the client, and at the server that LTXID has a VALID status, indicating that the database owns that transaction. Any other access attempt returns an error.
- Using the second method, the client driver does not provide the LTXID to the application until a recoverable error, or other named condition, occurs on that session.

31.3 How to Use Transaction Guard with XA

This section contains the following sections:

Obtaining the Commit Outcome with Promotable XA

For local transactions, the request obtains an LTXID as the transaction key, when there is a recoverable exception. When a second branch starts, then the request is promoted to XA, or converted to XA, and a Global Transaction ID (GTRID) is allocated to it. If a recoverable outage occurs during commit processing, where the application does not receive a reply from the Transaction Manager, then the application can ask a Transaction Manager for the outcome. Most requests to the database use either local transactions or single branch optimization. When you use either local transactions or



promotable XA, then there is no overhead in round trips and management for XA, because the majority of the transactions are local. The workflow of these transactions follows:

- **1.** Prior to converting to XA, transaction processing is local. Authentication, SELECT statements, and local transactions carry and use the LTXID of the local transaction.
- 2. The Transaction Manager allocates a GTRID to the transaction only when it starts to use XA due to opening a second branch of the transaction.
- 3. Following a recoverable error, when the application does not receive a commit outcome, if the transaction is local, then the Transaction Manager can use the LTXID with the GET_LTXID_OUTCOME procedure to retrieve the commit outcome and return COMMITTED or UNCOMMITTED outcome to the application.

Replaying if Promotable XA Is Added

Before being promoted, promotable XA supports RDBMS commits through calls and settings that are not supported by static XA. These calls include auto-commit mode, DDL, DCL, COMMIT embedded in PL/SQL, and COMMIT through remote procedure calls. The COMMIT outcome for these user calls and modes is controlled by the RDBMS, and following an error, the commit outcome can be found using Transaction Guard.

Until promoted, the Transaction Manager is unaware whether the request has issued any COMMIT or not. If the Transaction Manager wishes to replay a request following a recoverable error, then the Transaction Manager must determine if any RDBMS COMMIT has occurred. If any RDBMS COMMIT occurs, or can occur, then replay does not happen. The GET_LTXID_OUTCOME procedure is insufficient in determining this because the procedure only reports the current transaction outcome. If the LTXID is changed, then the transaction is committed. So, the invocation of the LTXID callback indicates that the transaction is committed.

31.4 Transaction Guard for Java APIs

This section discusses the APIs associated with Transaction Guard for Java for the following activities:

- Retrieving the Logical Transaction Identifiers
- Retrieving the Updated Logical Transaction Identifiers

31.4.1 Retrieving the Logical Transaction Identifiers

Use the getLogicalTransactionId method of the oracle.jdbc.OracleConnection interface to retrieve the current Logical Transaction Identifiers that are sent by the server. This method call does not make a database round-trip.

Example

```
OracleConnection oconn = (OracleConnection) ods.getConnection();
...
// Getting the 1st LTXID after connecting
LogicalTransactionId firstLtxid = oconn.getLogicalTransactionId();
```



31.4.2 Retrieving the Updated Logical Transaction Identifiers

Use the oracle.jdbc.LogicalTransactionIdEventListener interface for receiving updates to Logical Transaction Identifiers. You must implement this interface in your application to process the Logical Transaction Identifier events.

31.4.2.1 Registering Event Listeners

Use the addLogicalTransactionIdEventListener method to register a listener to the Logical Transaction Identifier events.

Example

OracleConnection oconn = (OracleConnection) ods.getConnection();
...
// The subsequent LTXID updates can be obtained through the listener
oconn.addLogicalTransactionIdEventListener(this);

You can also use the

```
addLogicalTransactionIdEventListener(LogicalTransactionIdEventListener
listener, java.util.concurrent.Executor executor) method to register a listener
with an executor.
```

31.4.2.2 Unregistering Event Listeners

Use the removeLogicalTransactionIdEventListener method to unregister a listener from the Logical Transaction Identifier events.

Example

```
OracleConnection oconn = (OracleConnection) ods.getConnection();
...
// The subsequent LTXID updates can be obtained through the listener
oconn.removeLogicalTransactionIdEventListener(this);
```

31.5 Complete Example: Using Transaction Guard APIs

The following is a complete example using the Transaction Guard APIs.

```
import oracle.jdbc.pool.OracleDataSource;
import oracle.jdbc.OracleConnection;
import oracle.jdbc.LogicalTransactionId;
import oracle.jdbc.LogicalTransactionIdEvent;
import oracle.jdbc.LogicalTransactionIdEventListener;
public class transactionGuardExample
{
    ...
    OracleDataSource ods = new OracleDataSource();
    ods.setURL(url);
    ods.setUser("user");
    ods.setPassword("password");
```



```
OracleConnection oconn = (OracleConnection) ods.getConnection();
    // Getting the 1st LTXID after connecting
    LogicalTransactionId firstLtxid = oconn.getLogicalTransactionId();
    // The subsequent LTXID updates can be obtained via the listener
    oconn.addLogicalTransactionIdEventListener(this);
  }
public class LtxidListenerImpl
  implements LogicalTransactionIdEventListener
{
    ...
    public void onLogicalTransactionIdEvent(LogicalTransactionIdEvent ltxidEvent)
    {
        LogicalTransactionId newLtxid = ltxidEvent.getLogicalTransactionId();
        // process newLtxid ......
    }
}
```

31.6 About Using Server-Side Transaction Guard APIs

The DBMS_APP_CONT package contains the GET_LTXID_OUTCOME procedure that contains the server-side Transaction Guard APIs. This procedure forces the outcome of a transaction. If the transaction is not committed, then a fake transaction is committed. Otherwise, the state of the transaction is returned. By default, the EXECUTE privilege for this package is granted to Database Administrators.

Syntax

```
PROCEDURE GET_LTXID_OUTCOME(CLIENT_LTXID IN RAW,
committed OUT BOOLEAN,
USER_CALL_COMPLETED OUT BOOLEAN);
```

Input Parameter

CLIENT LTXID specifies the LTXID from the client driver.

Output Parameter

COMMITTED specifies that the transaction is committed.

USER_CALL_COMPLETED specifies that the user call, which committed the transaction, is complete.

Exceptions

SERVER_AHEAD is thrown when the server is ahead of the client. So, the transaction is an old transaction and must have already been committed.

CLIENT_AHEAD is thrown when the client is ahead of the server. This can only happen if the server is flashed back or the LTXID is corrupted. In either of these situations, the outcome cannot be determined.

ERROR is thrown when an error occurs during processing and the outcome cannot be determined. It specifies the error code raised during the execution of the current procedure.



Example

Example 31-1 shows how you can call the GET_LTXID_OUTCOME procedure and find out the outcome of an LTXID:

Example 31-1 Finding Out the Outcome of an LTXID

```
. . .
OracleConnection oconn = (OracleConnection) ods.getConnection();
LogicalTransactionId ltxid = oconn.getLogicalTransactionId();
boolean committed = false;
boolean call completed = false;
try
{
 CallableStatement cstmt = oconn.prepareCall(GET_LTXID_OUTCOME);
 cstmt.setObject(1, ltxid);
 cstmt.registerOutParameter(2, OracleTypes.BIT);
 cstmt.registerOutParameter(3, OracleTypes.BIT);
 cstmt.execute();
  committed = cstmt.getBoolean(2);
  call completed = cstmt.getBoolean(3);
 System.out.println("LTXID committed ? " + committed);
 System.out.println("User call completed ? " + call completed);
}
catch (SQLException sqlexc)
{
 System.out.println("Calling GET LTXID OUTCOME failed");
  sqlexc.printStackTrace();
}
```



32 Application Continuity for Java

The outages of the underlying software, hardware, communications, and storage layers can cause application execution to fail. In the worst cases, the middle-tier servers may need to be restarted to deal with a logon storm, which is a sudden increase in the number of client connection requests.

To overcome such problems, Oracle Database 12c Release 1 (12.1) introduced the Application Continuity feature that masks database outages to the application and end users are not exposed to such outages.

Note:

- You must use Transaction Guard for using Application Continuity.
- Explicit switch of container and service in applications through the ALTER SESSION SET CONTAINER statement is not supported with Application Continuity.

Application Continuity provides a general purpose, application-independent solution that enables recovery of work from an application perspective, after the occurrence of a planned or unplanned outage. The outage can be related to system, communication, or hardware following a repair, a configuration change, or a patch application.

This chapter discusses the JDBC aspect of Application Continuity in the following sections:

- About Configuring Oracle JDBC for Application Continuity for Java
- About Configuring Oracle Database for Application Continuity for Java
- Application Continuity with DRCP
- Application Continuity Support for XA Data Source
- About Identifying Request Boundaries in Application Continuity for Java
- Support for Transparent Application Continuity
- Establishing the Initial State Before Application Continuity Replays
- About Delaying the Reconnection in Application Continuity for Java
- About Retaining Mutable Values in Application Continuity for Java
- Application Continuity Statistics
- About Disabling Replay in Application Continuity for Java

Related Topics

Transaction Guard for Java



32.1 About Configuring Oracle JDBC for Application Continuity for Java

You must use the

oracle.jdbc.replay.OracleDataSourceImpl,oracle.jdbc.replay.OracleConnectio nPoolDataSourceImpl, or oracle.jdbc.replay.driver.OracleXADataSourceImpl data source to obtain JDBC connections. You can use both

 $\verb|oracle.jdbc.replay.OracleDataSourceImpl| and \\$

oracle.jdbc.replay.OracleConnectionPoolDataSourceImpl in a standalone manner, or configure them as connection factories for a connection pool, such as Universal Connection Pool (UCP), or Oracle WebLogic Server connection pool. Starting from Oracle Database 12c Release 2 (12.2.0.1), the JDBC Replay Data Source provides a new data source, the XA Replay Data Source, which supports JDBC operations replay, and also works with both UCP data source and WebLogic Active GridLink single-pool data source for all the Oracle RAC features.

For using this data source, your application must implement the oracle.jdbc.replay.OracleXADataSource interface. The actual data source implementation class is oracle.jdbc.replay.driver.OracleXADataSourceImpl. You can specify the implementation class to UCP data sources and Oracle WebLogic Server GridLink data source as a connection factory. The factory class for JNDI is oracle.jdbc.replay.OracleXADataSourceFactory.

Note:

- The XA replay data source does not provide a configurable replay mode. For enabling replay, you must use the replay data source and set FAILOVER_TYPE to TRANSACTION on the database service at the server side, if not set already.
- Starting from Oracle Database Release 18c, you can also set FAILOVER_TYPE to AUTO for using Transparent Application Continuity.

See Also:

Oracle Real Application Clusters Administration and Deployment Guide

• For enabling and disabling replay dynamically, you must use a separate API available on the replay connection proxy. The XA replay data source does not provide connection pooling. Any getXAConnection method call produces a new JDBC XAConnection proxy dynamically, which holds a new JDBC physical connection as the delegate. The delegate is an Oracle JDBC driver object.

The following code snippet illustrates the usage of oracle.jdbc.replay.OracleDataSourceImpl and oracle.jdbc.replay.OracleConnectionPoolDataSourceImpl in a standalone JDBC application:



```
import java.sql.Connection;
import javax.sql.PooledConnection;
import oracle.jdbc.OracleConnection;
import oracle.jdbc.replay.OracleDataSourceFactory;
import oracle.jdbc.replay.OracleDataSource;
import oracle.jdbc.replay.OracleConnectionPoolDataSource;
. . .
{
. . . . . .
OracleDataSource rds = OracleDataSourceFactory.getOracleDataSource();
rds.setUser(user);
rds.setPassword(passwd);
rds.setURL(url);
..... // Other data source configuration like callback, timeouts, etc.
Connection conn = rds.getConnection();
((OracleConnection) conn).beginRequest(); // Explicit request begin
..... // JDBC calls protected by Application Continuity
((OracleConnection) conn).endRequest(); // Explicit request end
conn.close();
OracleConnectionPoolDataSource rcpds =
OracleDataSourceFactory.getOracleConnectionPoolDataSource();
rcpds.setUser(user);
rcpds.setPassword(passwd);
rcpds.setURL(url);
..... // other data source configuration like callback, timeouts, and so on
PooledConnection pc = rcpds.getPooledConnection();
Connection conn2 = pc.getConnection(); // Implicit request begin
..... // JDBC calls protected by Application Continuity
conn2.close(); // Implicit request end
. . . . . .
```

You must remember the following points while using the connection URL:

- Always use the thin driver in the connection URL.
- Always connect to a service. Never use instance_name or SID because these do not direct to known good instances and SID is deprecated.
- If the addresses in the ADDRESS_LIST at the client does not match the REMOTE_LISTENER setting for the database, then it does not connect showing services cannot be found. So, the addresses in the ADDRESS_LIST at the client *must* match the REMOTE_LISTENER setting for the database:
 - If REMOTE LISTENER is set to the SCAN_VIP, then the ADDRESS_LIST uses SCAN_VIP
 - If REMOTE_LISTENER is set to the host VIPs, then the ADDRESS_LIST uses the same host VIPs
 - If REMOTE_LISTENER is set to both SCAN_VIP and host VIPs, then the ADDRESS_LIST uses SCAN VIP and the same host VIPs



Note:

For Oracle clients prior to release 11.2, the <code>ADDRESS_LIST</code> must be upgraded to use SCAN, which means expanding the <code>ADDRESS_LIST</code> to three <code>ADDRESS</code> entries corresponding to the three SCAN IP addresses.

If such clients connect to a database that is upgraded from an earlier release through Database AutoUpgrade, then you must retain the ADDRESS_LIST of these clients set to the HOST VIPs. However, if REMOTE_LISTENER is changed to ONLY SCAN, or the clients are moved to a newly installed Oracle Database 12*c* Release 1, where REMOTE_LISTENER is ONLY SCAN, then they do not get a complete service map, and may not always be able to connect.

Set RETRY_COUNT, RETRY_DELAY, CONNECT_TIMEOUT, and TRANSPORT_CONNECT_TIMEOUT parameters in the connection string. This is a general recommendation for configuring the JDBC thin driver connections, starting from Oracle Database Release 12.1.0.2. These settings improve acquiring new connections at run time, at replay, and during work drains for planned outages.

The CONNECT_TIMEOUT parameter is equivalent to the SQLNET.OUTBOUND_CONNECT_TIMEOUT parameter in the sqlnet.ora file and applies to the full connection. The TRANSPORT_CONNECT_TIMEOUT parameter applies as per the ADDRESS parameter. If the service is not registered for a failover or restart, then retrying is important when you use SCAN. For example, for using remote listeners pointing to SCAN addresses, you should use the following settings:

Note:

In the following code snippets, the TRANSPORT_CONNECT_TIMEOUT parameter is set to 3. However, if you are using 12.2.0.1 JDBC driver in your application, without the required patch that fixes bug 25977056, then you must set the TRANSPORT_CONNECT_TIMEOUT parameter to 3000.

```
jdbc:oracle:thin:@(DESCRIPTION =
  (TRANSPORT_CONNECT_TIMEOUT=3)
  (RETRY_COUNT=20) (RETRY_DELAY=3) (FAILOVER=ON)
  (ADDRESS_LIST = (ADDRESS=(PROTOCOL=tcp)
    (HOST=CLOUD-SCANVIP.example.com) (PORT=5221))
  (CONNECT_DATA=(SERVICE_NAME=orcl)))
```

REMOTE LISTENERS=CLOUD-SCANVIP.example.com:5221

Similarly, for using remote listeners pointing to VIPs at the database, you should use the following settings:

```
jdbc:oracle:thin:@(DESCRIPTION =
  (TRANSPORT_CONNECT_TIMEOUT=3)
  (CONNECT_TIMEOUT=60) (RETRY_COUNT=20) (RETRY_DELAY=3) (FAILOVER=ON)
  (ADDRESS_LIST=
   (ADDRESS=(PROTOCOL=tcp) (HOST=CLOUD-VIP1.example.com) (PORT=5221) )
  (ADDRESS=(PROTOCOL=tcp) (HOST=CLOUD-VIP2.example.com) (PORT=5221) )
```



Related Topics

Data Sources and URLs

32.1.1 Support for Concrete Classes with Application Continuity

Starting from Oracle Database Release 18c, JDBC driver supports the following concrete classes with Application Continuity:

- oracle.sql.CLOB
- oracle.sql.NCLOB
- oracle.sql.BLOB
- oracle.sql.BFILE
- oracle.sql.STRUCT
- oracle.sql.REF
- oracle.sql.ARRAY

32.1.2 About Using LONG and LONG RAW columns with Application Continuity

Typically, Oracle recommends using LOB columns instead of LONG and LONG RAW columns. This also applies to Application Continuity.

If you have one or more LONG or LONG RAW columns in your table, then the JDBC driver transfers those to the client in streaming mode, when a query selects those columns. In streaming mode, the JDBC driver does not read the column data from the network for LONG or LONG RAW columns, until required. As a result of this, a query involving one or more such columns can result in a replay failure. So, if there is a situation when you cannot avoid using such columns, then perform the following tasks to resolve the issue:



About Streaming LONG or LONG RAW Columns



- Use the defineColumnType method to redefine the type of the LONG or LONG RAW column. For example, if you redefine the LONG or LONG RAW column as VARCHAR or VARBINARY type, then the driver will not stream the data automatically.
- Declare the types of all the columns involved in the query.

Note:

If you redefine the column types with the defineColumnType method, then you must declare the types of all the columns involved in the query. If you do not do so, then the executeQuery method will fail.

• Cast the Statement object to oracle.jdbc.OracleStatement.

32.2 About Configuring Oracle Database for Application Continuity for Java

You must have the following configuration for Oracle Database to use Application Continuity for Java:

- Use Oracle Database 12c Release 1 (12.1) or later
- If you are using Oracle Real Application Clusters (Oracle RAC) or Oracle Data Guard, then ensure that FAN is configured with Oracle Notification System (ONS) to communicate with Oracle WebLogic Server or the Universal Connection Pool (UCP)
- Use an application service for all database work. To create the service you must:
 - Run the SRVCTL command if you are using Oracle RAC
 - Use the DBMS SERVICE package if you are not using Oracle RAC
- Set the required properties on the service for replay and load balancing. For example, set:
 - aq ha notifications = TRUE for enabling FAN notification
 - FAILOVER_TYPE = TRANSACTION or FAILOVER_TYPE = AUTO for using Application Continuity
 - COMMIT OUTCOME = TRUE for enabling Transaction Guard
 - REPLAY_INITIATION_TIMEOUT = 900 for setting the duration in seconds for which replay will occur
 - FAILOVER_RETRIES = 30 for specifying the number of connection retries for each replay
 - FAILOVER_DELAY = 10 for specifying the delay in seconds between connection retries
 - GOAL = SERVICE_TIME, if you are using Oracle RAC, then this is a recommended setting



- CLB_GOAL = LONG, typically useful for closed workloads. If you are using Oracle RAC, then this is a recommended setting. For most of the other workloads, SHORT is the recommended setting.
- Do not use the database service, that is, the default service corresponding to the DB_NAME or DB_UNIQUE_NAME. This service is reserved for Oracle Enterprise Manager and for DBAs. Oracle does not recommend the use of the database service for high availability because this service cannot be:
 - Enabled and disabled
 - Relocated on Oracle RAC
 - Switched over to Oracle Data Guard

See Also:

Oracle Database Development Guide for more information on the operation and usage of Application Continuity.

32.3 Application Continuity with DRCP

Oracle Database Release 18c JDBC driver supports Application Continuity when Database Resident Connection Pooling (DRCP) is enabled on the server side. For using Application Continuity with DRCP, you must configure an application service to a server that uses DRCP. The following code snippet shows how to use Application Continuity with DRCP:

Example 32-1 Using Application Continuity with DRCP

```
String url = "jdbc:oracle:thin:@(DESCRIPTION =
                          (TRANSPORT CONNECT TIMEOUT=3000)
                          (RETRY COUNT=20) (RETRY DELAY=3) (FAILOVER=ON)
                          (ADDRESS LIST = (ADDRESS=(PROTOCOL=tcp)
                          (HOST=CLOUD-SCANVIP.example.com) (PORT=5221))
                          (CONNECT DATA=(SERVICE NAME=ac-service)
(SERVER=POOLED)))";
PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();
pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
// Set DataSource Property
pds.setUser("HR");
pds.setPassword("hr");
System.out.println ("Connecting to " + url);
pds.setURL(url);
pds.setConnectionPoolName("HR-Pool1");
pds.setMinPoolSize(2);
pds.setMaxPoolSize(3);
pds.setInitialPoolSize(2);
Properties prop = new Properties();
prop.put("oracle.jdbc.DRCPConnectionClass", "HR-Pool1");
```



```
pds.setConnectionProperties(prop);
...
```

Related Topics

Overview of Database Resident Connection Pooling

Related Topics

About Configuring Oracle Database for Application Continuity for Java

32.4 Application Continuity Support for XA Data Source

Oracle Database 12c Release 2 (12.2.0.1) introduced a new feature that enhances Application Continuity with support for Oracle XA data source (javax.sql.XADataSource), which is similar to non-XA data source (javax.sql.DataSource). Both JDBC and Java Transaction API (JTA) allow a JDBC connection to interchangeably participate in local and global/XA transactions. However, many customer applications obtain connections from an XA data source, but use these connections to perform only local transactions. With the new feature, Application Continuity also covers applications that are using XA-capable data sources but with local transactions, including local transactions that are promotable to global/XA transactions. So, the benefits of Application Continuity, such as, failover and forward-recovery are extended to these applications.

Note:

You must use Transaction Guard 12.2 for using this feature.

Whenever an underlying physical connection participates in a global/XA transaction, or engages in any XA operation, replay is disabled on that connection. All other XA operations function normally, but the application does not get Application Continuity protection.

Once replay is disabled on a connection for the above reasons, it remains disabled until the next request begins. Switching from global/XA transaction to local transaction mode does not automatically reenable replay on a connection.

The following code snippet illustrates how replay is supported for local transactions, and disabled for XA transaction, when the connections are obtained from OracleXADataSource:

```
import javax.transaction.xa.*;
import oracle.jdbc.replay.OracleXADataSource;
import oracle.jdbc.replay.OracleXADataSourceFactory;
import oracle.jdbc.replay.ReplayableConnection;
```

```
OracleXADataSource xards =
OracleXADataSourceFactory.getOracleXADataSource();
xards.setURL(connectURL);
xards.setUser(<user_name>);
xards.setPassword(<password>);
```



```
XAConnection xaconn = xards.getXAConnection();
// Implicit request begins
Connection conn = xaconn.getConnection();
/* Local transaction case */
// Request-boundary detection OFF
((ReplayableConnection) conn).beginRequest();
conn.setAutoCommit(false);
PreparedStatement pstmt=conn.prepareStatement("select
cust first name, cust last name from customers where customer id=1");
ResultSet rs=pstmt.executeQuery();
// Outage happens at this point
// Replay happens at this point
rs.next();
rs.close();
pstmt.close();
((ReplayableConnection) conn).endRequest();
. . .
/* Global/XA transaction case */
((ReplayableConnection) conn).beginRequest();
conn.setAutoCommit(false);
XAResource xares = xaconn.getXAResource();
Xid xid = createXid();
// Replay is disabled here
xares.start(xid, XAResource.TMNOFLAGS);
conn.prepareStatement ("INSERT INTO TEST TAB VALUES (200, 'another new
record')");
// outage happens at this point
try {
//No replay here and throws exception
conn.executeUpdate();
}
// sqlrexc.getNextException() shows the reason for the replay failure
catch (SQLRecoverableException sqlrexc) {
    .........
}
```

32.5 About Identifying Request Boundaries in Application Continuity for Java

A Request is a unit of work on a physical connection to Oracle Database that is protected by Application Continuity. Request demarcation varies with specific use-case scenarios. A request begins when a connection is borrowed from the Universal Connection Pool (UCP) or



WebLogic Server connection pool, and ends when this connection is returned to the connection pool.

The JDBC driver provides explicit request boundary declaration APIs <code>beginRequest</code> and <code>endRequest</code> in the <code>oracle.jdbc.OracleConnection</code> interface. These APIs enable applications, frameworks, and connection pools to indicate to the JDBC Replay Data Source about demarcation points, where it is safe to release the call history, and to enable replay if it had been disabled by a prior request. At the end of the request, the JDBC Replay Data Source purges the recorded history on the connection, where the API is called. This helps to further conserve memory consumption for applications that use the same connections for an extended period of time without returning them to the pool.

For the connection pool to work, the application must get connections when needed, and release connections when not in use. This scales better and provides request boundaries transparently. The APIs have no impact on the applications other than improving resource consumption, recovery, and load balancing performance. These APIs do not involve altering a connection state by calling any JDBC method, SQL, or PL/SQL. An error is returned if an attempt is made to begin or end a request while a local transaction is open.

32.6 Support for Transparent Application Continuity

Oracle Database Release 18c introduced the Transparent Application Continuity feature, which is a functional mode of Application Continuity. Transparent Application Continuity transparently tracks and records session and transactional state, so that a database session can be recovered following recoverable outages. This is performed safely and without the need for any knowledge of the application or application code changes. Transparency is achieved by using a state-tracking infrastructure that categorizes session state usage as an application issues user calls. This feature enables the driver to detect and inject possible request boundaries, which are known as implicit request boundaries. For an implicit request boundary:

- No objects are open
- Cursors are returned to the driver statement cache
- No transactions are open

The session state in such a case is known to be restorable. The driver either closes the current capture and starts a new event, or enables capture if there had been a disabling event. On the next call to the server, the server verifies and, if applicable, creates a request boundary, where there was previously no explicit boundary.

To use Transparent Application Continuity, you must set the server-side service attribute FAILOVER TYPE on the database service to AUTO.

Support for implicit request helps to reduce application failover recovery time and optimizes Application Continuity. Using Transparent Application Continuity, the server and the drivers can track transaction and session state usage. However, this feature should be used with caution for applications that change server session states during a request. The JDBC Thin driver provides the oracle.jdbc.enableImplicitRequests property to turn off implicit requests, if needed. This property can be set at the system level, which applies to all connections, or at the connection level, which applies to a particular connection. By default, the value of this property is true, which means that support for implicit request is enabled.



See Also: Oracle Real Application Clusters Administration and Deployment Guide

Starting from Oracle Database Release 19c, if you set the value of the FAILOVER_TYPE service attribute to AUTO, then the Oracle JDBC driver implicitly begins a request on each new physical connection created through the replay data source. With this feature, applications using third-party connection pools can use Transparent Application Continuity (TAC) easily, without making any code change to inject request boundaries.

This implicit beginRequest applies only to the replay data source, and only to the physical connections created during runtime, when TAC is enabled. The driver does not implicitly begin a request after each reconnection during replay attempts (that is, a beginRequest is not implicitly injected during a replay connection) or in manual Application Continuity mode, where the FAILOVER TYPE service attribute is set to TRANSACTION.

If you want to turn off this feature explicitly, you can set the value of the Java system property oracle.jdbc.beginRequestAtConnectionCreation to false. The default value of this property is true.

Related Topics

About Enabling FAILOVER_RESTORE

This section describes how to use the FAILOVER RESTORE service attribute.

32.6.1 Support for Session State Stable Cursors

Starting from Release 23c, Oracle JDBC drivers support session state stable (SSS) cursors, which are typically long-running cursors that stay open outside of transactional work. This support increases TAC protection coverage for applications with SSS cursors because it establishes implicit request boundaries more often, even when such cursors are open.

A session state stable (SSS) cursor can remain valid for the entire duration of an explicit request, and can be replayed separately from the main request. Such cursors are commonly used in batch processes. A cursor is considered as a session state stable cursor if it meets the following criteria:

- The cursor must be opened when connected to a TAC service
- The cursor must be open when:
 - The session has no open transaction in progress
 - The non-transactional session state is either client restorable or template restorable
- It must be possible to re-execute the query using the system change number (SCN), if replay is required
- The cursor must not create, modify, or read session state during a fetch call, if the fetch call occurs during a round trip that is separate from the cursor execution call

Basically, session state referenced by the cursor must be frozen at the conclusion of the execution phase. Some examples of cursors that are generally not classified as session state stable cursors are the cursors that reference the following:

sequence.nextval, sequence.currval



- sys context or application context
- PL/SQL functions

Note:

The preceding references are still disqualifying if they occur during fetch-time as a result of VPD-imposed clauses.

Enabling SSS Cursor Support

To enable SSS cursor support in Oracle JDBC drivers, use a TAC-enabled database service, and set -Doracle.jdbc.enableSSSCursor=true on the client side.

See Also:

Real Application Clusters Administration and Deployment Guide

32.7 Establishing the Initial State Before Application Continuity Replays

Non-transactional session state (NTSS) is state of a database session that exists outside database transactions and is not protected by recovery. For applications that use stateful requests, the non-transactional state is re-established as the rebuilt session.

For applications that set state only at the beginning of a request, or for stateful applications that gain performance benefits from using connections with a preset state, one among the following callback options are provided:

- No Callback
- Connection Labeling
- Connection Initialization Callback
- About Enabling FAILOVER_RESTORE

32.7.1 No Callback

In this scenario, the application builds up its own state during each request.

32.7.2 Connection Labeling

This scenario is applicable only to Universal Connection Pool (UCP) and Oracle WebLogic server. The application can be modified to take advantage of the preset state on connections. Connection Labeling APIs determine how well a connection matches, and use a callback to populate the gap when a connection is borrowed.



See Also: Oracle Universal Connection Pool Developer's Guide

32.7.3 Connection Initialization Callback

In this scenario, the replay data source uses an application callback to set the initial state of the session during runtime and replay. The JDBC Replay Data Source provides an optional connection initialization callback interface as well as methods for registering and unregistering such callbacks.

When registered, the initialization callback is executed at each successful reconnection following a recoverable error. An application is responsible for ensuring that the initialization actions are the same as that on the original connection before failover. If the callback invocation fails, then replay is disabled on that connection.

This section discusses initialization callbacks in the following sections:

- Creating an Initialization Callback
- Registering an Initialization Callback
- Removing or Unregistering an Initialization Callback

32.7.3.1 Creating an Initialization Callback

To create a JDBC connection initialization callback, an application implements the oracle.jdbc.replay.ConnectionInitializationCallback interface. One callback is allowed for every instance of the oracle.jdbc.replay.OracleDataSource interface.

Note:

This callback is only invoked during failover, after a successful reconnection.

Example

The following code snippet demonstrates a simple initialization callback implementation:

```
import oracle.jdbc.replay.ConnectionInitializationCallback;
class MyConnectionInitializationCallback implements ConnectionInitializationCallback {
    public MyConnectionInitializationCallback()
    {
        ...
    }
    public void initialize(java.sql.Connection connection) throws SQLException
    {
        // Reset the state for the connection, if necessary (like ALTER SESSION)
        ...
    }
  }
```



For applications using an XA data source, the connection initialization callback is registered on the XA replay data source. The callback is executed every time when *both* of the following happen:

- A connection is borrowed from the connection pool.
- The replay XA data source gets a new physical connection at failover.

Note:

The connection initialization must be idempotent. If the connection is already initialized, then it must not repeat itself. This enables applications to reestablish session initial starting point after a failover and before the starting of replay. The callback execution must leave an open local transaction without committing it or rolling it back. If this is violated, an exception is thrown.

If a callback invocation fails, replay is disabled on that connection. For example, an application embeds the set up phase for a connection in this callback.

32.7.3.2 Registering an Initialization Callback

Use the following method that the JDBC Replay Data Source provides in the oracle.jdbc.replay.OracleDataSource interface for registering a connection initialization callback:

registerConnectionInitializationCallback(ConnectionInitializationCallback cbk)

One callback is allowed for every instance of the OracleDataSource interface.

For using an XA Data Source, use the

registerConnectionInitializationCallback(ConnectionInitializationCallback cbk) method in the oracle.jdbc.replay.OracleXADataSource interface.

32.7.3.3 Removing or Unregistering an Initialization Callback

Use the following method that the JDBC Replay Data Source provides in the oracle.jdbc.replay.OracleDataSource interface for unregistering a connection initialization callback:

unregisterConnectionInitializationCallback(ConnectionInitializationCallback cbk)

For using an XA Data Source, use the

unregisterConnectionInitializationCallback(ConnectionInitializationCallbac
k cbk) method in the oracle.jdbc.replay.OracleXADataSource interface.

32.7.4 About Enabling FAILOVER_RESTORE

This section describes how to use the FAILOVER RESTORE service attribute.

FAILOVER_RESTORE service attribute was introduced in Oracle Database 12c Release 2 (12.2.0.1). Setting the FAILOVER_RESTORE attribute to LEVEL1 automatically restores the



common initial state before replaying a request. By default, the value of the FAILOVER RESTORE attribute is set to NONE, which means that it is disabled.

Starting from Oracle Database Release 18c, you can also set the value of this attribute to AUTO. Also, if you set the value of the FAILOVER_TYPE attribute to AUTO, then FAILOVER_RESTORE is set to AUTO automatically. You cannot change the value of FAILOVER_RESTORE to anything else as long as FAILOVER_TYPE is set to AUTO. When FAILOVER_RESTORE is set to AUTO, then the common initial state is also set. As far as session state restore is concerned, this setting provides the same function as FAILOVER_RESTORE set to LEVEL1.

See Also:

Real Application Clusters Administration and Deployment Guide



Note:

For Application Continuity for Java available with Oracle Database Release 23c, the following initial states are supported for FAILOVER_RESTORE:

- NLS CALENDAR
- NLS CURRENCY
- NLS_DATE_FORMAT
- NLS_DATE_LANGUAGE
- NLS DUAL CURRENCY
- NLS_ISO_CURRENCY
- NLS LANGUAGE
- NLS_LENGTH_SEMANTICS
- NLS_NCHAR_CONV_EXCP
- NLS_NUMERIC_CHARACTER
- NLS_SORT
- NLS TERRITORY
- NLS_TIME_FORMAT
- NLS_TIME_TZ_FORMAT
- NLS_TIMESTAMP_FORMAT
- NLS_TIMESTAMP_TZ_FORMAT
- TIME_ZONE (OCI, ODP.NET 12201)
- CURRENT_SCHEMA
- MODULE
- ACTION
- CLIENT_ID
- ECONTEXT ID
- ECONTEXT_SEQ
- DB_OP
- AUTOCOMMIT states (Java and SQL*Plus)
- CONTAINER (PDB) and SERVICE for OCI and ODP.NET

In Oracle Database Release 19c, the following additional initial states are supported for FAILOVER_RESTORE:

- ERROR_ON_OVERLAP_TIME
- EDITION
- SQL_TRANSLATION_PROFILE
- ROW ARCHIVAL VISIBILITY



- ROLES
- CLIENT_INFO

Note:

For Oracle Database Release 19c, FAILOVER_RESTORE is automatically enabled in Transparent Application Continuity (TAC) mode.

The following states are excluded from the auto-restoration option because they are not supported by the Thin driver:

- NLS COMP
- CALL_COLLECT_TIME

For many applications, enabling FAILOVER_RESTORE is sufficient to automatically restore the initial state required for AC replay, without the use of a callback. If your application requires any initial state that is not mentioned in the preceding list, or if the application prefers explicit control over setting the initial state, then the application must use a callback, either connection labeling or an initialization callback. When a callback is configured, it overrides the initial states restored by FAILOVER RESTORE, in case the latter is enabled at the same time.

32.8 About Delaying the Reconnection in Application Continuity for Java

By default, when JDBC Replay Data Source initiates a failover, the driver attempts to recover the in-flight work at an instance where the service is available. For doing this, the driver must first reestablish a good connection to a working instance. This reconnection can take some time if the database or the instance needs to be restarted before the service is relocated and published. So, the failover should be delayed until the service is available from another instance or database.

You must use the FAILOVER_RETRIES and FAILOVER_DELAY parameters to sustain the delay because maximum delay is calculated as FAILOVER_RETRIES multiplied by FAILOVER_DELAY. These parameters can work well in conjunction with a planned outage, for example, an outage that may make a service unavailable for several minutes. While setting the FAILOVER_DELAY and FAILOVER_RETRIES parameters, check the value of the REPLAY_INITIAITION_TIMEOUT parameter first. The default value for this parameter is 900 seconds. A high value for the FAILOVER_DELAY parameter can cause replay to be canceled.

Parameter Name	Possible Value	Default Value
FAILOVER_RETRIES	Positive integer zero or above	30
FAILOVER_DELAY	Time in seconds	10

32.8.1 Configuration Examples Related to Application Continuity for Java

This section provides configuration examples for service creation and modification in the following subsections:



- Creating Services on Oracle RAC
- Modifying Services on Single-Instance Databases

32.8.1.1 Creating Services on Oracle RAC

If you are using Oracle RAC or Oracle RAC One, then use the SRVCTL command to create and modify services in the following way:

For Transparent Application Continuity

You can create services that use Transparent Application Continuity, as follows:

For policy-managed databases:

```
$ srvct1 add service -db codedb -service GOLD -serverpool ora.Srvpool -
clbgoal SHORT -rlbgoal SERVICE_TIME -failover_restore AUTO -
failoverretry 30
-failoverdelay 10 -commit_outcome TRUE -failovertype AUTO -
replay init time 1800 -retention 86400 -notification TRUE
```

For administrator-managed databases:

```
$ srvctl add service -db codedb -service GOLD -preferred serv1 -
available serv2 -clbgoal SHORT -rlbgoal SERVICE_TIME -failover_restore
AUTO
-failoverretry 30 -failoverdelay 10 -commit_outcome TRUE -failovertype
AUTO -replay init time 1800 -retention 86400 -notification TRUE
```

For Manual Application Continuity

You can create services that use manual Application Continuity, as follows:

For policy-managed databases:

```
$ srvctl add service -db codedb -service GOLD -serverpool ora.Srvpool -
clbgoal SHORT -rlbgoal SERVICE_TIME -failover_restore LEVEL1 -
failoverretry 30
-failoverdelay 10 -commit_outcome TRUE -failovertype TRANSACTION -
replay init time 1800 -retention 86400 -notification TRUE
```

For administrator-managed databases:

```
$ srvctl add service -db codedb -service GOLD -preferred serv1 -
available serv2 -clbgoal SHORT -rlbgoal SERVICE_TIME -failover_restore
LEVEL1
-failoverretry 30 -failoverdelay 10 -commit_outcome TRUE -failovertype
TRANSACTION -replay_init_time 1800 -retention 86400 -notification TRUE
```

32.8.1.2 Modifying Services on Single-Instance Databases

If you are using a single-instance database, then use the DBMS_SERVICE package to modify services in the following way:



```
declare
params dbms_service.svc_parameter_array;
begin
params('FAILOVER_TYPE'):='TRANSACTION';
params('REPLAY_INITIATION_TIMEOUT'):=1800;
params('RETENTION_TIMEOUT'):=604800;
params('FAILOVER_DELAY'):=10;
params('FAILOVER_RETRIES'):=30;
params('commit_outcome'):='true';
params('aq_ha_notifications'):='true';
dbms_service.modify_service('[your service]',params);
end;
'
```

32.9 About Retaining Mutable Values in Application Continuity for Java

A mutable object is a variable, function return value, or other structure that returns a different value each time that it is called. For example, Sequence.NextVal, SYSDATE, SYSTIMESTAMP, and SYS_GUID. To retain the function results for named functions at replay, the DBA must grant KEEP privileges to the user who invokes the function. This security restriction is imposed to ensure that it is valid for replay to save and restore function results for code that is not owned by that user.

See Also:

Oracle Database Development Guide

32.9.1 Grant and Revoke Interface

You can work with mutables values by using the standard GRANT and REVOKE interfaces in the following way:

- Dates and SYS_GUID Syntax
- Sequence Syntax
- GRANT ALL Statement
- Rules for Grants on Mutable Values

32.9.1.1 Dates and SYS_GUID Syntax

The DATE_TIME and SYS_GUID syntax is as follows:

GRANT [KEEP DATE TIME|SYSGUID]..[to USER} REVOKE [KEEP DATE TIME | KEEP SYSGUID] ... [from USER]

For example, for EBS standard usage with original dates

Grant KEEP DATE TIME, KEEP SYSGUID to [custom user]; Grant KEEP DATE TIME, KEEP SYSGUID to [apps user];



32.9.1.2 Sequence Syntax

The Sequence syntax can be of the following types:

- Owned Sequence Syntax
- Others Sequence Syntax

Owned Sequence Syntax

ALTER SEQUENCE [sequence object] [KEEP|NOKEEP];

This command retains the original values of sequence.nextval for replaying, so that the keys match after replay. Most applications need to retain the sequence values at replay. The ALTER SYNTAX is only for owned sequences.

Others Sequence Syntax

GRANT KEEP SEQUENCE..[to USER] on [sequence object]; REVOKE KEEP SEQUENCE ... [from USER] on [sequence object];

For example, use the following command for EBS standard usage with original sequence values:

Grant KEEP SEQUENCE to [apps user] on [sequence object]; Grant KEEP SEQUENCE to [custom user] on [sequence object];

32.9.1.3 GRANT ALL Statement

The GRANT ALL statement grants KEEP privilege on all the objects of a user. However, it excludes mutable values, that is, mutable values require explicit grants.

32.9.1.4 Rules for Grants on Mutable Values

Follow these rules while granting privileges on mutable objects:

- If a user has KEEP privilege granted on mutables values, then the objects inherit mutable access when the SYS_GUID, SYSDATE, and SYSTIMESTAMP functions are called.
- If the KEEP privilege on mutable values on a sequence object is revoked, then SQL or PL/SQL blocks using that object will not allow mutable collection or application for that sequence.
- If granted privileges are revoked between runtime and failover, then the mutable values that are collected are not applied for replay.
- If new privileges are granted between runtime and failover, mutable values are not collected and these values are not applied for replay.

32.10 Application Continuity Statistics

The JDBC Replay Data Source supports the following statistics for an application using Application Continuity:

• Total number of requests



- Total number of completed requests
- Total number of calls
- Total number of protected calls
- Total number of calls affected by outages
- Total number of calls triggering replay
- Total number of calls affected by outages during replay
- Total number of successful replay
- Total number of failed replay
- Total number of disabled replay
- Total number of replay attempts

All these metrics are available both on a per-connection basis and across-connections basis. You can use the following methods for obtaining these statistics:

- getReplayStatistics(StatisticsReportType)
 - Use the

oracle.jdbc.replay.ReplayableConnection.getReplayStatistics(StatisticsReport Type) method to obtain the snapshot statistics. The argument to this method is an enum type also defined in the same ReplayableConnection interface. To obtain statistics across connections, it is best calling this method after the main application logic. Applications can either use any oracle.jdbc.replay.ReplayableConnection that is still open, or open a new connection to the same data source. This applies to applications using both UCP and WLS data sources, and applications that directly use the replay data source.

getReplayStatistics()

Use the oracle.jdbc.replay.OracleDataSource.getReplayStatistics() method to obtain across-connection statistics. This applies only to applications that directly use replay data source.

Both methods return an oracle.jdbc.replay.ReplayStatistics object, from which you can retrieve individual replay metrics. The following is a sample output that prints a ReplayStatistics object as String:

```
AC Statistics:
_____
TotalRequests = 1
TotalCompletedRequests = 1
TotalCalls = 19
TotalProtectedCalls = 19
_____
TotalCallsAffectedByOutages = 3
TotalCallsTriggeringReplay = 3
TotalCallsAffectedByOutagesDuringReplay = 0
_____
SuccessfulReplayCount = 3
FailedReplayCount = 0
ReplayDisablingCount = 0
TotalReplayAttempts = 3
_____
```



If you want to clear the accumulated replay statistics per connection or for all connections, then you can use the following methods:

- oracle.jdbc.replay.ReplayableConnection.clearReplayStatistics(Replayab leConnection.StatisticsReportType reportType)
- oracle.jdbc.replay.OracleDataSource.clearReplayStatistics()

Note:

All statistics reflect only updates since the latest clearing.

32.11 About Disabling Replay in Application Continuity for Java

This section describes the following concepts:

- How to Disable Replay
- When to Disable Replay
- Diagnostics and Tracing

32.11.1 How to Disable Replay

If any application module uses a design that is unsuitable for replay, then the disable replay API disables replay on a per request basis. Disabling replay can be added to the callback or to the main code by using the disableReplay method of the oracle.jdbc.replay.ReplayableConnection interface. For example:

```
if (connection instanceof oracle.jdbc.replay.ReplayableConnection)
{
    (( oracle.jdbc.replay.ReplayableConnection)connection).disableReplay();
}
```

Disabling replay does not alter the connection state by reexecuting any JDBC method, SQL or PL/SQL. When replay is disabled using the disable replay API, both recording and replay are disabled until that request ends. There is no API to reenable replay because it is invalid to reestablish the database session with time gaps in a replayed request. This ensures that replay runs *only* if a complete history of needed calls has been recorded.

32.11.2 When to Disable Replay

By default, the JDBC Replay Data Source replays following a recoverable error. The disable replay API can be used in the entry point of application modules that are unable to lose the database sessions and recover. For example, if the application uses the UTL_SMTP package and does not want messages to be repeated, then the disableReplay API affects only the request that needs to be disabled. All other requests continue to be replayed.

The following are scenarios to consider before configuring an application for replay:



- Application Calls External PL/SQL Actions that Should not Be Repeated
- Application Synchronizes Independent Sessions
- Application Uses Time at the Middle-tier in the Execution Logic
- Application assumes that ROWIds do not change
- Application Assumes that Side Effects Execute Once
- Application Assumes that Location Values Do not Change

32.11.2.1 Application Calls External Systems that Should not Be Repeated

During replay, autonomous transactions and external systems (like PL/SQL calls or Java calls) can have side effects that are separate from the main transaction. These side effects are replayed unless you specify otherwise and leave persistent results behind. These side effects include writing to an external table, sending email, forking sessions out of PL/SQL or Java, transferring files, accessing external URLs, and so on. For example, in case of PL/SQL messaging, suppose, you walk away in-between some work without committing and the session times out. Now, if you issue a Ctrl+C command, then the foreground of a component fails. When you resubmit the work, then this side effect can also be repeated.

See Also:

Oracle Real Application Clusters Administration and Deployment Guide for more information about potential side effects of Application Continuity

You must make a conscious decision about whether to enable replay for external actions or not. For example, you can consider the following situations where this decision is important:

- Using the UTL HTTP package to issue a SOA call
- Using the UTL SMTP package to send a message
- Using the UTL URL package to access a web site

Use the disableReplay API if you do not want such external actions to be replayed.

32.11.2.2 Application Synchronizes Independent Sessions

You can configure an application for replay if the application synchronizes independent sessions using volatile entities that are held until commit, rollback, or session loss. In this case, the application synchronizes multiple sessions connected to several data sources that are otherwise inter-dependent using resources such as a database lock. This synchronization may be fine if the application is only serializing these sessions and understands that any session may fail. However, if the application assumes that a lock or any other volatile resource held by one data source implies exclusive access to data on the same or a separate data source from other connections, then this assumption may be invalidated when replaying.

During replay, the driver is not aware that the sessions are dependent on one session holding a lock or other volatile resource. You can also use pipes, buffered queues, stored procedures taking a resource (such as a semaphore, device, or socket) to implement the synchronization that are lost by failures.





32.11.2.3 Application Uses Time at the Middle-tier in the Execution Logic

In this case, the application uses the wall clock at the middle-tier as part of the execution logic. The JDBC Replay Data Source does not repeat the middle-tier time logic, but uses the database calls that execute as part of this logic. For example, an application using middle-tier time may assume that a statement executed at Time T1 is not reexecuted at Time T2, unless the application explicitly does so.

32.11.2.4 Application assumes that ROWIds do not change

If an application caches ROWIDs, then access to these ROWIDs may be invalidated due to database changes. Although a ROWID uniquely identifies a row in a table, a ROWID may change its value in the following situations:

- The underlying table is reorganized
- An index is created on the table
- The underlying table is partitioned
- The underlying table is migrated
- The underlying table is exported and imported using EXP/IMP/DUL
- The underlying table is rebuilt using Golden Gate or Logical Standby or other replication technology
- The database of the underlying table is flashed back or restored

It is bad practice for an application to store ROWIDs for later use as the corresponding row may either not exist or contain completely different data.

32.11.2.5 Application Assumes that Side Effects Execute Once

In this case, the following are replayed during a replay:

- Autonomous transactions
- Opening of back channels separate to the main transaction side effects

Examples of back channels separate to the main transaction include writing to an external table, sending email, forking sessions out of PL/SQL or Java, writing to output files, transferring files, and writing exception files. Any of these actions leave persistent side effects in the absence of replay. Back channels can leave persistent results behind. For example, if a user leaves a transaction midway without committing and the session times out, then the user presses Ctrl+C, the foreground or any component fails. If the user resubmits work, then the side effects can be repeated.

32.11.2.6 Application Assumes that Location Values Do not Change

SYSCONTEXT options comprise a location-independent set such as National Language Support (NLS) settings, ISDBA, CLIENT_IDENTIFIER, MODULE, and ACTION, and a location-dependent set that uses physical locators. Typically, an application does not



use the physical identifier, except in testing environments. If physical locators are used in mainline code, then the replay finds the mismatch and rejects it. However, it is fine to use physical locators in callbacks.

Example

```
select
    sys_context('USERENV', 'DB_NAME')
    ,sys_context('USERENV', 'HOST')
    ,sys_context('USERENV', 'INSTANCE')
    ,sys_context('USERENV', 'IP_ADDRESS')
    ,sys_context('USERENV', 'ISDBA')
    ,sys_context('USERENV', 'SESSIONID')
    ,sys_context('USERENV', 'TERMINAL')
    ,sys_context('USERENV', ID')
from dual
```

32.11.3 Diagnostics and Tracing

The JDBC Replay Data Source supports standard JDK logging. Logging is enabled using the Java command-line -Djava.util.logging.config.file=<file> option. Log level is controlled with the oracle.jdbc.level attribute in the log configuration file. For example:

oracle.jdbc.level = FINER|FINEST

where, FINER produces external APIs and FINEST produces large volumes of trace. You must use FINEST only under supervision.

If you use the java.util.logging.XMLFormatter class to format a log record, then the logs are more readable but larger. If you are using replay with FAN enabled on UCP or WebLogic Server, then you should also enable FAN-processing logging.

See Also: Oracle Universal Connection Pool for JDBC Developer's Guide

32.11.3.1 Writing Replay Trace to Console

Following is the example of a configuration file for logging configuration.

```
oracle.jdbc.level = FINER
handlers = java.util.logging.ConsoleHandler
java.util.logging.ConsoleHandler.level = ALL
java.util.logging.ConsoleHandler.formatter = java.util.logging.XMLFormatter
```

32.11.3.2 Writing Replay Trace to a File

Following is the example of a properties file for logging configuration.

oracle.jdbc.level = FINEST

```
# Output File Format (size, number and style)
# count: Number of output files to cycle through, by appending an integer to the base
file name:
# limit: Limiting size of output file in bytes
```



```
handlers = java.util.logging.FileHandler
java.util.logging.FileHandler.pattern = [file location]/replay_%U.trc
java.util.logging.FileHandler.limit = 500000000
java.util.logging.FileHandler.count = 1000
java.util.logging.FileHandler.formatter = java.util.logging.XMLFormatter
```

See Also:

Restrictions and Other Considerations for Application Continuity

33 Oracle JDBC Support for FAN Events

Starting from Oracle Database 12c Release 2 (12.2.0.1), Oracle JDBC driver supports Oracle RAC Fast Application Notification (FAN) events, for planned and unplanned outages. This facilitates third-party connection pools to leverage Oracle RAC features for high availability. Java applications not using Oracle Universal Connection Pool (UCP) or WebLogic Server can now leverage on this support. For example, scenarios like rolling upgrades at the Oracle RAC server-side do not cause JDBC errors within applications.

Note:

Although the Oracle JDBC drivers now support the FAN events, Oracle UCP provides more comprehensive support for all FAN events.

See Also:

Oracle Universal Connection Pool Developer's Guide

- Overview of Oracle JDBC Support for FAN events
- Safe Draining APIs for Planned Maintenance
- Installation and Configuration of Oracle JDBC Driver for FAN Events Support
- Example of Oracle JDBC Driver FAN support for Planned Maintenance
- Using Third-Party Connection Pools with Oracle JDBC

33.1 Overview of Oracle JDBC Support for FAN events

You must use an Oracle RAC Database or an Oracle Restart on a single instance database to use this feature.

This feature supports:

Planned maintenance

This case deals with planned maintenance on Oracle RAC servers, where an Oracle RAC service can be gracefully shutdown. In this case, borrowed or in-use connections from a connection pool are not interrupted, and are closed only until any safe-draining API is invoked. For example, when an application completes work on such a connection and returns it to the connection pool.

Unplanned outages

In this case, dead connections are rapidly detected and terminated, so that the network connections to the server do not become nonresponsive.. In this case, borrowed and inuse connections are interrupted during unplanned outages. Applications are expected to



handle any exception on affected connections and perform necessary recovery, either on their own, or using Oracle high-availability solutions such as Application Continuity.

Related Topics

Application Continuity for Java

The outages of the underlying software, hardware, communications, and storage layers can cause application execution to fail. In the worst cases, the middle-tier servers may need to be restarted to deal with a logon storm, which is a sudden increase in the number of client connection requests.

Related Topics

Safe Draining APIs for Planned Maintenance

🖍 See Also:

Oracle Real Application Clusters Administration and Deployment Guidefor server-side configuration about using Oracle RAC. This chapter describes only the client-side configuration steps that an application must perform when using Oracle JDBC driver support for FAN events.

33.2 Safe Draining APIs for Planned Maintenance

For planned Oracle RAC maintenance, the JDBC driver supports a list of safe-draining APIs, which are required for additional handshake or integration work with a third-party Java connection pool. These APIs serve as the *draining-points*, where the driver can safely close any connection affected by a planned maintenance, without causing application-visible errors. Following is the list of safe-draining APIs that driver FAN supports:

- java.sql.Connection.isValid(int timeout)
- oracle.jdbc.OracleConnection.pingDatabase()
- oracle.jdbc.OracleConnection.pingDatabase(int timeout)
- oracle.jdbc.OracleConnection.endRequest()
- All standard JDBC and Oracle JDBC extension EXECUTE*** calls on Statement, PreparedStatement, and CallableStatement interfaces

For the standard JDBC and Oracle JDBC extension EXECUTE*** calls, the executed SQL command string must contain the following SQL hint as the first noncomment token within the SQL string:

```
/*+ CLIENT CONNECTION VALIDATION */
```

Qualified SQLs are treated as connection-validation SQLs. For example:

```
/*+ CLIENT CONNECTION VALIDATION */ SELECT 1 FROM DUAL
```



Typically, a third-party connection pool places calls to these APIs. It is expected that on detection of any bad connection with such invocations, a third-party connection pool closes and removes the related connection from the pool, so that no errors are visible to applications. When the application itself calls these APIs, then it is expected that application is actively validating the underlying connection and will close and remove any bad connection detected.

33.3 Installation and Configuration of Oracle JDBC Driver for FAN Events Support

Oracle JDBC driver automatically determines whether to enable Oracle JDBC support for FAN events, by checking the database server that it connects to, and whether the following necessary JAR files are available in the application environment, in addition to the Oracle JDBC driver or not:

The simplefan.jar and ons.jar files

You must install the 12.2.0.1 version of the simplefan.jar and ons.jar files from the following link and include them in the CLASSPATH.

JDBC and UCP Downloads page

If either one is missing, or the driver is unable to load it, then this feature is disabled. When used with a third-party connection pool, these JAR files must be placed in the same location, where the connection pool retrieves and loads the driver JAR files.

Oracle JDBC data sources

You can use the same typical Oracle JDBC data sources, such as oracle.jdbc.pool.OracleDataSource or oracle.jdbc.OracleDriver for obtaining JDBC connections. When used together with a third-party connection pool, your application must specify these classes as connection factories for the connection pool.

Applications that want to explicitly disable this feature, can set the oracle.jdbc.fanEnabled property to FALSE. This property is available as both a system property and a connection property. For applications using Universal Connection Pool (UCP) or WebLogic Server Active GridLink (AGL), this property is set to FALSE by default. Otherwise, the default value is TRUE.

Note:

- When the JDBC driver automatically enables support for FAN events, with both the simplefan.jar and the ons.jar files present on the CLASSPATH, then calling the getConnection method may throw an exception, such as, java.lang.IllegalArgumentException. To avoid this, you can perform either of the following:
 - Remove either simplefan.jar or ons.jar from the CLASSPATH.
 - Set the oracle.jdbc.fanEnabled property to FALSE to disable this feature explicitly.
- Setting the oracle.jdbc.fanEnabled property to TRUE may not enable Oracle JDBC Support for FAN Events feature as the feature depends on other factors too.



The JDBC driver requires minimal configuration changes or code changes to a thirdparty connection pool for supporting Oracle FAN events. For a connection pool that does not need any configuration change or code change, it is assumed that it fulfils one of the following criteria:

- The pool has a configuration option for validating JDBC connections at pool checkout time.
- The pool uses javax.sql.PooledConnection and has a configuration option for plugging in a javax.sql.ConnectionPoolDataSource implementation. Such a connection pool is also assumed to be able to check for closed or bad physical connections at connection returns.

Following are a few connection validation options on some third-party Java connection pools. The majority of these options are based on SQL, and not on validation APIs:

Java Connection Pool	Connection Validation Options
Oracle WebLogic Generic and MDS data sources	TestConnectionsOnReserve, TestConnectionsOnRelease, TestConnectionsOnCreate
IBM WebSphere	PreTest Connection
RedHat JBoss	check-valid-connection-sql
Apache TomCat	TestonBorrow, TestonRelease

When you use Oracle JDBC support for FAN events feature with an Oracle RAC server Release 11*g*, then applications must explicitly set the remote ONS configuration string for Oracle RAC FAN through the oracle.jdbc.fanONSConfig system property. The value and the format of the property are the same as for UCP Fast Connection Failover (FCF).

See Also:

Universal Connection Pool Developer's Guide

33.4 Example of Oracle JDBC Driver FAN support for Planned Maintenance

The following example illustrates how you can typically enable and use JDBC Oracle FAN support with planned maintenance on Oracle RAC. Applications should not receive any exception during a planned maintenance after following these instructions:

- 1. Upgrade Oracle JDBC driver to Release 12.2.0.1 to use the ojdbc8.jar file.
- 2. Install and use the 12.2.0.1 version of the ons.jar and simplefan.jar files.
- 3. Use the oracle.jdbc.pool.OracleDataSource class to obtain physical connections or configure this class as the connection factory on a third-party Java connection pool. In the latter case, you must set the specific pool property that enables connection validation.



Optionally, when running against Oracle RAC Release 11g, specify the system property oracle.jdbc.fanONSConfig to configure remote ONS.

- 4. The application runs until you are ready to perform planned maintenance activities like a rolling upgrade on the Oracle RAC. During the planned maintenance, for each service based on usage patterns, a DBA will perform the following activities using the extended 12.2 srvctl interface:
 - Relocate or stop the services on the next instance to upgrade, with no -f (force)
 - Wait until all connections to this service are drained by driver FAN
 - When the timeout is reached, disconnect the sessions with the defined stop mode (transactional is recommended)
 - When all services are relocated or stopped, shutdown the instance and apply the upgrade or patch
 - Restart the instance and restart the services if they were stopped
 - Iterate until all instances are upgraded/patched

33.5 Using Third-Party Connection Pools with Oracle JDBC

This section lists the requirements to configure third-party connection pools for supporting planned and unplanned outages.

For configuring the third-party connection pools, ensure that you:

- Enable High Availability (HA) support in the driver
- Enforce connection validation while borrowing the connection
- Tune the connection validation technique to limit the cost
- Enable statement caching in the driver

With the preceding settings, the connections in the third-party connection pools use the draining APIs during a planned down event, for example, when a service is stopped on an instance. For handling unplanned outages, you must also enable Transparent Application Continuity (TAC) on the database service and configure one of the following replay data sources:

- oracle.jdbc.datasource.impl.OracleDataSource with Oracle Databases 21c Release or later
- oracle.jdbc.replay.OracleDataSource with Oracle Database 19c Release

Example

For example, if you want to support planned outages with HikariCP, then you must take care of the following settings:

- Ensure HA support: Include the ons.jar and simplefan.jar files in the classpath
- Ensure connection validation:

```
com.zaxxer.hikari.aliveBypassWindowMs=-1
```



• Tune the connection validation technique:

oracle.jdbc.defaultConnectionValidation=LOCAL

• Enable statement caching in the driver:

oracle.jdbc.implicitStatementCacheSize=50



34 Transparent Application Failover

This chapter contains the following sections:

- Overview of Transparent Application Failover
- Failover Type Events
- TAF Callbacks
- Java TAF Callback Interface
- Comparison of TAF and Fast Connection Failover

34.1 Overview of Transparent Application Failover

Transparent Application Failover (TAF) is a feature of the Java Database Connectivity (JDBC) Oracle Call Interface (OCI) driver. It enables the application to automatically reconnect to a database, if the database instance to which the connection is made fails. In this case, the active transactions roll back.

When an instance to which a connection is established fails or is shut down, the connection on the client-side becomes stale and would throw exceptions to the caller trying to use it. TAF enables the application to transparently reconnect to a preconfigured secondary instance, creating a fresh connection, but identical to the connection that was established on the first original instance. That is, the connection properties are the same as that of the earlier connection. This is true regardless of how the connection was lost.

Note:

- TAF is always active and does not have to be set.
- TAF is not supported with LOB and XML types.

34.2 Failover Type Events

The following are possible failover events in the OracleOCIFailover interface:

• FO SESSION

Is equivalent to FAILOVER_MODE=SESSION in the tnsnames.ora file CONNECT_DATA flags. This means that only the user session is authenticated again on the server side, while open cursors in the OCI application need to be reprocessed.

• FO SELECT

Is equivalent to FAILOVER_MODE=SELECT in tnsnames.ora file CONNECT_DATA flags. This means that not only the user session is re-authenticated on the server side, but open cursors in the OCI can continue fetching. This implies that the client-side logic maintains fetch-state of each open cursor.



FO_NONE

Is equivalent to FAILOVER_MODE=NONE in the tnsnames.ora file CONNECT_DATA flags. This is the default, in which no failover functionality is used. This can also be explicitly specified to prevent failover from happening. Additionally, FO TYPE UNKNOWN implies that a bad failover type was returned from the OCI driver.

FO_BEGIN

Indicates that failover has detected a lost connection and failover is starting.

• FO_END

Indicates successful completion of failover.

FO_ABORT

Indicates that failover was unsuccessful and there is no option of retrying.

• FO_REAUTH

Indicates that a user handle has been re-authenticated.

FO ERROR

Indicates that failover was temporarily unsuccessful, but it gives the application the opportunity to handle the error and retry failover. The usual method of error handling is to issue the sleep method and retry by returning the value FO RETRY.

• FO RETRY

Indicates that the application should retry failover.

• FO_EVENT_UNKNOWN

Indicates a bad failover event.

34.3 TAF Callbacks

TAF callbacks are used in the event of the failure of one database connection, and failover to another database connection. TAF callbacks are callbacks that are registered in case of failover. The callback is called during the failover to notify the JDBC application of events generated. The application also has some control of failover.

Note:

The callback setting is optional.

34.4 Java TAF Callback Interface

The OracleOCIFailover interface includes the callbackFn method, supporting the following types and events:

public interface OracleOCIFailover{

```
// Possible Failover Types
public static final int FO_SESSION = 1;
```



Handling the FO_ERROR Event

In case of an error while failing over to a new connection, the JDBC application is able to retry failover. Typically, the application sleeps for a while and then it retries, either indefinitely or for a limited amount of time, by having the callback return FO RETRY.

Handling the FO_ABORT Event

Callback registered should return the FO ABORT event if the FO ERROR event is passed to it.

34.5 Comparison of TAF and Fast Connection Failover

Transparent Application Failover (TAF) differs from Fast Connection Failover in the following ways:

Application-level connection retries

TAF supports connection retries only at the OCI/Net layer. Fast Connection Failover supports application-level connection retries. This gives the application control of responding to connection failovers. The application can choose whether to retry the connection or to rethrow the exception.

Integration with the Universal Connection Pool

TAF works at the network level on a per-connection basis, which means that the connection cache cannot be notified of failures. Fast Connection Failover is well-integrated with the Universal Connection Pool, which enables the Connection Cache Manager to manage the cache for high availability. For example, failed connections are automatically invalidated in the cache.

Event-based

Fast Connection Failover is based on the Oracle RAC event mechanism. This means that Fast Connection Failover is efficient and detects failures quickly for both active and inactive connections.

Load-balancing support

Fast Connection Failover supports UP event load balancing of connections and run-time work request distribution across active Oracle RAC instances.



See Also:

Oracle Universal Connection Pool for JDBC Developer's Guide

Note:

Oracle recommends *not* to use TAF and Fast Connection Failover in the same application.



35 Single Client Access Name

Single Client Access Name (SCAN) is an Oracle Real Application Clusters (Oracle RAC) feature that provides a single name for clients to access Oracle Databases running in a cluster. This chapter discusses the following concepts related to the SCAN:

- Overview of Single Client Access Name
- About Configuring the Database Using the SCAN
- Using the SCAN in a Maximum Availability Architecture Environment
- Using the SCAN With Oracle Connection Manager

35.1 Overview of Single Client Access Name

Note:

The Grid Naming Service (GNS) feature of Oracle Grid Infrastructure is deprecated in Oracle Database 23c.

The SCAN is a domain name registered to at least one and up to three IP addresses, either in Domain Naming Service (DNS) or Grid Naming Service (GNS). When you use GNS and Dynamic Host Configuration Protocol (DHCP), Oracle Clusterware configures the Virtual IP (VIP) addresses for the SCAN name that is provided during cluster configuration. The node VIP and the three SCAN VIPs are obtained from the DHCP server when you use GNS.

🖍 See Also:

Oracle Clusterware Administration and Deployment Guide for more information about GNS

If a new server joins the cluster, then Oracle Clusterware dynamically obtains the required VIP address from the DHCP server, updates the cluster resource, and makes the server accessible through GNS. The benefit of using the SCAN is that the connection information of the client does not need to change if you add or remove nodes in the cluster. Having a single name to access the cluster enables the client to use the EZConnect client and the simple JDBC thin URL to access any Database running in the cluster, independent of the active servers in the cluster. The SCAN provides load balancing and failover for client connections to the Database. The SCAN works as a cluster alias for Databases in the cluster.



35.2 About Configuring the Database Using the SCAN

The SCAN is an essential part of Database configuration. So, by default, the REMOTE_LISTENER parameter is set to the SCAN, assuming that the Database is created using standard Oracle tools.

This enables the instances to register with the SCAN Listeners as remote listeners to provide information on what services are being provided by the instance, the current load, and a recommendation on how many incoming connections should be directed to the instance.

In this context, you must set the LOCAL_LISTENER parameter to the node-VIP. If you need fully qualified domain names, then ensure that the LOCAL_LISTENER parameter is set to the fully qualified domain name. By default, a node listener is created on each node in the cluster during cluster configuration. With Oracle Grid Infrastructure, the node listener runs out of the Oracle Grid Infrastructure home and listens on the node-VIP using the specified port. The default port is 1521.

Unlike in earlier Database versions, Oracle does not recommend to set your REMOTE_LISTENER parameter to a server side TNSNAMES alias that resolves the host to the SCAN in the address list entry, for example, HOST=sales1-scan. Instead, you must use the simplified SCAN:port syntax as shown in the following table that shows typical setting for a LOCAL_LISTENER and REMOTE_LISTENER:

Name	Туре	Value
LOCAL_LISTENER	string	<pre>(DESCRIPTION=(ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCP)(HOST=localhost) (PORT=5221))))</pre>
REMOTE_LISTENER	string	myoracleexample.com:5221

Note:

If you are using the easy connect naming method, you may need to modify the SQLNET.ORA file to ensure that EZCONNECT is in the list when you specify the order of the naming methods used for the client name resolution lookups.

35.3 How Connection Load Balancing Works Using the SCAN

For clients connecting using Oracle SQL*Net, three IP addresses are received by the client by resolving the SCAN name through DNS. The client then goes through the list that it receives from the DNS and tries connecting through one of the IP addresses in the list. If the client receives an error, then it tries connecting to the other addresses before returning an error to the user or application. This is similar to how client connection failover works in earlier Database releases, when an address list is provided in the client connection string.

When a SCAN Listener receives a connection request, the SCAN Listener checks for the least loaded instance providing the requested service. It then re-directs the



connection request to the local listener on the node where the least loaded instance is running. Subsequently, the client is given the address of the local listener. The local listener then finally creates the connection to the Database instance.

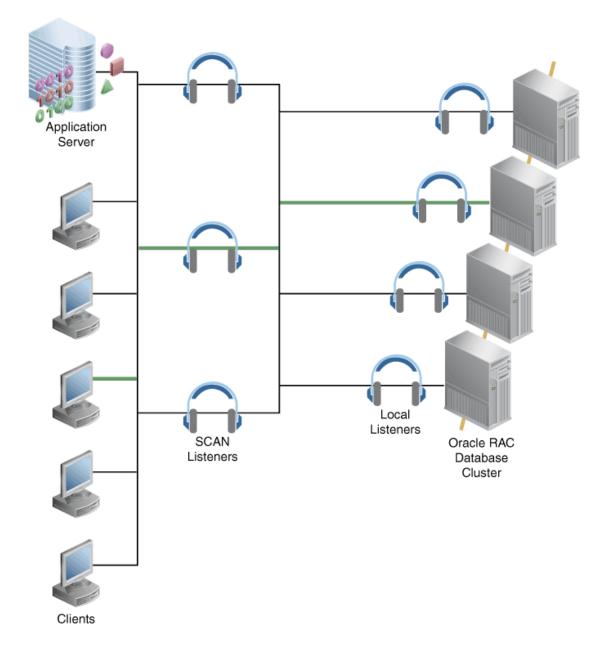


Figure 35-1 Connection Load Balancing Using the SCAN

Example

This example assumes an Oracle client using a default **TNSNAMES.ora** file:

ORCLservice = (DESCRIPTION = (ADDRESS = (PROTOCOL = TCP) (HOST = sales1-scan.example.com)
(PORT = 1521)) (CONNECT_DATA = (SERVER = DEDICATED) (SERVICE_NAME = MyORCLservice)))



35.4 Version and Backward Compatibility

This section describes how to use the SCAN for connecting to an Oracle RAC Database.

To successfully use the SCAN to connect to an Oracle RAC Database in the cluster depends on the following two factors:

- Ability of the client to understand and use the SCAN
- The correct configuration of the REMOTE_LISTENER parameter setting in the Database

If the version of the Oracle Client connecting to the Database and the Oracle Database version used are both Oracle Database 11g Release 2, and the default configuration is used as described in the preceding sections, then typically you do not need to make any change to the system.

If both the Oracle Client version and the version of the Oracle Database that the client is connecting to are earlier than Oracle Database 11g Release 2, then typically you do not need to make any change to the system. In this case, the client uses a TNS connect descriptor that resolves to the node-VIPs of the cluster, while Oracle Database uses a REMOTE_LISTENER entry pointing to the node-VIPs. The disadvantage of this configuration is that the SCAN is not used and therefore every time the cluster changes in the back end, the clients are exposed to changes.

If you are using Oracle Database 11g Release 2, but the clients are on an earlier version of the Database, then you must change the Oracle client, or the Oracle Database REMOTE_LISTENER parameter settings, or both accordingly. You must consider the following cases in such a scenario:

Oracle Client Version	Oracle Database Version	Comment
Oracle Database 11g Release 2	Oracle Database 11g Release 2	No change is required
Oracle Database 11 <i>g</i> Release 2	Oracle Database Version earlier than Oracle Database 11g Release 2	Add the SCAN VIPs as hosts to the REMOTE_LISTENER parameter.
Oracle Database Version earlier than Oracle Database 11 <i>g</i> Release 2	Oracle Database 11 <i>g</i> Release 2	Update the client TNSNAMES.ora file to include the SCAN VIPs. If the Database is upgraded using the Database AutoUpgrade from a Database earlier than 11g Release 2, then the Database AutoUpgrade configures the REMOTE_LISTENER parameter to point to the node-VIPs and the SCAN.

Table 35-1Oracle Client and Oracle Database Version Compatibility for theSCAN



Oracle Client Version	Oracle Database Version	Comment
Oracle Database Version earlier than Oracle Database 11 <i>g</i> Release 2	Oracle Database Version earlier than Oracle Database 11 <i>g</i> Release 2	If you want to make use of the SCAN (recommended), then add the SCAN VIPs as hosts to the REMOTE_LISTENER parameter and update the client TNSNAMES.ora file to include the SCAN VIPs. Otherwise, no change required.

Table 35-1	(Cont.) Oracle Client and Oracle Database Version Compatibility for
the SCAN	

If you are using a client earlier than Oracle Database 11g Release 2, then you cannot fully benefit from the advantages of the SCAN because the Oracle Client cannot handle a set of three IP addresses returned by the DNS for the SCAN. Instead, it tries to connect to only the first address returned in the list and ignores the other two addresses. If the SCAN Listener listening on this specific IP address is not available or the IP address itself is not available, then the connection fails. To ensure load balancing and connection failover with clients earlier than Oracle Database 11g Release 2, you must update the TNSNAMES.ora file of the client, so that it uses three address lines, where each address line resolves to one of the SCAN VIPs. The following example shows a sample TNSNAMES.ora file for a client earlier than Oracle Database 11g Release 2:

```
sales.example.com = (DESCRIPTION=
(ADDRESS_LIST= (LOAD_BALANCE=on) (FAILOVER=ON)
(ADDRESS=(PROTOCOL=tcp) (HOST=133.22.67.192) (PORT=1521))
(ADDRESS=(PROTOCOL=tcp) (HOST=133.22.67.193) (PORT=1521)))
(ADDRESS=(PROTOCOL=tcp) (HOST=133.22.67.194) (PORT=1521)))
(CONNECT DATA=(SERVICE NAME= salesservice.example.com)))
```

35.5 Using the SCAN in a Maximum Availability Architecture Environment

If you have a Maximum Availability Architecture (MAA) environment implemented, in which you use Oracle RAC for both your primary and standby Databases that are synchronized using Oracle Data Guard, then using the SCAN provides a simplified TNSNAMES configuration that a client can use to connect to the Database, independent of whether the primary or standby Database is the currently active Database.

To use this simplified configuration, Oracle Database 11g Release 2 introduced the following two SQL*Net parameters that can be used for connection strings of individual clients:

• The CONNECT TIMEOUT parameter

It specifies the timeout duration in seconds for a client to establish an Oracle Net connection to an Oracle Database. This parameter overrides the <code>SQLNET.OUTBOUT_CONNECT_TIMEOUT</code> parameter in the <code>SQLNET.ORA</code> file.

• The RETRY COUNT parameter

It specifies the number of times an ADDRESS_LIST is traversed before the connection attempt is terminated.



Using these two parameters, both the SCANs, the one on the primary site and the one on the standby site, can be used in the client connection strings. Also, if the randomly selected address points to the site that is not currently active, then the timeout enables the connection request to failover before the client waits for an unreasonably long time. The following example shows a sample TNSNAMES.ORA entry for a MAA environment:

```
sales.example.com = (DESCRIPTION= (CONNECT_TIMEOUT=10) (RETRY_COUNT=3)
(ADDRESS_LIST= (LOAD_BALANCE=on) (FAILOVER=ON)
(ADDRESS=(PROTOCOL=tcp) (HOST=sales1-scan) (PORT=1521))
(ADDRESS=(PROTOCOL=tcp) (HOST=sales2-scan) (PORT=1521)))
(CONNECT DATA=(SERVICE NAME= salesservice.example.com)))
```

35.6 Using the SCAN With Oracle Connection Manager

If you use Oracle Connection Manager (CMAN) with your Oracle RAC Database, then the REMOTE_LISTENER parameter for the Oracle RAC instances must include the CMAN server, so that the CMAN server receives load balancing related information and can load balance connections across the available instances. The easiest way to achieve this is to add the CMAN server as an entry to the REMOTE_LISTENER parameter of the Databases that clients want to connect to through CMAN. You must also remove the SCAN from the TNSNAMES connect descriptor of the clients and configure the CMAN server. The following example shows a server-side TNSNAMES.ora example entry when you use CMAN:

SQL> show parameters listener

NAME	TYPE	VALUE
listener_networks	string	
local_listener	string	(DESCRIPTION=(ADDRESS_LIST=
		(ADDRESS=(PROTOCOL=TCP)
		(HOST=148.87.58.109) (PORT=1521))))
remote_listener	string	<pre>stscan3.oracle.com:1521,(DESCRIPTION=</pre>
		(ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCP)
		(HOST=CMANserver) (PORT=1521))))

See Also:

Oracle Database Net Services Reference for more information about configuring the CMAN server



Part VII Transaction Management

This part provides information about transaction management in Oracle Java Database Connectivity (JDBC). It includes a chapter that discusses the Oracle JDBC implementation of distributed transactions.

Part VII contains the following chapter:

Distributed Transactions



36 Distributed Transactions

This chapter discusses the Oracle Java Database Connectivity (JDBC) implementation of distributed transactions. These are multiphased transactions, often using multiple databases, which must be committed in a coordinated way. There is also related discussion of XA, which is a general standard, and not specific to Java, for distributed transactions.

The following topics are discussed:

- AboutDistributed Transactions
- XA Components
- Error Handling and Optimizations
- About Implementing a Distributed Transaction
- Native-XA in Oracle JDBC Drivers

Note:

This chapter discusses features of the JDBC 2.0 Optional Package, formerly known as the JDBC 2.0 Standard Extension application programming interface (API) that is available through the <code>javax</code> packages.

For further introductory and general information about distributed transactions, refer to the specifications for the JDBC 2.0 Optional Package and the Java Transaction API (JTA).

36.1 About Distributed Transactions

The section covers the following topics:

- Overview of Distributed Transaction
- Distributed Transaction Components and Scenarios
- Distributed Transaction Concepts
- About Switching Between Global and Local Transactions
- Oracle XA Packages

36.1.1 Overview of Distributed Transaction

A **distributed transaction**, sometimes referred to as a **global transaction**, is a set of two or more related transactions that must be managed in a coordinated way. The transactions that constitute a distributed transaction might be in the same database, but more typically are in different databases and often in different locations. Each individual transaction of a distributed transaction branch.



For example, a distributed transaction might consist of money being transferred from an account in one bank to an account in another bank. You would not want either transaction committed without assurance that both will complete successfully.

In JDBC, distributed transaction functionality is built on top of connection pooling functionality. This distributed transaction functionality is also built upon the open XA standard for distributed transactions. XA is part of the X/Open standard and is not specific to Java.

JDBC is used to connect to database resources. However, to include all changes to multiple databases within a transaction, you must use the JDBC connections within a JTA global transaction. The process of including database SQL updates within a transaction is referred to as enlisting a database resource.

36.1.2 Distributed Transaction Components and Scenarios

In reading the remainder of the distributed transactions section, it will be helpful to keep the following points in mind:

 A distributed transaction system typically relies on an external transaction manager, such as a software component that implements standard JTA functionality, to coordinate the individual transactions.

Many vendors offer XA-compliant JTA modules, including Oracle, which includes JTA in Oracle9*i* Application Server and Oracle Application Server 10*g*.

• XA functionality is usually isolated from a client application, being implemented instead in a middle-tier environment, such as an application server.

In many scenarios, the application server and transaction manager will be together on the middle tier, possibly together with some of the application code as well.

- Discussion throughout this section is intended mostly for middle-tier developers.
- The term resource manager is often used in discussing distributed transactions. A
 resource manager is simply an entity that manages data or some other kind of
 resource. Wherever the term is used in this chapter, it refers to a database.

Note:

Using JTA functionality requires jta.jar to be in the CLASSPATH environment variable. This file is located at ORACLE_HOME/jlib. Oracle includes this file with the JDBC product.

36.1.3 Distributed Transaction Concepts

When you use XA functionality, the transaction manager uses XA resource instances to prepare and coordinate each transaction branch and then to commit or roll back all transaction branches appropriately.

XA functionality includes the following key components:

• XA data sources

These are extensions of connection pool data sources and other data sources, and similar in concept and functionality.



There will be one XA data source instance for each resource manager that will be used in the distributed transaction. You will typically create XA data source instances in your middle-tier software.

XA data sources produce XA connections.

XA connections

These are extensions of pooled connections and similar in concept and functionality. An XA connection encapsulates a physical database connection. Individual connection instances are temporary handles to these physical connections.

An XA connection instance corresponds to a single Oracle session, although the session can be used in sequence by multiple logical connection instances, as with pooled connection instances.

You will typically get an XA connection instance from an XA data source instance in your middle-tier software. You can get multiple XA connection instances from a single XA data source instance if the distributed transaction will involve multiple sessions in the same database.

XA connections produce OracleXAResource instances and JDBC connection instances.

XA resources

These are used by a transaction manager in coordinating the transaction branches of a distributed transaction.

You will get one OracleXAResource instance from each XA connection instance, typically in your middle-tier software. There is a one-to-one correlation between OracleXAResource instances and XA connection instances. Equivalently, there is a one-to-one correlation between OracleXAResource instances and Oracle sessions.

In a typical scenario, the middle-tier component will hand off OracleXAResource instances to the transaction manager, for use in coordinating distributed transactions.

Each OracleXAResource instance corresponds to a single Oracle session. So, there can be only a single active transaction branch associated with an OracleXAResource instance at any given time. However, there can be additional suspended transaction branches.

Each OracleXAResource instance has the functionality to start, end, prepare, commit, or roll back the operations of the transaction branch running in the session with which the OracleXAResource instance is associated.

The prepare step is the first step of a two-phase commit operation. The transaction manager will issue a PREPARE to each OracleXAResource instance. Once the transaction manager sees that the operations of each transaction branch have prepared successfully, it will issue a COMMIT to each OracleXAResource instance to commit all the changes.

Transaction IDs

These are used to identify transaction branches. Each ID includes a transaction branch ID component and a distributed transaction ID component. This is how a branch is associated with a distributed transaction. All <code>OracleXAResource</code> instances associated with a given distributed transaction would have a transaction ID that includes the same distributed transaction ID component.

• OracleXAResource.ORATRANSLOOSE

Start a loosely coupled transaction with transaction ID ${\tt xid}.$



36.1.4 About Switching Between Global and Local Transactions

Applications can share connections between local and global transactions. Applications can also switch connections between local transactions and global transactions.

A connection is always in one of the following modes:

• NO_TXN

No transaction is actively using this connection.

LOCAL_TXN

A local transaction with auto-commit turned off or disabled is actively using this connection.

• GLOBAL TXN

A global transaction is actively using this connection.

Each connection switches automatically between these modes depending on the operations carried out on the connection. A connection is always in NO_TXN mode when it is instantiated.

Note:

The modes are maintained internally by the JDBC drivers in association with Oracle Database.

Table 36-1 describes the connection mode transition rules.

Current Mode	Switches to NO_TXN When	Switches to LOCAL_TXN When	Switches to GLOBAL_TXN When
NO_TXN	NA	Auto-commit mode is false and an Oracle data manipulation language (DML) statement is run.	The start method is called on an XAResource obtained from the XAconnection that provided the current connection.
LOCAL_TXN	 Any of the following happens: An Oracle data definition language (DDL) statement is run. commit is called. rollback is called, but without parameters. 	NA	The start method is called on an XAResource obtained from the XAconnection that provided the current connection. This feature is available starting from Oracle Database 12 <i>c</i> Release 1 (12.1.0.2).

Table 36-1 Connection Mode Transitions



Current Mode	Switches to NO_TXN	Switches to	Switches to
	When	LOCAL_TXN When	GLOBAL_TXN When
GLOBAL_TXN	Within a global transaction open on this connection, end is called on an XAResource obtained from the XAconnection that provided this connection.	NEVER	NA

Table 36-1	(Cont.) Connection Mode Transitions	
------------	-------------------------------------	--

If none of these rules is applicable, then the mode does not change.

Mode Restrictions on Operations

The current connection mode restricts which operations are valid within a transaction.

- In the LOCAL_TXN mode, applications must not call prepare, commit, rollback, forget, or end on an XAResource. Doing so causes an XAException to be thrown.
- In the GLOBAL_TXN mode, applications must not call commit, rollback, rollback(Savepoint), setAutoCommit(true), Or setSavepoint On a java.sql.Connection, and must not call OracleSetSavepoint Or oracleRollback On an oracle.jdbc.OracleConnection. Doing so causes a SQLException to be thrown.

Note:

This mode-restriction error checking is in addition to the standard error checking on the transaction and savepoint APIs.

36.1.5 Oracle XA Packages

Oracle supplies the following three packages that have classes to implement distributed transaction functionality according to the XA standard:

- oracle.jdbc.xa
- oracle.jdbc.xa.client
- oracle.jdbc.xa.server

Classes for XA data sources, XA connections, and XA resources are in both the client package and the server package. An abstract class for each is in the top-level package. The OracleXid and OracleXAException classes are in the top-level oracle.jdbc.xa package, because their functionality does not depend on where the code is running.

In middle-tier scenarios, you will import OracleXid, OracleXAException, and the oracle.jdbc.xa.client package.

If you intend your XA code to run in the target Oracle Database, however, you will import the oracle.jdbc.xa.server package instead of the client package.



If code that will run inside a target database must also access remote databases, then do not import either package. Instead, you must fully qualify the names of any classes that you use from the client package to access a remote database or from the server package to access the local database. Class names are duplicated between these packages.

36.2 XA Components

This section discusses the XA components, that is, the standard XA interfaces specified in the JDBC standard, and the Oracle classes that implement them. The following topics are covered:

- XADatasource Interface and Oracle Implementation
- XAConnection Interface and Oracle Implementation
- XAResource Interface and Oracle Implementation
- OracleXAResource Method Functionality and Input Parameters
- Xid Interface and Oracle Implementation

36.2.1 XADatasource Interface and Oracle Implementation

The javax.sql.XADataSource interface outlines standard functionality of XA data sources, which are factories for XA connections. The overloaded getXAConnection method returns an XA connection instance and optionally takes a user name and password as input:

```
public interface XADataSource
{
    XAConnection getXAConnection() throws SQLException;
    XAConnection getXAConnection(String user, String password)
    throws SQLException;
    ...
}
```

Oracle JDBC implements the XADataSource interface with the OracleXADataSource class, located both in the oracle.jdbc.xa.client package and the oracle.jdbc.xa.server package.

The OracleXADataSource classes also extend the OracleConnectionPoolDataSource class, which extends the OracleDataSource class, and therefore, include all the connection properties.

The getXAConnection methods of the OracleXADataSource class returns the Oracle implementation of XA connection instances, which are OracleXAConnection instances.

Note:

You can register XA data sources in Java Naming Directory and Interface (JNDI) using the same naming conventions as discussed previously for nonpooling data sources.



See Also:

For information about Fast Connection Failover, refer to Oracle Universal Connection Pool for JDBC Developer's Guide.

36.2.2 XAConnection Interface and Oracle Implementation

An XA connection instance, as with a pooled connection instance, encapsulates a physical connection to a database. This would be the database specified in the connection properties of the XA data source instance that produced the XA connection instance.

Each XA connection instance also has the facility to produce the OracleXAResource instance that will correspond to it for use in coordinating the distributed transaction.

An XA connection instance is an instance of a class that implements the standard javax.sql.XAConnection interface:

```
public interface XAConnection extends PooledConnection
{
    javax.jta.xa.XAResource getXAResource() throws SQLException;
}
```

As you see, the XAConnection interface extends the javax.sql.PooledConnection interface, so it also includes the getConnection, close, addConnectionEventListener, and removeConnectionEventListener methods.

Oracle JDBC implements the XAConnection interface with the OracleXAConnection class, located both in the oracle.jdbc.xa.client package and the oracle.jdbc.xa.server package.

The OracleXAConnection classes also extend the OraclePooledConnection class.

The OracleXAConnection class getXAResource method returns the Oracle implementation of an OracleXAResource instance, which is an OracleXAResource instance. The getConnection method returns an OracleConnection instance.

A JDBC connection instance returned by an XA connection instance acts as a temporary handle to the physical connection, as opposed to encapsulating the physical connection. The physical connection is encapsulated by the XA connection instance. The connection obtained from an XAConnection object behaves exactly like a regular connection, until it participates in a global transaction. At that time, auto-commit status is set to false. After the global transaction ends, auto-commit status is returned to the value it had before the global transaction. The default auto-commit status on a connection obtained from XAConnection is false in all releases prior to Oracle Database 10g. Starting from Oracle Database 10g, the default status is true.

Each time an XA connection instance getConnection method is called, it returns a new connection instance that exhibits the default behavior, and closes any previous connection instance that still exists and had been returned by the same XA connection instance. However, it is advisable to explicitly close any previous connection instance before opening a new one.

Calling the close method of an XA connection instance closes the physical connection to the database. This is typically performed in the middle tier.



36.2.3 XAResource Interface and Oracle Implementation

The transaction manager uses OracleXAResource instances to coordinate all the transaction branches that constitute a distributed transaction.

Each OracleXAResource instance provides the following key functionality, typically invoked by the transaction manager:

- It associates and disassociates distributed transactions with the transaction branch operating in the XA connection instance that produced this <code>OracleXAResource</code> instance. Essentially, it associates distributed transactions with the physical connection or session encapsulated by the XA connection instance. This is done through use of transaction IDs.
- It performs the two-phase commit functionality of a distributed transaction to ensure that changes are not committed in one transaction branch before there is assurance that the changes will succeed in all transaction branches.

Note:

- Because there must always be a one-to-one correlation between XA connection instances and OracleXAResource instances, an
 OracleXAResource instance is implicitly closed when the associated XA connection instance is closed.
- If a transaction is opened by a given OracleXAResource instance, then it must also be closed by the same OracleXAResource instance.

An OracleXAResource instance is an instance of a class that implements the standard javax.transaction.xa.XAResource interface. Oracle JDBC implements the XAResource interface with the OracleXAResource class, located both in the oracle.jdbc.xa.client package and the oracle.jdbc.xa.server package.

Oracle JDBC driver creates and returns an OracleXAResource instance whenever the getXAResource method of the OracleXAConnection class is called, and it is Oracle JDBC driver that associates an OracleXAResource instance with a connection instance and the transaction branch being run through that connection.

This method is how an OracleXAResource instance is associated with a particular connection and with the transaction branch being run in that connection.

36.2.4 OracleXAResource Method Functionality and Input Parameters

The OracleXAResource class has several methods to coordinate a transaction branch with the distributed transaction with which it is associated. This functionality usually involves two-phase commit operations.

A transaction manager, receiving OracleXAResource instances from a middle-tier component, such as an application server, typically invokes this functionality.

Each of these methods takes a transaction ID as input, in the form of an Xid instance, which includes a transaction branch ID component and a distributed transaction ID component. Every transaction branch has a unique transaction ID, but transaction

branches belonging to the same global transaction have the same global transaction component as part of their transaction IDs.

start

Starts work on behalf of a transaction branch, associating the transaction branch with a distributed transaction.

void start(Xid xid, int flags)

The flags parameter must be one or more of the following values:

• XAResource.TMNOFLAGS

Flags the start of a new transaction branch for subsequent operations in the session associated with this XA resource instance. This branch will have the transaction ID xid, which is an <code>OracleXid</code> instance created by the transaction manager. This will map the transaction branch to the appropriate distributed transaction.

• XAResource.TMJOIN

Joins subsequent operations in the session associated with this XA resource instance to the existing transaction branch specified by xid.

• XAResource.TMRESUME

Resumes the transaction branch specified by xid.

Note:

A transaction branch can be resumed only if it had been suspended earlier.

• OracleXAResource.TMPROMOTE

Promotes a local transaction to a global transaction

• OracleXAResource.ORATMSERIALIZABLE

Starts a serializable transaction with transaction ID xid.

• OracleXAResource.ORATMREADONLY

Starts a read-only transaction with transaction ID xid.

• OracleXAResource.ORATMREADWRITE

Starts a read/write transaction with transaction ID xid.

• OracleXAResource.ORATRANSLOOSE

Starts a loosely coupled transaction with transaction ID xid.

TMNOFLAGS, TMJOIN, TMRESUME, TMPROMOTE, ORATMSERIALIZABLE, ORATMREADONLY, and ORATMREADWRITE are defined as static members of the XAResource interface and OracleXAResource class. ORATMSERIALIZABLE, ORATMREADONLY, and ORATMREADWRITE are the isolation-mode flags. The default isolation behavior is READ COMMITTED.



Note:

- Instead of using the start method with TMRESUME, the transaction manager can cast to OracleXAResource and use the resume (Xid xid) method, an Oracle extension.
- If you use TMRESUME, then you must also use TMNOMIGRATE, as in start(xid, XAResource.TMRESUME | OracleXAResource.TMNOMIGRATE). This prevents the application from receiving the error ORA 1002: fetch out of sequence.
- If you use the isolation-mode flags incorrectly, then an exception with code XAER_INVAL is raised. Furthermore, you cannot use isolation-mode flags when resuming a global transaction, because you cannot set the isolation level of an existing transaction. If you try to use the isolation-mode flags when resuming a transaction, then an external Oracle exception with code ORA-24790 is raised.
- In order to avoid Error ORA 1002: fetch out of sequence, include the TMNOMIGRATE flag as part of the start method. For example:

start(xid, XAResource.TMSUSPEND | OracleXAResource.TMNOMIGRATE);

• All the flags defined in OracleXAResource are Oracle extensions. When writing a transaction manager that uses these flags, you should be mindful of this.

Note that to create an appropriate transaction ID in starting a transaction branch, the transaction manager must know to which distributed transaction the transaction branch belongs. The mechanics of this are handled between the middle tier and transaction manager.

end

Ends work on behalf of the transaction branch specified by xid, disassociating the transaction branch from its distributed transaction.

void end(Xid xid, int flags)

The flags parameter can have one of the following values:

• XAResource.TMSUCCESS

This is to indicate that this transaction branch is known to have succeeded.

• XAResource.TMFAIL

This is to indicate that this transaction branch is known to have failed.

• XAResource.TMSUSPEND

This is to suspend the transaction branch specified by xid. By suspending transaction branches, you can have multiple transaction branches in a single session. Only one can be active at any given time, however. Also, this tends to be more expensive in terms of resources than having two sessions.

TMSUCCESS, TMFAIL, and TMSUSPEND are defined as static members of the XAResource interface and OracleXAResource class.



Note:

- Instead of using the end method with TMSUSPEND, the transaction manager can cast to OracleXAResource and use the suspend(Xid xid) method, an Oracle extension.
- This XA functionality to suspend a transaction provides a way to switch between various transactions within a single JDBC connection. You can use the XA classes to accomplish this, even if you are not in a distributed transaction environment and would otherwise have no need for the XA classes.
- If you use TMSUSPEND, then you must also use TMNOMIGRATE, as in end(xid, XAResource.TMSUSPEND | OracleXAResource.TMNOMIGRATE). This prevents the application from receiving the error ORA 1002: fetch out of sequence.
- In order to avoid Error ORA 1002: fetch out of sequence, include the TMNOMIGRATE flag as part of the end method. For example:

end(xid, XAResource.TMSUSPEND | OracleXAResource.TMNOMIGRATE);

• All the flags defined in OracleXAResource are Oracle extensions. Any transaction manager that uses these flags should take heed of this.

prepare

Prepares the changes performed in the transaction branch specified by xid. This is the first phase of a two-phase commit operation, to ensure that the database is accessible and that the changes can be committed successfully.

int prepare(Xid xid)

This method returns an integer value as follows:

• XAResource.XA RDONLY

This is returned if the transaction branch runs only read-only operations such as SELECT statements.

• XAResource.XA OK

This is returned if the transaction branch runs updates that are all prepared without error.

XA_RDONLY and XA_OK are defined as static members of the XAResource interface and OracleXAResource class.



Note: The prepare method sometimes does not return any value if the transaction branch runs updates and any of them encounters errors during preparation. In this case, an XA exception is thrown. Always call the end method on a branch before calling the prepare method. If there is only one transaction branch in a distributed transaction, then there is no need to call the prepare method. You can call the OracleXAResource commit method without preparing first.

commit

Commits prepared changes in the transaction branch specified by xid. This is the second phase of a two-phase commit and is performed only after all transaction branches have been successfully prepared.

void commit(Xid xid, boolean onePhase)

Set the onePhase parameter as follows:

• true

This is to use one-phase instead of two-phase protocol in committing the transaction branch. This is appropriate if there is only one transaction branch in the distributed transaction; the prepare step would be skipped.

false

This is to use two-phase protocol in committing the transaction branch.

rollback

Rolls back prepared changes in the transaction branch specified by xid.

```
void rollback(Xid xid)
```

forget

Tells the resource manager to forget about a heuristically completed transaction branch.

public void forget (Xid xid)

recover

The transaction manager calls this method during recovery to obtain the list of transaction branches that are currently in prepared or heuristically completed states.

public Xid[] recover(int flag)



Note:

Values for flag other than TMSTARTRSCAN, TMENDRSCAN, or TMNOFLAGS, cause an exception to be thrown, otherwise flag is ignored.

The resource manager returns zero or more Xids for the transaction branches that are currently in a prepared or heuristically completed state. If an error occurs during the operation, then the resource manager throws the appropriate XAException.

Note:

The recover method requires SELECT privilege on DBA_PENDING_TRANSACTIONS and EXECUTE privilege on SYS.DBMS_XA in Oracle database server. For database versions prior to Oracle Database 11g Release 1, where an Oracle patch including a fix for bug 5945463 is not available, or it is infeasible to apply the patch for the particular application scenario, the recover method requires SYSBDBA privilege. Regular use of SYSDBA privilege is a security risk. So, Oracle strongly recommends that you upgrade your Database or apply the fix for bug 5945463, if you need to use the recover method.

isSameRM

To determine if two OracleXAResource instances correspond to the same resource manager, call the isSameRM method from one OracleXAResource instance, specifying the other OracleXAResource instance as input. In the following example, presume <code>xares1</code> and <code>xares2</code> are OracleXAResource instances:

boolean sameRM = xares1.isSameRM(xares2);

36.2.5 Xid Interface and Oracle Implementation

The transaction manager creates transaction ID instances and uses them in coordinating the branches of a distributed transaction. Each transaction branch is assigned a unique transaction ID, which includes the following information:

Format identifier

A format identifier specifies a Java transaction manager. For example, there could be a format identifier orcl. This field *cannot* be null. The size of a format identifier is 4 bytes.

Global transaction identifier

It is also known as a distributed transaction ID component. The size of a global transaction identifier is 64 bytes.

Branch qualifier

It is also known as transaction branch ID component. The size of a branch qualifier is 64 bytes.

The 64-byte global transaction identifier value will be identical in the transaction IDs of all transaction branches belonging to the same distributed transaction. However, the overall transaction ID is unique for every transaction branch.



An XA transaction ID instance is an instance of a class that implements the standard javax.transaction.xa.Xid interface, which is a Java mapping of the X/Open transaction identifier XID structure.

Oracle implements this interface with the OracleXid class in the oracle.jdbc.xa package. OracleXid instances are employed only in a transaction manager, transparent to application programs or an application server.

Note:

Oracle does not require the use of OracleXid for OracleXAResource calls. Instead, use any class that implements the javax.transaction.xa.Xid interface.

A transaction manager may use the following in creating an OracleXid instance:

public OracleXid(int fId, byte gId[], byte bId[]) throws XAException

fId is an integer value for the format identifier, gId[] is a byte array for the global transaction identifier, and bId[] is a byte array for the branch qualifier.

The Xid interface specifies the following getter methods:

- public int getFormatId()
- public byte[] getGlobalTransactionId()
- public type[] getBranchQualifier()

36.3 Error Handling and Optimizations

This section focuses on the functionality of XA exceptions and error handling and the Oracle optimizations in its XA implementation. It covers the following topics:

- XAException Classes and Methods
- Mapping Between Oracle Errors and XA Errors
- XA Error Handling
- Oracle XA Optimizations

The exception and error-handling discussion includes the standard XA exception class and the Oracle-specific XA exception class, as well as particular XA error codes and error-handling techniques.

36.3.1 XAException Classes and Methods

XA methods throw XA exceptions, as opposed to general exceptions or SQLExceptions. An XA exception is an instance of the standard class javax.transaction.xa.XAException or a subclass.

An Oracle XAException is an instance that consists of an Oracle error portion and an XA error portion. Oracle provides the oracle.jdbc.xa.OracleXAException subclasses



of the standard javax.transaction.xa.XAException class. An OracleXAException instance is constructed using one of the following constructors:

```
public OracleXAException()
```

public OracleXAException(int error)

The error value is an error code that combines an Oracle SQL error value and an XA error value. The JDBC driver determines exactly how to combine the Oracle and XA error values.

The OracleXAException class has the following methods:

• public int getOracleError()

This method returns the Oracle SQL error code pertaining to the exception, a standard ORA error number or 0 if there is no Oracle SQL error.

• public int getXAError()

This method returns the XA error code pertaining to the exception. XA error values are defined in the javax.transaction.xa.XAException class.

36.3.2 Mapping Between Oracle Errors and XA Errors

Oracle errors correspond to XA errors in OracleXAException instances as documented in Table 36-2.

Oracle Error Code	XA Error Code
ORA 24756	XAException.XAER_NOTA
ORA 24764	XAException.XA_HEURCOM
ORA 24765	XAException.XA_HEURRB
ORA 24766	XAException.XA_HEURMIX
ORA 24767	XAException.XA_RDONLY
ORA 25351	XAException.XA_RETRY
ORA 30006	XAException.XA_RETRY
ORA 24763	XAException.XAER_PROTO
ORA 24769	XAException.XAER_PROTO
ORA 24770	XAException.XAER_PROTO
ORA 24776	XAException.XAER_PROTO
ORA 2056	XAException.XAER_PROTO
ORA 17448	XAException.XAER_PROTO
ORA 24768	XAException.XAER_PROTO
ORA 24775	XAException.XAER_PROTO
ORA 24761	XAException.XA_RBROLLBACK
ORA 2091	XAException.XA_RBROLLBACK
ORA 2092	XAException.XA_RBROLLBACK
ORA 24780	XAException.XAER_RMERR

Table 36-2 Oracle-XA Error Mapping



Table 36-2	(Cont.) Oracle-XA	Error Mapping
------------	-------------------	---------------

Oracle Error Code	XA Error Code
All other ORA errors	XAException.XAER_RMFAIL

36.3.3 XA Error Handling

The following code snippet uses the OracleXAException class to process an XA exception:

```
try {
    ...
    ...Perform XA operations...
    ...
} catch(OracleXAException oxae) {
    int oraerr = oxae.getOracleError();
    System.out.println("Error " + oraerr);
}
    catch(XAException xae)
{...Process generic XA exception...}
```

In case the XA operations did not throw an Oracle-specific XA exception, the code drops through to process a generic XA exception.

36.3.4 Oracle XA Optimizations

Oracle JDBC has functionality to improve performance if two or more branches of a distributed transaction use the same database instance, meaning that the OracleXAResource instances associated with these branches are associated with the same resource manager.

In such a circumstance, the prepare method of only one of these <code>OracleXAResource</code> instances will return XA_OK or will fail. The rest will return XA_RDONLY, even if updates are made. This allows the transaction manager to implicitly join all the transaction branches and commit or roll back, in case of failure, the joined transaction through the <code>OracleXAResource</code> instance that returned XA_OK or failed.

The transaction manager can use the <code>OracleXAResource class isSameRM</code> method to determine if two <code>OracleXAResource</code> instances are using the same resource manager. This way it can interpret the meaning of <code>XA_RDONLY</code> return values.

36.4 About Implementing a Distributed Transaction

This section provides an example of how to implement a distributed transaction using Oracle XA functionality. This section covers the following topics:

- Summary of Imports for Oracle XA
- Oracle XA Code Sample

36.4.1 Summary of Imports for Oracle XA

You must import the following for Oracle XA functionality:



import oracle.jdbc.xa.OracleXid; import oracle.jdbc.xa.OracleXAException; import oracle.jdbc.pool.*; import oracle.jdbc.xa.client.*; import javax.transaction.xa.*;

The oracle.jdbc.pool package has classes for connection pooling functionality, some of which have XA-related classes as subclasses.

Alternatively, if the code will run inside Oracle Database and access that database for SQL operations, you must import oracle.jdbc.xa.server instead of oracle.jdbc.xa.client.

import oracle.jdbc.xa.server.*;

If your application must access another Oracle Database as part of an XA transaction using the server-side Thin driver, then your code can use the fully qualified names of the oracle.xa.client classes.

The client and server packages each have versions of the OracleXADataSource, OracleXAConnection, and OracleXAResource classes. Abstract versions of these three classes are in the top-level oracle.jdbc.xa package.

36.4.2 Oracle XA Code Sample

This example uses a two-phase distributed transaction with two transaction branches, each to a separate database.

Note that for simplicity, this example combines code that would typically be in a middle tier with code that would typically be in a transaction manager, such as the <code>OracleXAResource</code> method invocations and the creation of transaction IDs.

For brevity, the specifics of creating transaction IDs and performing SQL operations are not shown here. The complete example is shipped with the product.

This example performs the following sequence:

- **1**. Start transaction branch #1.
- 2. Start transaction branch #2.
- 3. Execute DML operations on branch #1.
- 4. Execute DML operations on branch #2.
- 5. End transaction branch #1.
- 6. End transaction branch #2.
- 7. Prepare branch #1.
- 8. Prepare branch #2.
- 9. Commit branch #1.
- **10.** Commit branch #2.

```
// You need to import the java.sql package to use JDBC
import java.sql.*;
import javax.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.*;
import oracle.jdbc.xa.OracleXid;
import oracle.jdbc.xa.OracleXAException;
```



```
import oracle.jdbc.xa.client.*;
import javax.transaction.xa.*;
class XA4
{
 public static void main (String args [])
      throws SQLException
  {
    try
    {
        String URL1 = "jdbc:oracle:oci:@";
        // You can put a database name after the @ sign in the connection URL.
        String URL2 ="jdbc:oracle:thin:@(description=(address=(host=localhost))
                     (protocol=tcp) (port=5521))
(connect data=(service name=orcl)))";
        // Create first DataSource and get connection
        OracleDataSource ods1 = new OracleDataSource();
        ods1.setURL(URL1);
        ods1.setUser("HR");
        ods1.setPassword("hr");
        Connection conna = ods1.getConnection();
        // Create second DataSource and get connection
        OracleDataSource ods2 = new OracleDataSource();
        ods2.setURL(URL2);
        ods2.setUser("HR");
        ods2.setPassword("hr");
        Connection connb = ods2.getConnection();
        // Prepare a statement to create the table
        Statement stmta = conna.createStatement ();
        // Prepare a statement to create the table
        Statement stmtb = connb.createStatement ();
        try
        {
          // Drop the test table
         stmta.execute ("drop table my table");
        }
        catch (SQLException e)
        {
          // Ignore an error here
        }
        try
        {
          // Create a test table
         stmta.execute ("create table my table (col1 int)");
        }
        catch (SQLException e)
        {
          //\ Ignore an error here too
        }
        try
        {
         // Drop the test table
          stmtb.execute ("drop table my tab");
        }
```



```
catch (SQLException e)
{
  // Ignore an error here
}
trv
{
  // Create a test table
 stmtb.execute ("create table my_tab (col1 char(30))");
}
catch (SQLException e)
  // Ignore an error here too
}
// Create XADataSource instances and set properties.
OracleXADataSource oxds1 = new OracleXADataSource();
oxds1.setURL("jdbc:oracle:oci:@");
oxds1.setUser("HR");
oxds1.setPassword("hr");
OracleXADataSource oxds2 = new OracleXADataSource();
oxds2.setURL("jdbc:oracle:thin:@(description=(address=(host=localhost))
           (protocol=tcp) (port=5521)) (connect data=(service name=orcl)))");
oxds2.setUser("HR");
oxds2.setPassword("hr");
//\ensuremath{\,{\rm Get}} XA connections to the underlying data sources
XAConnection pc1 = oxds1.getXAConnection();
XAConnection pc2 = oxds2.getXAConnection();
// Get the physical connections
Connection conn1 = pc1.getConnection();
Connection conn2 = pc2.getConnection();
// Get the XA resources
XAResource oxar1 = pc1.getXAResource();
XAResource oxar2 = pc2.getXAResource();
// Create the Xids With the Same Global Ids
Xid xid1 = createXid(1);
Xid xid2 = createXid(2);
// Start the Resources
oxar1.start (xid1, XAResource.TMNOFLAGS);
oxar2.start (xid2, XAResource.TMNOFLAGS);
// Execute SQL operations with conn1 and conn2
doSomeWork1 (conn1);
doSomeWork2 (conn2);
// END both the branches -- IMPORTANT
oxar1.end(xid1, XAResource.TMSUCCESS);
oxar2.end(xid2, XAResource.TMSUCCESS);
// Prepare the RMs
int prp1 = oxar1.prepare (xid1);
int prp2 = oxar2.prepare (xid2);
System.out.println("Return value of prepare 1 is " + prp1);
```

```
System.out.println("Return value of prepare 2 is " + prp2);
   boolean do commit = true;
   if (!((prp1 == XAResource.XA_OK) || (prp1 == XAResource.XA_RDONLY)))
      do commit = false;
   if (!((prp2 == XAResource.XA OK) || (prp2 == XAResource.XA RDONLY)))
      do commit = false;
   System.out.println("do commit is " + do commit);
   System.out.println("Is oxar1 same as oxar2 ? " + oxar1.isSameRM(oxar2));
   if (prp1 == XAResource.XA OK)
     if (do commit)
        oxar1.commit (xid1, false);
     else
        oxar1.rollback (xid1);
   if (prp2 == XAResource.XA OK)
     if (do commit)
        oxar2.commit (xid2, false);
     else
        oxar2.rollback (xid2);
    // Close connections
   conn1.close();
   conn1 = null;
   conn2.close();
   conn2 = null;
   pc1.close();
   pc1 = null;
   pc2.close();
   pc2 = null;
   ResultSet rset = stmta.executeQuery ("select col1 from my table");
   while (rset.next())
     System.out.println("Col1 is " + rset.getInt(1));
   rset.close();
   rset = null;
   rset = stmtb.executeQuery ("select coll from my tab");
   while (rset.next())
     System.out.println("Col1 is " + rset.getString(1));
   rset.close();
   rset = null;
   stmta.close();
   stmta = null;
   stmtb.close();
   stmtb = null;
   conna.close();
   conna = null;
   connb.close();
   connb = null;
} catch (SQLException sqe)
```



```
{
    sqe.printStackTrace();
  } catch (XAException xae)
    if (xae instanceof OracleXAException) {
      System.out.println("XA Error is " +
                    ((OracleXAException)xae).getXAError());
      System.out.println("SQL Error is " +
                    ((OracleXAException)xae).getOracleError());
    }
  }
}
static Xid createXid(int bids)
 throws XAException
{...Create transaction IDs...}
private static void doSomeWork1 (Connection conn)
throws SQLException
{...Execute SQL operations...}
private static void doSomeWork2 (Connection conn)
throws SQLException
{...Execute SQL operations...}
```

36.5 Native-XA in Oracle JDBC Drivers

In general, XA commands can be sent to the server in the following ways:

- Through non-native APIs
- Through native APIs

There is a huge performance difference between the two methods of sending XA commands to the server. The use of native APIs provide high performance gains as compared to the use of non-native APIs.

Prior to Oracle Database 10*g*, the Thin driver used non-native APIs to send XA commands to the server because Thin native APIs were not available. The non-native APIs use PL/SQL procedures, so they have the following disadvantages:

- They require different messages on the wire.
- They cause more round-trips to the database.
- They cause more cursors to remain open.
- They damage statement caching by occupying space in the Statement Cache.

Moreover, the implementation of non-native APIs is in the server. So, in order to solve any problem in sending XA commands, it requires a server patch. This creates a major issue because sometimes the patch requires restarting the server.

Starting from Oracle Database 10*g*, the Thin native APIs are available and are used to send XA commands, by default. Native APIs are more than 10 times faster than the non-native ones.

This section covers the following topics:

OCI Native XA

ORACLE

• Thin Native XA

36.5.1 OCI Native XA

Native XA is enabled through the use of the tnsEntry and nativeXA properties of the OracleXADataSource class.

Note:

Currently, OCI Native XA does not work in a multithreaded environment. The OCI driver uses the C/XA library of Oracle to support distributed transactions, which requires that an XAConnection be obtained for each thread before resuming a global transaction.

Configuration and Installation

On a Solaris or Linux system, you need the libheteroxall.so shared library to enable the Native XA feature. This library must be installed and available in the search path for the Native XA feature to work properly.

On a Microsoft Windows system, you need the heteroxal1.dll file to enable the Native XA feature. This file must be installed and available in the Windows DLL path for the Native XA feature to work properly.

Exception Handling

When using the Native XA feature in distributed transactions, it is recommended that the application simply check for XAException or SQLException, rather than OracleXAException or OracleSQLException.

Note:

The mapping from SQL error codes to standard XA error codes does not apply to the Native XA feature.

Native XA Code Example

The following portion of code shows how to enable the Native XA feature:

```
// Create a XADataSource instance
OracleXADataSource oxds = new OracleXADataSource();
oxds.setURL(url);
// Set the nativeXA property to use Native XA feature
oxds.setNativeXA(true);
// Set the tnsEntry property to an older DB as required
oxds.setTNSEntryName("ora805");
...
```



Related Topics

- Features and Properties of Data Sources
- Native XA Messages

36.5.2 Thin Native XA

Like the JDBC OCI driver, the JDBC Thin driver also provides support for Native XA. However, the JDBC Thin driver provides support for Native XA by default. This is unlike the case of the JDBC OCI driver in which the support for Native XA is not enabled by default.

You can disable Native XA by calling setNativeXA(false) on the XA data source as follows:

```
// Create a XADataSource instance
OracleXADataSource oxds = new OracleXADataSource();
// Disabling Native XA
oxds.setNativeXA(false);
...
```

For example, you may need to disable Native XA as a workaround for a bug in the Native XA code.



Part VIII Manageability

This part discusses the database management and diagnosability support in Oracle Java Database Connectivity (JDBC) drivers.

Part VIII contains the following chapters:

- Database Administration
- Diagnosability in JDBC
- JDBC DMS Metrics



37 Database Administration

This chapter discusses the database administration methods introduced in Oracle Database 11*g* Release 1. This chapter contains the following sections:

- Using the Database Administration Methods
- Using the startup Method
- Using the shutdown Method
- A Complete Example

37.1 Using the Database Administration Methods

Starting from Oracle Database 11g Release 1, two JDBC methods, startup and shutdown, has been added in the oracle.jdbc.OracleConnection interface, which enable you to start up and shut down an Oracle Database instance. This is similar to the way you would start up or shut down a database instance from SQL*Plus.

To use the startup and shutdown methods, you must:

- Have a dedicated connection to the server. You cannot be connected to a shared server through a dispatcher.
- Be connected as SYSDBA or SYSOPER. To connect as SYSDBA or SYSOPER with Oracle JDBC drivers, you need to set the INTERNAL LOGON connection property accordingly.

To log on as SYSDBA with the JDBC Thin driver you must configure the server to use the password file. For example, to configure system/manager to connect as SYSDBA with the JDBC Thin driver, perform the following:

1. From the command line, type:

```
orapwd file=$ORACLE_HOME/dbs/orapw entries=5
Enter password: password
```

2. Connect to database as SYSDBA and run the following commands from SQL*Plus:

```
GRANT SYSDBA TO system;

PASSWORD system

Changing password for system

New password: password

Retype new password: password
```

3. Edit init.ora and add the following line:

REMOTE LOGIN PASSWORDFILE=EXCLUSIVE

4. Restart the database instance.

As opposed to the JDBC Thin driver, the JDBC OCI driver can connect as SYSDBA or SYSOPER locally without specifying a password file on the server.



37.2 Using the startup Method

To start a database instance using the startup method, the application must first connect to the database as a SYSDBA or SYSOPER in the PRELIM_AUTH mode, which is the only connection mode that is permitted when the database is down. You can do this by setting the connection property PRELIM_AUTH to true. In the PRELIM_AUTH mode, you can *only* start up a database instance that is down. You *cannot* run any SQL statements in this mode.

Example

The following code snippet shows how to start up a database instance that is down:

```
OracleDataSource ds = new OracleDataSource();
    Properties prop = new Properties();
    prop.setProperty("user","sys");
    prop.setProperty("password","manager");
    prop.setProperty("internal_logon","sysdba");
    prop.setProperty("prelim_auth","true");
    ds.setConnectionProperties(prop);
    ds.setURL("jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp))
(HOST=XYZ.com)(PORT=5221))"
+ "(CONNECT_DATA=(SERVICE_NAME=rdbms.devplmt.XYZ.com)))");
    OracleConnection conn = (OracleConnection)ds.getConnection();
    conn.startup(OracleConnection.DatabaseStartupMode.NO_RESTRICTION);
    conn.close();
```

Note:

The startup method will start up the database using the server parameter file. Oracle JDBC drivers do *not* support database startup using the client parameter file.

37.2.1 Database Startup Options

The startup method takes a parameter that specifies the database startup option.

The following table lists the supported database startup options. These options are defined in the oracle.jdbc.OracleConnection.DatabaseStartupMode class.

Option Description	
FORCE	Shuts down the current instance of the database, if any, in the ABORT mode, before starting a new instance.
NO_RESTRICTION	Starts up the database with no restrictions.
RESTRICT	Starts up the database and allows database access only to users with both the CREATE SESSION and RESTRICTED SESSION privileges, typically, the DBA.

Table 37-1 Supported Database Startup Options



The startup method only starts up a database instance. It neither mounts it nor opens it. For mounting and opening the database instance, you have to reconnect as SYSDBA or SYSOPER, without the PRELIM AUTH mode.

Example

The following code snippet shows how to mount and open a database instance:

```
OracleDataSource ds1 = new OracleDataSource();
Properties prop1 = new Properties();
prop1.setProperty("user","sys");
prop1.setProperty("password","manager");
prop1.setProperty("internal_logon","sysdba");
ds1.setConnectionProperties(prop1);
ds1.setURL(DB_URL);
OracleConnection conn1 = (OracleConnection)ds1.getConnection();
Statement stmt = conn1.createStatement();
stmt.executeUpdate("ALTER DATABASE MOUNT");
stmt.executeUpdate("ALTER DATABASE OPEN");
```

37.3 Using the shutdown Method

The shutdown method enables you to shut down an Oracle Database instance. To use this method, you must be connected to the database as a SYSDBA or SYSOPER.

Example

The following code snippet shows how to shut down a database instance:

```
OracleDataSource ds2 = new OracleDataSource();
...
OracleConnection conn2 = (OracleConnection)ds2.getConnection();
conn2.shutdown(OracleConnection.DatabaseShutdownMode.IMMEDIATE);
Statement stmt1 = conn2.createStatement();
stmt1.executeUpdate("ALTER DATABASE CLOSE NORMAL");
stmt1.executeUpdate("ALTER DATABASE DISMOUNT");
stmt1.close();
conn2.shutdown(OracleConnection.DatabaseShutdownMode.FINAL);
conn2.close();
```

37.3.1 Database Shutdown Options

Like the startup method, the shutdown method also takes a parameter. In this case, the parameter specifies the database shutdown option. Table 37-2 lists the supported database shutdown options. These options are defined in the oracle.jdbc.OracleConnection.DatabaseShutdownMode class.

OptionDescriptionABORTDoes not wait for current calls to complete or users to disconnect from the database.CONNECTRefuses any new connection and waits for existing connection to end.FINALShuts down the database.IMMEDIATEDoes not wait for current calls to complete or users to disconnect from the database.

Table 37-2 Supported Database Shutdown Options



Table 37-2 (Cont.) Su	upported Database	Shutdown Options
-----------------------	-------------------	------------------

Option	Description
TRANSACTIONAL	Refuses new transactions and waits for active transactions to end.
TRANSACTIONAL_LOCAL	Refuses new local transactions and waits for active local transactions to end.

For shutdown options other than ABORT and FINAL, you must call the shutdown method again with the FINAL option to actually shut down the database.

Note:

The shutdown (DatabaseShutdownMode.FINAL) method must be preceded by another call to the shutdown method with one of the following options: CONNECT, TRANSACTIONAL, TRANSACTIONAL_LOCAL, or IMMEDIATE. Otherwise, the call hangs.

37.3.2 Standard Database Shutdown Process

A standard way to shut down the database is as follows:

- 1. Initiate shutdown by prohibiting further connections or transactions in the database. The shut down option can be either CONNECT, TRANSACTIONAL, TRANSACTIONAL LOCAL, OR IMMEDIATE.
- 2. Dismount and close the database by calling the appropriate ALTER DATABASE command.
- 3. Finish shutdown using the FINAL option.

In special circumstances to shut down the database as fast as possible, the ABORT option can be used. This is the equivalent to SHUTDOWN ABORT in SQL*Plus.

37.4 A Complete Example

Example 37-1 illustrates the use of the startup and shutdown methods.

Example 37-1 Database Startup and Shutdown

```
import java.sql.Statement;
import java.util.Properties;
import oracle.jdbc.OracleConnection;
import oracle.jdbc.pool.OracleDataSource;
/**
 * To logon as sysdba, you need to create a password file for user "sys":
 * orapwd file=/path/orapw password=password entries=300
 * and add the following setting in init.ora:
 * REMOTE_LOGIN_PASSWORDFILE=EXCLUSIVE
 * then restart the database.
 */
public class DBStartup
{
 static final String DB URL =
```



```
"jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=localhost)(PORT=5221))"
+ "(CONNECT DATA=(SERVICE NAME=rdbms.devplmt.XYZ.com)))";
 public static void main(String[] argv) throws Exception
// Starting up the database:
   OracleDataSource ds = new OracleDataSource();
   Properties prop = new Properties();
   prop.setProperty("user", "sys");
   prop.setProperty("password", "manager");
   prop.setProperty("internal_logon", "sysdba");
   prop.setProperty("prelim auth","true");
   ds.setConnectionProperties(prop);
    ds.setURL(DB URL);
   OracleConnection conn = (OracleConnection)ds.getConnection();
    conn.startup(OracleConnection.DatabaseStartupMode.NO RESTRICTION);
    conn.close();
// Mounting and opening the database
   OracleDataSource ds1 = new OracleDataSource();
    Properties prop1 = new Properties();
   prop1.setProperty("user","sys");
   prop1.setProperty("password", "manager");
   prop1.setProperty("internal logon","sysdba");
   ds1.setConnectionProperties(prop1);
   ds1.setURL(DB URL);
   OracleConnection conn1 = (OracleConnection)ds1.getConnection();
   Statement stmt = conn1.createStatement();
   stmt.executeUpdate("ALTER DATABASE MOUNT");
   stmt.executeUpdate("ALTER DATABASE OPEN");
   stmt.close();
   conn1.close();
// Shutting down the database
   OracleDataSource ds2 = new OracleDataSource();
    Properties prop = new Properties();
   prop.setProperty("user", "sys");
   prop.setProperty("password", "manager");
   prop.setProperty("internal logon","sysdba");
   ds2.setConnectionProperties(prop);
   ds2.setURL(DB URL);
   OracleConnection conn2 = (OracleConnection) ds2.getConnection();
   conn2.shutdown(OracleConnection.DatabaseShutdownMode.IMMEDIATE);
   Statement stmt1 = conn2.createStatement();
   stmt1.executeUpdate("ALTER DATABASE CLOSE NORMAL");
   stmt1.executeUpdate("ALTER DATABASE DISMOUNT");
    stmt1.close();
    conn2.shutdown(OracleConnection.DatabaseShutdownMode.FINAL);
    conn2.close();
 }
}
```

38 Diagnosability in JDBC

The JDBC diagnosability feature is enhanced in this release to make it more user-friendly.

The enhancement also facilitates the use of the JDBC drivers in the Oracle Cloud Infrastructure (OCI) environment because unlike the On-Premises users, the Cloud users do not have absolute access to the system. The enhanced diagnosability feature provides diagnostic capabilities that are accessible and configurable in the Cloud environment.

38.1 Overview of JDBC Diagnosability

JDBC diagnosability feature is a client-only feature. It does not work with the Server-side Thin driver or the Server-side internal driver.

In the previous releases, the debug JAR files are used for logging purposes. These files are indicated with a _g in the file name, for example, ojdbc6_g.jar or ojdbc7_g.jar, and must be included in the CLASSPATH. The enhanced diagnosability feature eliminates the need to use the debug JAR files, and also the need to switch between the regular JAR files and the debug JAR files. Now, the regular JAR files include logging capabilities, which can be turned on with the following property:

-Doracle.jdbc.diagnostic.enableLogging=true

The logging configuration file can be configured in the following way:

-Djava.util.logging.config.file=./logging.config

Features of JDBC Diagnosability

The improved JDBC diagnosability feature is built upon the following features of Oracle Database:

- **Observability**: It enables Java developers to access summary information about the execution and performance of JDBC.
- **Diagnosability**: It records critical execution state in the event of a failure. Ideally, such a state is sufficient to diagnose the first occurrence of a failure and come up with a resolution of the problem causing the failure. You must balance the amount of state information recorded against the cost of recording such a state.
- **Execution Trace**: It records the execution sequence details. As execution trace has significant cost involved, you must enable it only in limited contexts.

Each diagnosability feature has two modes, public and sensitive. In the public mode, these features do not record or persist sensitive information. This severely limits the amount of information captured, and reduces the effectiveness of the diagnosability features. In the sensitive mode, these features record and persist sensitive information. You must be a privileged user for permitting the sensitive mode.



Sensitive mode is controlled by two switches, one for enabling and one for permitting. Sensitive mode for each of the diagnosability feature is permitted by setting the appropriate Java command-line switch. If the switch is not set on the Java command line, then sensitive mode cannot be permitted. After permitting the sensitive mode, you must enable it, as shown in the following example:

```
-Doracle.jdbc.diagnostic.permitSensitiveDiagnostics=true
-Doracle.jdbc.diagnostic.enableSensitiveDiagnostics=true
```

Note:

Permitting the sensitive mode does not automatically enable it. Attempting to enable sensitive mode, when not permitted, causes a fatal error.

Examples

The following code snippets show how to configure the logging configuration file:

Example 38-1 Configuring the Logging File

```
handlers = java.util.logging.ConsoleHandler
oracle.jdbc.level = FINEST
java.util.logging.ConsoleHandler.level = FINEST
java.util.logging.ConsoleHandler.formatter =
oracle.jdbc.diagnostics.OracleSimpleFormatter
```

Example 38-2 Configuring the Logging File (detailed)

```
handlers=java.util.logging.FileHandler
java.util.logging.FileHandler.level = FINEST
java.util.logging.FileHandler.formatter =
oracle.jdbc.diagnostics.OracleSimpleFormatter
java.util.logging.FileHandler.pattern = client.log
java.util.logging.FileHandler.limit = 1000000000
java.util.logging.FileHandler.count = 1000
oracle.ucp.level=FINEST
oracle.jdbc.level=FINEST
```

38.1.1 The Diagnose First Failure Feature

One of the major features of the enhanced diagnosability is the Diagnose First Failure feature.

When you enable this feature, it diagnoses the first occurrence of a failure, and the most critical trace information is captured in memory. As that memory fills, the oldest trace records are overwritten, and are subsequently dumped into java.util.logging, when signaled. One of the following can signal the dumping of the trace records:

- Occurrence of an error in the connection
- Client application code calling an API
- An MBean



This feature captures only the most critical information, that is, the extent of information that provides a reasonable chance of diagnosing the most likely problems. In the public mode, this information is severely restricted, so that any sensitive information is omitted. In the sensitive mode, this information includes the entire network conversation.

This feature is enabled by default. You can disable it either by setting the CONNECTION_PROPERTY_ENABLE_DIAGNOSE_FIRST_FAILURE property to FALSE or through the DiagnosticMBeans interface.

See Also:

Oracle Database JDBC Java API Reference



39 JDBC DMS Metrics

DMS metrics are used to measure the performance of application components.

This chapter discusses the following topics:

- Overview of JDBC DMS Metrics
- About Determining the Type of Metric to Be Generated
- About Generating the SQLText Metric
- About Accessing DMS Metrics Using JMX

Note:

There is another kind of metrics called end-to-end metrics. End-to-end metrics are used for tagging application activity from the entry into the application code through JDBC to the database and back.

JDBC supports the following end-to-end metrics:

- Action
- ClientId
- ExecutionContextId
- Module
- State

For earlier releases, to work with the preceding metrics, you could use the setEndToEndMetrics and getEndToEndMetrics methods of the
oracle.jdbc.OracleConnection interface. However, starting from Oracle Database
12c Release 1 (12.1), these methods have been deprecated. Oracle recommends
to use the setClientInfo and getClientInfo methods instead of the
setEndToEndMetrics and getEndToEndMetrics methods.

In Oracle Database 10*g*, Oracle Java Database Connectivity (JDBC) supports endto-end metrics. In Oracle Database 12*c* Release 1 (12.1), an application can set the end-to-end metrics directly only when it does not use a DMS-enabled JAR files. But, if the application uses a DMS-enabled JAR file, the end-to-end metrics can be set only through DMS.

🔶 WARNING:

Oracle strongly recommends using DMS metrics, if the application uses a DMSenabled JAR file.



See Also:

Oracle Database JDBC Java API Reference for more information about endto-end metrics

39.1 Overview of JDBC DMS Metrics

DMS metrics enable application and system developers to measure and export customized performance metrics for specific software components.

All DMS metrics are available in the following DMS-enabled JAR files:

- ojdbc8dms.jar
- ojdbc8dms_g.jar
- ojdbc11dms.jar
- ojdbc11dms_g.jar

Any other JDBC JAR files do not generate any DMS metrics. The metrics generated in Oracle Database 23c are different from the metrics generated for earlier versions of Oracle JDBC as it makes no attempt to retain compatibility with earlier versions. There are also no compatibility modes. A system that is dependent on the exact details of the DMS metrics generated by earlier versions of JDBC may have unexpected behavior when processing the metrics generated by Oracle JDBC 23c. This is by design and cannot be changed.

Statement metrics can be reported consolidated for all statements in a connection or individually for each statement. All DMS metrics, except those related to individual statements, are enabled at all times.

Note:

You can enable or disable the SQLText statement metric. It is disabled by default. If enabled, it is enabled for all statements.

39.2 About Determining the Type of Metric to Be Generated

To determine whether to use consolidated or individual metrics, JDBC checks the DMSConsole sensor weight. If the sensor weight is less than or equal to DMSConsole.NORMAL, then JDBC generates consolidated statement metrics. If the sensor weight is greater than DMSConsole.NORMAL, then JDBC generates individual statement metrics.

JDBC checks the DMSConsole sensor weight when creating a Prepared or Callable statement and depending on the sensor weight at the time the statement is created, the metrics are generated. Changing the value of the sensor weight, after the statement has been created, does not cause a statement to switch between consolidated and individual metrics.



Note:

In the presence of Statement caching, it may appear that changing sensor weight has no impact as statements are retrieved from the cache rather than created anew.

There is only one list of statement metrics that is generated for both consolidated and individual statement metrics. The only difference between these two lists is the aggregation of the statements. When individual statement metrics are generated, one set of metrics is generated for each distinct statement object created by the JDBC driver. On the other hand, when consolidated statement metrics are generated, all statements created by a given connection use the same set of statement metrics.

For example, consider an 'execute' phase event. If individual statement metrics are used, then each statement created will have a distinct 'execute' phase event. So, from two such statements, it will be possible to distinguish the execution statistics for the two separate statements. If one has an execution time of 1 second and the other an execution time of 3 seconds, then there will be two distinct 'execute' phase events, one with a total time and average of 1 second and the other with a total time and average of 3 seconds. But, if consolidated statement metrics are used, all statements will use the single 'execute' phase event common to the connection. So, from two such statements created by the same connection, it will not be possible to distinguish the execution statistics for the two statements. If one has an execution time of 1 second and the other an execution time of 3 seconds, then the common 'execute' phase event will report a total execution time of 4 seconds and an average of 2 seconds.

39.3 About Generating the SQLText Metric

Depending on the version of DMS, there are two mechanisms for determining the generating of the SQLText statement metrics:

- If the 12c version of the DMS JAR file is present in the classpath environment variable, then JDBC checks the DMS update SQL text flag. If this flag is true, then the SQLText metric is updated.
- If the 12c version of the DMS JAR file is not present in the classpath environment variable, then JDBC uses the value of the DMSStatementMetrics connection property. If this statement property is true, then SQLText metric is updated. The default value of this connection property is false.

Whether or not the SQLText metric will be generated is independent of the use of the type of statement metrics used, that is, individual statement metrics or consolidated statement metrics.

39.4 About Accessing DMS Metrics Using JMX

JMX (Java Management Extensions) is a Java technology that supplies tools for managing and monitoring applications, system objects, devices, service-oriented networks, and the JVM (Java Virtual Machine). You can easily access DMS metrics at run time using a management application that supports JMX. For more information about using JMX to access DMS data, go to the following URL http://www.oracle.com/technetwork/middleware/ toplink/overview/index.html



💉 See Also:

Oracle Database Java Developer's Guide for more information about JMX

Part IX Appendixes

This part consists of appendixes that discuss Java Database Connectivity (JDBC) reference information, tips for coding JDBC applications, JDBC error messages, and troubleshooting JDBC applications.

Part IX contains the following appendixes:

- JDBC Reference Information
- Oracle RAC Fast Application Notification
- JDBC Coding Tips
- JDBC Error Messages
- Troubleshooting

A JDBC Reference Information

This appendix contains detailed Java Database Connectivity (JDBC) reference information, including the following topics:

- Supported SQL-JDBC Data Type Mappings
- Supported SQL and PL/SQL Data Types
- About Using PL/SQL Types
- Using Embedded JDBC Escape Syntax
- Oracle JDBC Notes and Limitations

A.1 Supported SQL-JDBC Data Type Mappings

This section lists all the possible Java types to which a given SQL data type can have a valid mapping.

Oracle JDBC drivers will support these nondefault mappings. For example, to materialize SQL CHAR data in an oracle.sql.CHAR object, use the getCHAR method. To materialize it as a java.math.BigDecimal object, use the getBigDecimal method.

Note: The classes, where oracle.jdbc.OracleData appears in italic, can be generated by Oracle JVM Web Services Call-Out Utility.

SQL data type	Java types
BOOLEAN	boolean,java.lang.Boolean,oracle.sql.BOOLEAN
CHAR, VARCHAR2, LONG	java.lang.String
	oracle.sql.CHAR

Table A-1 Valid SQL Data Type-Java Class Mappings



SQL data type	Java types
NUMBER	boolean
	char
	byte
	short
	int
	long
	float
	double
	java.lang.Byte
	java.lang.Short
	java.lang.Integer java.lang.Long
	java.lang.Float
	java.lang.Double
	java.math.BigDecimal
	oracle.sql.NUMBER
BINARY INTEGER	boolean
_	char
	byte
	short
	int
	long
BINARY_FLOAT	oracle.sql.BINARY_FLOAT
BINARY_DOUBLE	oracle.sql.BINARY_DOUBLE
DATE	oracle.sql.DATE
RAW	oracle.sql.RAW
BLOB	oracle.jdbc.OracleBlob ¹
CLOB	oracle.jdbc.OracleClob ²
BFILE	oracle.sql.BFILE
ROWID	oracle.sql.ROWID
TIMESTAMP	oracle.sql.TIMESTAMP
TIMESTAMP WITH TIME ZONE	oracle.sql.TIMESTAMPTZ
TIMESTAMP WITH LOCAL TIME ZONE	oracle.sql.TIMESTAMPLTZ
ref cursor	java.sql.ResultSet
	sqlj.runtime.ResultSetIterator
user defined named types, ADTs	oracle.jdbc.OracleStruct ³
opaque named types	oracle.jdbc.OracleOpaque ⁴
nested tables and VARRAY named types	oracle.jdbc.OracleArray ⁵

Table A-1 (Cont.) Valid SQL Data Type-Java Class Mappings



S	SQL data type	Java types	
r	eferences to named types	oracle.jdbc.OracleRef ⁶	
1		se 1 (12.1), the oracle.sql.BLOB class is deprecated and replaced	
2	with the oracle.jdbc.OracleBlob i Starting from Oracle Database 12c Release	<pre>nterface. se 1 (12.1), the oracle.sql.CLOB class is deprecated and replaced</pre>	
	with the oracle.jdbc.OracleClob i	nterface.	
3	Starting from Oracle Database 12c Release replaced with the oracle.jdbc.Oracle	se 1 (12.1), the oracle.sql.STRUCT class is deprecated and	
4			
5	replaced with the oracle.jdbc.OracleOpaque interface. 5 Starting from Oracle Database 12c Release 1 (12.1), the oracle.sql.ARRAY class is deprecated and replaced		
	with the oracle.jdbc.OracleArray interface.		
6	Starting from Oracle Database 12c Release with the oracle.jdbc.OracleRef int	se 1 (12.1), the oracle.sql.REF class is deprecated and replaced terface.	
	A Note		
	Note:		
	• The type UROWID is not supported.		
	_	class is abstract. The value passed to a parameter of must be of the Java type corresponding to the	

Table A-1 (Cont.) Valid SQL Data Type-Java Class Mappings

type oracle.sql.Datum must be of the Java type corresponding to the underlying SQL type. Likewise, the value returned by a method with return type oracle.sql.Datum must be of the Java type corresponding to the underlying SQL type.

A.2 Supported SQL and PL/SQL Data Types

The tables in this section list SQL and PL/SQL data types, and Oracle JDBC driver support for them. The following table describes Oracle JDBC driver support for SQL data types.

SQL Data Type	Supported by JDBC Drivers?
BFILE	yes
BLOB	yes
CHAR	yes
CLOB	yes
DATE	yes
NCHAR	no ¹
NCHAR VARYING	no
NUMBER	yes
NVARCHAR2	yes ²
RAW	yes

Table A-2Support for SQL Data Types



SQL Data Type	Supported by JDBC Drivers?
REF	yes
ROWID	yes
UROWID	no
VARCHAR2	yes

 Table A-2
 (Cont.) Support for SQL Data Types

¹ The NCHAR type is supported indirectly. There is no corresponding java.sql.Types type, but if your application calls the formOfUse(NCHAR) method, then this type can be accessed.

² In JSE 6, the NVARCHAR2 type is supported directly. In J2SE 5.0, the NVARCHAR2 type is supported indirectly. There is no corresponding java.sql.Types type, but if your application calls the formOfUse(NCHAR) method, then this type can be accessed.

The following table describes Oracle JDBC support for the ANSI-supported SQL data types.

ANSI-Supported SQL Data Type	Supported by JDBC Drivers?
CHARACTER	yes
DEC	yes
DECIMAL	yes
DOUBLE PRECISION	yes
FLOAT	yes
INT	yes
INTEGER	yes
NATIONAL CHARACTER	no
NATIONAL CHARACTER VARYING	no
NATIONAL CHAR	yes
NATIONAL CHAR VARYING	no
NCHAR	yes
NCHAR VARYING	no
NUMERIC	yes
REAL	yes
SMALLINT	yes
VARCHAR	yes

Table A-3 Support for ANSI-92 SQL Data Types

The following table describes Oracle JDBC driver support for SQL User-Defined types.

Table A-4 Support for SQL User-Defined Types

SQL User-Defined type	Supported by JDBC Drivers?
OPAQUE	yes
Reference types	yes



SQL User-Defined type	Supported by JDBC Drivers?
Object types (JAVA_OBJECT)	yes
Nested table types and VARRAY types	yes

Table A-4 (Cont.) Support for SQL User-Defined Types

The following table describes Oracle JDBC driver support for PL/SQL data types. The PL/SQL data types include the following categories:

- Scalar types
- Scalar character types, which includes DATE data type
- Composite types
- Reference types
- Large object (LOB) types

Table A-5 Support for PL/SQL Data Types

Scalar Types:BINARY INTEGERyesDECyesDECIMALyesDOUBLE PRECISIONyesFLOATyesINTyesINTEGERyesNATURALyesNATURALyesNUMBERyesNUMERICyesPLS_INTEGERyesPOSITIVEyesPOSITIVEno
DECyesDECIMALyesDOUBLE PRECISIONyesFLOATyesINTyesINTyesINTEGERyesNATURALyesNATURALnnoNUMBERyesNUMERICyesPLS_INTEGERyesPOSITIVEyesPOSITIVEnno
DECIMALyesDOUBLE PRECISIONyesFLOATyesINTyesINTyesINTEGERyesNATURALyesNATURALnnoNUMBERyesNUMERICyesPLS_INTEGERyesPOSITIVEyesPOSITIVEnno
DOUBLE PRECISIONyesFLOATyesINTyesINTyesINTEGERyesNATURALyesNUMBERnoNUMERICyesPLS_INTEGERyesPOSITIVEyesPOSITIVEnno
FLOATyesINTyesINTEGERyesNATURALyesNATURALnnoNUMBERyesNUMERICyesPLS_INTEGERyesPOSITIVEyesPOSITIVEnno
INT yes INTEGER yes NATURAL yes NATURAL <i>n</i> no NUMBER yes NUMERIC yes PLS_INTEGER yes POSITIVE no no
INTEGER yes NATURAL yes NATURAL <i>n</i> no NUMBER yes NUMERIC yes PLS_INTEGER yes POSITIVE yes
NATURALyesNATURALnnoNUMBERyesNUMERICyesPLS_INTEGERyesPOSITIVEyesPOSITIVEnno
NATURALnnoNUMBERyesNUMERICyesPLS_INTEGERyesPOSITIVEyesPOSITIVEnno
NUMBERyesNUMERICyesPLS_INTEGERyesPOSITIVEyesPOSITIVEnno
NUMERICyesPLS_INTEGERyesPOSITIVEyesPOSITIVEnno
PLS_INTEGER yes POSITIVE yes POSITIVEn no
POSITIVE yes POSITIVEn no
POSITIVE <i>n</i> no
REAL yes
SIGNTYPE yes
SMALLINT yes
BOOLEAN yes
Scalar Character Types:
CHAR yes
CHARACTER yes
LONG yes
LONG RAW yes



PL/SQL Data Type	Supported by JDBC Drivers?
NCHAR	no (see Note)
NVARCHAR2	no (see Note)
RAW	yes
ROWID	yes
STRING	yes
UROWID	no
VARCHAR	yes
VARCHAR2	yes
DATE	yes
Composite Types:	
RECORD	no
TABLE	no
VARRAY	yes
Reference Types:	
REF CURSOR types	yes
object reference types	yes
LOB Types:	
BFILE	yes
BLOB	yes
CLOB	yes
NCLOB	yes

Table A-5 (Cont.) Support for PL/SQL Data Types

Note:

- The types NATURAL, NATURAL*n*, POSITIVE, POSITIVE*n*, and SIGNTYPE are subtypes of BINARY INTEGER.
- The types DEC, DECIMAL, DOUBLE PRECISION, FLOAT, INT, INTEGER, NUMERIC, REAL, and SMALLINT are subtypes of NUMBER.
- The types NCHAR and NVARCHAR2 are supported indirectly. There is no corresponding java.sql.Types type, but if your application calls formOfUse (NCHAR), then these types can be accessed.

Related Topics

• NCHAR, NVARCHAR2, NCLOB and the defaultNChar Property

A.3 About Using PL/SQL Types

Starting from Oracle Database 12c Release 1 (12.1), you can map schema-level PL/SQL types as generic java.sql.Struct type and PL/SQL collections as java.sql.Array types. So, instead of creating schema-level types that are mapped to PL/SQL package types for binding, you can describe and bind PL/SQL types using only the JDBC APIs.

For example, you can call the <code>Connection.createStruct(type_name)</code> method to first create a descriptor that can be used to describe a PL/SQL type and then to create a new <code>STRUCT</code> representation of this type on the client. In Oracle Database 12c Release 1 (12.1), you can reuse this API by specifying <code>type_name</code> as "schema.package.typename" or "package.typename".

All PL/SQL package types are mapped to a system-wide unique name that can be used by JDBC to retrieve the server-side type metadata. The name is in the following form:

[SCHEMA.] <PACKAGE>.<TYPE>

Note:

If the schema is the same as the package name, and if there is a type with the same name as the PL/SQL type, then it will not be able to identify an object with the two part name format, that is, <package>.<type>. In such cases, you must use three part names <schema>.<package>.<type>.

The following code snippet explains how to bind types declared in PL/SQL packages:

```
/*
------
# Perform the following SQL operations prior to running this sample
------
conn HR/<password>;
create or replace package TEST PKG is
   type V TYP is varray(10) of varchar2(200);
  type R TYP is record(c1 pls integer, c2 varchar2(100));
  procedure VARR PROC(p1 in V TYP, p2 OUT V TYP);
  procedure REC_PROC(p1 in R_TYP, p2 OUT R_TYP);
end;
create or replace package body TEST PKG is
procedure VARR PROC(p1 in V TYP, p2 OUT V TYP) is
 begin
   p2 := p1;
  end;
 procedure REC PROC(p1 in R TYP, p2 OUT R TYP) is
 begin
   p2 := p1;
 end;
end;
/
*/
import java.sql.Array;
import java.sql.CallableStatement;
import java.sql.Connection;
```

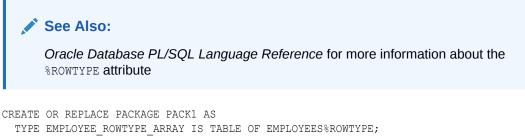


```
import java.sql.SQLException;
import java.sql.Struct;
import java.sql.Types;
import oracle.jdbc.OracleConnection;
public class PLSQLTypesSample
  public static void main(String[] args) throws SQLException
    System.out.println("begin...");
    Connection conn = null;
    oracle.jdbc.pool.OracleDataSource ods = new
oracle.jdbc.pool.OracleDataSource();
    ods.setURL("jdbc:oracle:oci:localhost:5521:orcl");
    ods.setUser("HR");
    ods.setPassword("hr");
    //get connection
    conn = ods.getConnection();
    //call procedure TEST PKG.VARR PROC
    CallableStatement cstmt = null;
    try {
     cstmt = conn.prepareCall("{ call TEST PKG.VARR PROC(?,?) }");
     //PLSQL VARRAY type binding
     Array arr = ((OracleConnection)conn).createArray("TEST PKG.V TYP", new
String[]{"A", "B"});
     cstmt.setArray(1, arr);
     cstmt.registerOutParameter(2, Types.ARRAY, "TEST PKG.V TYP");
     cstmt.execute();
     //get PLSQL VARRAY type out parameter value
     Array outArr = cstmt.getArray(2);
     //...
    }
    catch(Exception e) {
     e.printStackTrace();
    }finally {
     if (cstmt != null)
        cstmt.close();
    }
    //call procedure TEST PKG.REC PROC
    try {
     cstmt = conn.prepareCall("{ call TEST PKG.REC PROC(?,?) }");
      //PLSQL RECORD type binding
     Struct struct = conn.createStruct("TEST PKG.R TYP", new Object[]{12345,
"B"});
     cstmt.setObject(1, struct);
     cstmt.registerOutParameter(2, Types.STRUCT, "TEST PKG.R TYP");
     cstmt.execute();
     //get PLSQL RECORD type out parameter value
     Struct outStruct = (Struct)cstmt.getObject(2);
      //...
    1
    catch(Exception e) {
     e.printStackTrace();
    }finally {
     if (cstmt != null)
        cstmt.close();
    }
    if (conn != null)
```

```
conn.close();
System.out.println("done!");
}
```

Creating Java level objects for each row using %ROWTYPE Attribute

You can create Java-level objects using the <code>%ROWTYPE</code> attribute. In this case, each row of the table is created as a <code>java.sql.Struct</code> object. For example, if you have a package <code>pack1</code> with the following specification:



```
END PACK1;
```

The following code snippet shows how you can retrieve the value of the EMPLOYEE ROWTYPE ARRAY array using JDBC APIs:

This example returns a java.sql.Array of java.sql.Struct objects, where every Struct element represents one row of the EMPLOYEES table.

Example A-1 Creating Struct Objects for Database Table Rows

```
CallableStatement cstmt = conn.prepareCall("BEGIN SELECT * BULK COLLECT INTO :1 FROM
EMPLOYEE; END;");
cstmt.registerOutParameter(1,OracleTypes.ARRAY, "PACK1.EMPLOYEE_ROWTYPE_ARRAY");
cstmt.execute();
Array a = cstmt.getArray(1);
```

A.4 Using Embedded JDBC Escape Syntax

Oracle JDBC drivers support some embedded JDBC escape syntax, which is the syntax that you specify between curly braces. The current support is basic.

Note:

JDBC escape syntax was previously known as SQL92 Syntax or SQL92 escape syntax.

This section describes the support offered by the drivers for the following constructs:

- Time and Date Literals
- Scalar Functions
- LIKE Escape Characters



- MATCH_RECOGNIZE Clause
- Outer Joins
- Function Call Syntax

Where driver support is limited, these sections also describe possible workarounds.

Disabling Escape Processing

The processing for JDBC escape syntax is enabled by default, which results in the JDBC driver performing escape substitution before sending the SQL code to the database. If you want the driver to use regular Oracle SQL syntax, which is more efficient than JDBC escape syntax processing, then use this statement:

```
stmt.setEscapeProcessing(false);
```

A.4.1 Time and Date Literals

Databases differ in the syntax they use for date, time, and timestamp literals. JDBC supports dates and times written only in a specific format. This section describes the formats you must use for date, time, and timestamp literals in SQL statements.

A.4.1.1 Date Literals

The JDBC drivers support date literals in SQL statements written in the format:

```
{d 'yyyy-mm-dd'}
```

Where yyyy-mm-dd represents the year, month, and day. For example:

```
{d '1995-10-22'}
```

The JDBC drivers will replace this escape clause with the equivalent Oracle representation: "22 OCT 1995".

The following code snippet contains an example of using a date literal in a SQL statement.

```
// Connect to the database
// You can put a database name after the @ sign in the connection URL.
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:oracle:oci:@");
ods.setUser("HR");
ods.setPassword("hr");
Connection conn = ods.getConnection();
// Create a Statement
Statement stmt = conn.createStatement ();
// Select the first name column from the employees table where the hire date is
Jan-23-1982
ResultSet rset = stmt.executeQuery
                 ("SELECT first name FROM employees WHERE hire date = {d
'1982-01-23'}");
// Iterate through the result and print the employee names
while (rset.next ())
   System.out.println (rset.getString (1));
```



A.4.1.2 Time Literals

The JDBC drivers support time literals in SQL statements written in the format:

```
{t 'hh:mm:ss'}
```

where, hh:mm:ss represents the hours, minutes, and seconds. For example:

{t '05:10:45'}

The JDBC drivers will replace this escape clause with the equivalent Oracle representation: "05:10:45".

If the time is specified as:

{t '14:20:50'}

Then the equivalent Oracle representation would be "14:20:50", assuming the server is using a 24-hour clock.

This code snippet contains an example of using a time literal in a SQL statement.

A.4.1.3 Timestamp Literals

The JDBC drivers support timestamp literals in SQL statements written in the format:

{ts 'yyyy-mm-dd hh:mm:ss.f...'}

where yyyy-mm-dd hh:mm:ss.f... represents the year, month, day, hours, minutes, and seconds. The fractional seconds portion (.f...) is optional and can be omitted. For example: {ts '1997-11-01 13:22:45'} represents, in Oracle format, NOV 01 1997 13:22:45.

This code snippet contains an example of using a timestamp literal in a SQL statement.

```
ResultSet rset = stmt.executeQuery
    ("SELECT first name FROM employees WHERE hire date = {ts '1982-01-23 12:00:00'}");
```

Mapping SQL DATE Data type to Java

Oracle Database 8*i* and earlier versions did not support TIMESTAMP data, but Oracle DATE data used to have a time component as an extension to the SQL standard. So, Oracle Database 8*i* and earlier versions of JDBC drivers mapped oracle.sql.DATE to java.sql.Timestamp to preserve the time component. Starting with Oracle Database 9.0.1, TIMESTAMP support was included and 9*i* JDBC drivers started mapping oracle.sql.DATE to java.sql.Date. This mapping was incorrect as it truncated the time component of Oracle DATE data. To overcome this problem, Oracle Database 11*g* Release 1 introduced a new flag mapDateToTimestamp. The default value of this flag is true, which means that by default the drivers will correctly map oracle.sql.DATE to java.sql.Timestamp, retaining the time information. If you still want the incorrect but 10*g* compatible oracle.sql.DATE to java.sql.Date mapping, then you can get it by setting the value of mapDateToTimestamp flag to false.



Note:

 Since Oracle Database 11g, if you have an index on a DATE column to be used by a SQL query, then to obtain faster and accurate results, you must use the setObject method in the following way:

```
Date d = parseIsoDate(val);
Timestamp t = new Timestamp(d.getTime());
stmt.setObject(pos, new oracle.sql.DATE(t,
(Calendar)UTC CAL.clone()));
```

This is because if you use the setDate method, then the time component of the Oracle DATE data will be lost and if you use the setTimestamp method, then the index on the DATE column will not be used.

• To overcome the problem of oracle.sql.DATE to java.sql.Date mapping, Oracle Database 9.2 introduced a flag, V8Compatible. The default value of this flag was false, which allowed the mapping of Oracle DATE data to java.sql.Date data. But, users could retain the time component of the Oracle DATE data by setting the value of this flag to true. This flag is desupported since 11g because it controlled Oracle Database 8i compatibility, which is no longer supported.

A.4.2 Scalar Functions

Oracle JDBC drivers do not support all scalar functions. To find out which functions the drivers support, use the following methods supported by the Oracle-specific oracle.jdbc.OracleDatabaseMetaData class and the standard Java java.sql.DatabaseMetadata interface:

getNumericFunctions()

Returns a comma-delimited list of math functions supported by the driver. For example, ABS, COS, SQRT.

• getStringFunctions()

Returns a comma-delimited list of string functions supported by the driver. For example, ASCII, LOCATE.

getSystemFunctions()

Returns a comma-delimited list of system functions supported by the driver. For example, DATABASE, USER.

getTimeDateFunctions()

Returns a comma-delimited list of time and date functions supported by the driver. For example, CURDATE, DAYOFYEAR, HOUR.

Note:

Oracle JDBC drivers support fn, the function keyword.



A.4.3 LIKE Escape Characters

The characters \$ and _ have special meaning in SQL LIKE clauses. You use \$ to match zero or more characters and _ to match exactly one character. If you want to interpret these characters literally in strings, then you precede them with a special escape character. For example, if you want to use ampersand (\$) as the escape character, then you identify it in the SQL statement as:

A.4.4 MATCH_RECOGNIZE Clause

The ? character is used as a token in MATCH_RECOGNIZE clause in Oracle Database 11g and later versions. As the JDBC standard defines the ? character as a parameter marker, the JDBC Driver and the Server SQL Engine cannot distinguish between different uses of the same token.

In earlier versions of JDBC Driver, if you want to interpret the ? character as a MATCH_RECOGNIZE token and not as a parameter marker, then you must use a Statement instead of a PreparedStatement and disable escape processing. However, starting from Oracle Database 12c Release 1 (12.1.0.2), you can use the '{\\ ... \\}' syntax while using the ? character, so that the JDBC driver does not process it as a parameter marker and allows the SQL engine to process it. The following code snippet shows how to use the '{\\ ... \\}' syntax:

```
String sql =
  "select T.firstW, T.lastZ, ? " + // use of parameter marker
  "from tkpattern S11 " +
 "MATCH RECOGNIZE ( " +
  .....
     MEASURES A.cl as firstW, last(Z.cl) as lastZ " +
  ...
      ALL MATCHES " +
  "
     {\\ PATTERN(A? X*? Y+? Z??)\\} " + // use of escape sequence
  ...
     DEFINE " +
  ...
          X as X.c2 > prev(X.c2), " +
  ...
          Y as Y.c2 < prev(Y.c2), " +
  ...
           Z as Z.c2 > prev(Z.c2)" +
  ") as T";
```

```
PreparedStatement ps = conn.prepareStatatement(sql);
ps.setString(1, "test");
ResultSet rs = ps.executeQuery();
```

Related Topics

Using Embedded JDBC Escape Syntax

A.4.5 Outer Joins

Oracle JDBC drivers do not support the outer join syntax. The workaround is to use Oracle outer join syntax:

Instead of:

```
Statement stmt = conn.createStatement ();
ResultSet rset = stmt.executeQuery
   ("SELECT ename, dname
    FROM {OJ dept LEFT OUTER JOIN emp ON dept.deptno = emp.deptno}
    ORDER BY ename");
```

Use Oracle SQL syntax:

```
Statement stmt = conn.createStatement ();
ResultSet rset = stmt.executeQuery
  ("SELECT ename, dname
    FROM emp b, dept a WHERE a.deptno = b.deptno(+)
    ORDER BY ename");
```

A.4.6 Function Call Syntax

Oracle JDBC drivers support the following procedure and function call syntax:

Procedure calls:

{ call procedure_name (argument1, argument2,...) }

Function calls:

{ ? = call procedure_name (argument1, argument2,...) }

A.4.7 JDBC Escape Syntax to Oracle SQL Syntax Example

You can write a simple program to translate JDBC escape syntax to Oracle SQL syntax. The following program prints the comparable Oracle SQL syntax for statements using JDBC escape syntax for function calls, date literals, time literals, and timestamp literals. In the program, the oracle.jdbc.OracleSql class parse() method performs the conversions.

```
public class Foo
{
    static oracle.jdbc.OracleDriver driver = new oracle.jdbc.OracleDriver();
    public static void main (String args[]) throws Exception
    {
        show ("{call foo(?, ?)}");
        show ("{? = call bar (?, ?)}");
        show ("{d '1998-10-22'}");
        show ("{t '16:22:34'}");
        show ("{ts '1998-10-22 16:22:34'}");
    }
}
```



```
}
public static void show (String s) throws Exception
{
   System.out.println (s + " => " +
      driver.processSqlEscapes(s));
}
```

The following code is the output that prints the comparable SQL syntax.

```
{call foo(?, ?)} => BEGIN foo(:1, :2); END;
{? = call bar (?, ?)} => BEGIN :1 := bar (:2, :3); END;
{d '1998-10-22'} => TO_DATE ('1998-10-22', 'YYYY-MM-DD')
{t '16:22:34'} => TO_DATE ('16:22:34', 'HH24:MI:SS')
{ts '1998-10-22 16:22:34'} => TO_TIMESTAMP ('1998-10-22 16:22:34', 'YYYY-MM-DD
HH24:MI:SS.FF')
```

A.5 Oracle JDBC Notes and Limitations

The following limitations exist in the Oracle JDBC implementation, but all of them are either insignificant or have easy workarounds. This section covers the following topics:

CursorName

}

- JDBC Outer Join Escapes
- IEEE 754 Floating Point Compliance
- Catalog Arguments to DatabaseMetaData Calls
- SQLWarning Class
- Executing DDL Statements
- Binding Named Parameters

A.5.1 CursorName

Oracle JDBC drivers do not support the getCursorName and setCursorName methods, because there is no convenient way to map them to Oracle constructs. Oracle recommends using ROWID instead.

Related Topics

Oracle ROWID Type

A.5.2 JDBC Outer Join Escapes

Oracle JDBC drivers do not support JDBC outer join escapes. Use Oracle SQL syntax with + instead.

Related Topics

Using Embedded JDBC Escape Syntax

A.5.3 IEEE 754 Floating Point Compliance

The arithmetic for the Oracle NUMBER type does not comply with the IEEE 754 standard for floating-point arithmetic. Therefore, there can be small disagreements between the results of computations performed by Oracle and the same computations performed by Java.

Oracle stores numbers in a format compatible with decimal arithmetic and guarantees 38 decimal digits of precision. It represents zero, minus infinity, and plus infinity exactly. For each positive number it represents, it represents a negative number of the same absolute value.

It represents every positive number between 10^{-30} and $(1 - 10^{-38}) * 10^{126}$ to full 38-digit precision.

A.5.4 Catalog Arguments to DatabaseMetaData Calls

Certain DatabaseMetaData methods define a catalog parameter. This parameter is one of the selection criteria for the method. Oracle does not have multiple catalogs, but it does have packages.

Related Topics

About Reporting DatabaseMetaData TABLE_REMARKS

A.5.5 SQLWarning Class

The java.sql.SQLWarning class provides information about a database access warning. Warnings typically contain a description of the warning and a code that identifies the warning. Warnings are silently chained to the object whose method caused it to be reported. Oracle JDBC drivers generally do not support SQLWarning. As an exception to this, scrollable result set operations do generate SQL warnings, but the SQLWarning instance is created on the client, not in the database.

Related Topics

About Processing SQL Exceptions

A.5.6 Executing DDL Statements

You must execute Data Definition Language (DDL) statements with Statement objects. If you use PreparedStatements objects or CallableStatements objects, then the DDL statement takes effect only on the first execution. This can cause unexpected behavior if the SQL statements are in a statement cache.

A.5.7 Binding Named Parameters

Binding by name is not supported when using the setXXX methods. Under certain circumstances, previous versions of Oracle JDBC drivers have allowed binding statement variables by name when using the setXXX methods. In the following statement, the named variable EmpId would be bound to the integer 314159.



```
PreparedStatement p = conn.prepareStatement
("SELECT name FROM emp WHERE id = :EmpId");
p.setInt(1, 314159);
```

This capability to bind by name using the setXXX methods is not part of the JDBC specification, and Oracle does not support it. The JDBC drivers can throw a SQLException or produce unexpected results. Starting from Oracle Database 10g JDBC drivers, bind by name is supported using the setXXXAtName methods.

The bound values are not copied by the drivers until you call the execute method. So, changing the bound value before calling the execute method could change the bound value. For example, consider the following code snippet:

```
PreparedStatement p;
.....
Date d = new Date(1181676033917L);
p.setDate(1, d);
d.setTime(0);
p.executeUpdate();
```

This code snippet inserts Date(0) in the database instead of Date(1181676033917L) because the bound values are not copied by JDBC driver implementation for performance reasons.

Related Topics

- Interface oracle.jdbc.OracleCallableStatement
- Interface oracle.jdbc.OraclePreparedStatement



B

Oracle RAC Fast Application Notification

Starting from Oracle Database 12*c* Release 1 (12.1), the Oracle RAC Fast Application Notification (FAN) APIs provide an alternative for taking advantage of the high-availability (HA) features of Oracle Database, if you do not use Universal Connection Pool or Oracle WebLogic Server with Active Grid Link (AGL).

This appendix covers the following topics:

- Overview of Oracle RAC Fast Application Notification
- Installing and Configuring Oracle RAC Fast Application Notification
- Using Oracle RAC Fast Application Notification
- Implementing a Connection Pool

This feature depends on the Oracle Notification System (ONS) message transport mechanism. This feature requires configuring your system, servers, and clients to use ONS.

For using Oracle RAC Fast Application Notification, the simplefan.jar file must be present in the CLASSPATH, and either the ons.jar file must be present in the CLASSPATH or an Oracle Notification Services (ONS) client must be installed and running in the client system.

B.1 Overview of Oracle RAC Fast Application Notification

The Oracle RAC Fast Application Notification (FAN) feature provides a simplified API for accessing FAN events through a callback mechanism. This mechanism enables third-party drivers, connection pools, and containers to subscribe, receive and process FAN events. These APIs are referred to as Oracle RAC FAN APIs in this appendix.

The Oracle RAC FAN APIs provide FAN event notification for developing more responsive applications that can take full advantage of Oracle Database HA features. If you do not want to use Universal Connection Pool, but want to work with FAN events implementing your own connection pool, then you should use Oracle RAC Fast Application Notification.

Note:

- If you do not want to implement your own connection pool, then you should use Oracle Universal Connection Pool to get all the advantages of Oracle RAC Fast Application Notification, along with many additional benefits.
- Starting from Oracle Database 12c Release 1 (12.1), implicit connection cache (ICC) is desupported. Oracle recommends to use Universal Connection Pool instead.

Your applications are enabled to respond to FAN events in the following way:

• Listening for Oracle RAC service down and node down events.



- Listening for Oracle RAC service up events representing any Oracle RAC or Global data Service (GDS) start or restart. For these UP events, FAN parameter status is UP and event_type is one of the following: database, instance, service, Or servicemember.
- Supporting FAN ONS event syntax and fields for both Oracle Database Release 12c and earlier releases, for example, the event_type and timezone fields added to every event, the db_domain field added to every event other than node or public-network events, the percentf field added to Run-Time Load Balancing (RLB) events, and so on.
- Listening for load balancing advisory events and responding to them.

This feature exposes the FAN events, which are the notifications sent by a cluster running Oracle RAC, to inform the subscribers about an event happening at the service level or the node level. The supported FAN events are the following:

Service up

The service up event notifies the connection pool that a new instance is available for use, allowing sessions to be created on the new instance. The ServiceUpEvent Client API is supported in the current release of the Oracle RAC FAN APIs, that is, in the simplefan.jar file.

Service down

The service down events notify that the managed resources are down and currently not available for access. There are two types of service down events:

- Events indicating that a particular instance of a service is down and the instance is no longer able to accept work.
- Events indicating that all-but-one instances of a service are down and the service is no longer able to accept work.
- Node down

The node down events notify that the Oracle RAC node identified by the host identifier is down and not reachable. The cluster sends node down events when a node is no longer able to accept work.

Planned down

Planned down events include all the down events, except node down event. These events have the following two fields set: status=down and reason=user.

Load balancing advisory

The load balancing advisory events provide metrics for load balancing algorithms. Load balancing advisories are sent regularly to inform subscribers of the recommended distribution of work among the available nodes.

Note:

If you want to implement your own connection pool, only then you should use Oracle RAC Fast Application Notification. Otherwise, you should use Oracle Universal Connection Pool to get all the advantages of Oracle RAC Fast Application Notification, along with many additional benefits.



Related Topics

Oracle Universal Connection Pool Developer's Guide

B.2 Installing and Configuring Oracle RAC Fast Application Notification

You can install the Oracle RAC FAN APIs by performing the following steps:

1. Download the simplefan.jar file from the following link:

https://www.oracle.com/database/technologies/appdev/jdbc-downloads.html

- 2. Add the simplefan.jar file to the classpath.
- 3. Perform the following in your Java code:
 - a. Get an instance of the FanManager class by using the getInstance method.
 - **b.** Configure the event daemon using the configure method of the FanManager class. The configure method sets the following properties:

onsNodes: A comma separated list of host:port pairs of ONS daemons that the ONS run time should communicate with. The host in a host:port pair is the host name of a system running the ONS daemon. The port is the local port configuration parameter for that daemon.

onsWalletFile: The path name of the ONS wallet file. The wallet file is the path to a local wallet file used by TLS to store TLS certificates. Same as wallet file configuration parameter to ONS daemon.

onsWalletPassword: The password for accessing the ONS wallet file.

See Also:

- For a detailed description of the Oracle RAC FAN APIs, refer to Oracle Database RAC FAN Events Java API Reference.
- Oracle Universal Connection Pool Developer's Guide

B.3 Using Oracle RAC Fast Application Notification

The following code snippet explains how to handle FAN down events. This example code prints the event data to the standard output device.

This example code demonstrates how to use Oracle RAC FAN APIs by overloading the handleFanEvent method to accept different FAN event notifications as arguments. The example code also displays event data such as:

- Name of the system sending the FAN event notification
- Timestamp of the FAN event notification
- Load status of the FAN event notification



Example B-1 Example of Sample Code Using Oracle RAC FAN API for FAN Down Events

```
. . .
... Properties props = new Properties();
props.putProperty("serviceName", "gl");
FanSubscription sub = FanManager.getInstance().subscribe(props);
sub.addListener(new FanEventListener()) {
  public void handleFanEvent(ServiceDownEvent se) {
    try {
      System.out.println(event.getTimestamp());
      System.out.println(event.getServiceName());
      System.out.println(event.getDatabaseUniqueName());
      System.out.println(event.getReason());
      ServiceMemberEvent me = se.getServiceMemberEvent();
      if (me != null) {
        System.out.println(me.getInstanceName());
        System.out.println(me.getNodeName());
        System.out.println(me.getServiceMemberStatus());
      }
      ServiceCompositeEvent ce = se.getServiceCompositeEvent();
      if (ce != null) {
        System.out.println(ce.getServiceCompositeStatus());
      }
    catch (Throwable t) {
      // handle all exceptions and errors
      t.printStackTrace(System.err);
  public void handleFanEvent(NodeDownEvent ne) {
    try {
      System.out.println(event.getTimestamp());
      System.out.println(ne.getNodeName());
      System.out.println(ne.getIncarnation());
    catch (Throwable t) {
      // handle all exceptions and errors
      t.printStackTrace(System.err);
  public void handleFanEvent(LoadAdvisoryEvent le) {
    try {
      System.out.println(event.getTimestamp());
      System.out.println(le.getServiceName());
      System.out.println(le.getDatabaseUniqueName());
      System.out.println(le.getInstanceName());
      System.out.println(le.getPercent());
      System.out.println(le.getServiceQuality());
      System.out.println(le.getLoadStatus());
    }
    catch (Throwable t) {
      // handle all exceptions and errors
      t.printStackTrace(System.err);
});
```



Example B-2 Example of Sample Code Using Oracle RAC FAN API for FAN Up Events

The following code snippet explains how to use Oracle RAC Fast Application Notification for service up events. The code uses the new Oracle RAC FAN APIs introduced in Oracle Database 12c Release 2 (12.2.0.1), namely, FanUpEventListener interface that extends the FanEventListener interface and ServiceUpEvent. You must implement the FanUpEventListener interface in the client application in the similar way you implement the FanEventListener interface.

```
import oracle.simplefan.*;
. . .
FanEventListener fanListener = new FanUpEventListener() {
public void handleEvent(ServiceUpEvent event) { ..... }
// Specify the next action here, when the node comes up
public void handleEvent(NodeUpEvent event) { ..... }
    . . . . . .
}
FanManager fanMngr = FanManager.getInstance();
Properties onsProps = new Properties();
onsProps.setProperty("onsNodes", .....);
fanMngr.configure(onsProps);
Properties subscriptionProps = new Properties();
subscriptionProps.setProperty("serviceName", .....);
fanSubscription = fanMngr.subscribe(subscriptionProps);
fanSubscription.addListener(fanListener);
. . .
```

B.4 Implementing a Connection Pool

You must implement your own connection pool for using Oracle RAC FAN APIs. Consider the following points before you implement a connection pool using the Oracle RAC FAN APIs:

- Oracle RAC FAN APIs provide a subset of FAN events.
- Oracle RAC FAN APIs support only ONS events. If you want your application to support corresponding supercluster events, then you may require additions to the subscription properties.



C JDBC Coding Tips

This appendix describes methods to optimize a Java Database Connectivity (JDBC) application. It includes the following topics:

- JDBC and Multithreading
- Performance Optimization of JDBC Programs
- Transaction Isolation Levels and Access Modes in JDBC

C.1 JDBC and Multithreading

Oracle JDBC drivers provide full support for, and are highly optimized for, applications that use Java multithreading. Controlled serial access to a connection, such as that provided by connection caching, is both necessary and encouraged. However, Oracle strongly discourages sharing a database connection among multiple threads. Avoid allowing multiple threads to access a connection simultaneously. If multiple threads must share a connection, use a disciplined begin-using/end-using protocol.

Keep the following points in mind while working on multithreaded applications:

- Use the Connection object as a local variable.
- Close the connection in the finally block before exiting the method. For example:

```
Connection conn = null;
try
{
    ...
}
finally
{
    if(conn != null) conn.close();
}
```

- Do not share Connection objects between threads.
- Never synchronize on JDBC objects because it is done internally by the driver.
- Use the Statement.setQueryTimeout method to set the time to execute a query instead of cancelling the long-running query from a different thread.
- Use the Statement.cancel method for SQL operations like SELECT, UPDATE, or DELETE.
- Use the Connection.cancel method for SQL operations like COMMIT, ROLLBACK, and so on.
- Do not use the Thread.interrupt method.

C.2 Performance Optimization of JDBC Programs

You can significantly enhance the performance of your JDBC programs by using any of these features:



- Disabling Auto-Commit Mode
- Standard Fetch Size and Oracle Row Prefetching
- About Setting the Session Data Unit Size
- JDBC Update Batching
- Statement Caching
- Mapping Between Built-in SQL and Java Types

C.2.1 Disabling Auto-Commit Mode

Auto-commit mode indicates to the database whether to issue an automatic COMMIT operation after every SQL operation. Being in auto-commit mode can be expensive in terms of time and processing effort if, for example, you are repeating the same statement with different bind variables.

By default, new connection objects are in auto-commit mode. However, you can disable auto-commit mode with the setAutoCommit method of the connection object, either java.sql.Conection or oracle.jdbc.OracleConnection.

In auto-commit mode, the COMMIT operation occurs either when the statement completes or the next execute occurs, whichever comes first. In the case of statements returning a ResultSet object, the statement completes when the last row of the Result Set has been retrieved or when the Result Set has been closed. In more complex cases, a single statement can return multiple results as well as output parameter values. Here, the COMMIT occurs when all results and output parameter values have been retrieved.

If you disable auto-commit mode with a setAutoCommit(false) call, then you must manually commit or roll back groups of operations using the commit or rollback method of the connection object.

Example

The following example illustrates loading the driver and connecting to the database. Because new connections are in auto-commit mode by default, this example shows how to disable auto-commit. In the example, conn represents the Connection object, and stmt represents the Statement object.

```
// Connect to the database
// You can put a database host name after the @ sign in the connection URL.
    OracleDataSource ods = new OracleDataSource();
    ods.setURL("jdbc:oracle:oci:@");
    ods.setUser("HR");
    ods.setPassword("hr");
    Connection conn = ods.getConnection();
// It's faster when auto commit is off
conn.setAutoCommit (false);
// Create a Statement
Statement stmt = conn.createStatement ();
...
```



C.2.2 Standard Fetch Size and Oracle Row Prefetching

Oracle JDBC connection and statement objects allow you to specify the number of rows to prefetch into the client with each trip to the database while a result set is being populated during a query. You can set a value in a connection object that affects each statement produced through that connection, and you can override that value in any particular statement object. The default value in a connection object is 10. Prefetching data into the client reduces the number of round-trips to the server.

Similarly, and with more flexibility, JDBC 2.0 enables you to specify the number of rows to fetch with each trip, both for statement objects (affecting subsequent queries) and for result set objects (affecting row refetches). By default, a result set uses the value for the statement object that produced it. If you do not set the JDBC 2.0 fetch size, then the Oracle connection row-prefetch value is used by default.

Related Topics

Row Fetch Size

C.2.3 About Setting the Session Data Unit Size

Session data unit (SDU) is a buffer that Oracle Net uses to place data before transmitting it across the network. Oracle Net sends the data in the buffer either when the request is completed or when it is full.

You can configure the SDU and obtain the following benefits, among others:

- · Reduction in the time required to transmit a SQL query and result across the network
- Transmission of larger chunks of data

Note:

The footprint of the client and the server process increase if you set a bigger SDU size.

See Also:

Oracle Database Net Services Administrator's Guide

This section describes the following:

- About Setting the SDU Size for the Database Server
- About Setting the SDU Size for JDBC Thin Client

C.2.3.1 About Setting the SDU Size for the Database Server

To set the SDU size for the database server, configure the DEFAULT_SDU_SIZE parameter in the sqlnet.ora file.



C.2.3.2 About Setting the SDU Size for JDBC OCI Client

The JDBC OCI client uses Oracle Net layer. So, you can set the SDU size for the JDBC OCI client by configuring the DEFAULT_SDU_SIZE parameter in the sqlnet.ora file.

C.2.3.3 About Setting the SDU Size for JDBC Thin Client

You can set the SDU size for JDBC thin client by specifying it in the DESCRIPTION parameter for a particular connection descriptor.

```
sales.example.com=
(DESCRIPTION=
    (SDU=11280)
        (ADDRESS=(PROTOCOL=tcp)(HOST=sales-server)(PORT=5221))
    (CONNECT_DATA=
        (SERVICE_NAME=sales.example.com))
)
```

C.2.4 JDBC Update Batching

Oracle JDBC drivers enable you to accumulate INSERT, DELETE, and UPDATE operations of prepared statements at the client and send them to the server in batches. This feature reduces round-trips to the server.

Note:

Oracle recommends to keep the batch sizes in the range of 100 or less. Larger batches provide little or no performance improvement and may actually reduce performance due to the client resources required to handle the large batch.

C.2.5 Statement Caching

Statement caching improves performance by caching executable statements that are used repeatedly, such as in a loop or in a method that is called repeatedly. Applications use the statement cache to cache statements associated with a particular physical connection. When you enable Statement caching, a Statement object is cached when you call the close method. Because each physical connection has its own cache, multiple caches can exist if you enable Statement caching for multiple physical connections.

Note:

The Oracle JDBC drivers are optimized for use with the Oracle Statement cache. Oracle strongly recommends that you use the Oracle Statement cache (implicit or explicit).



When you enable Statement caching on a connection cache, the logical connections benefit from the Statement caching that is enabled on the underlying physical connection. If you try to enable Statement caching on a logical connection held by a connection cache, then this will throw an exception.

Related Topics

Statement and Result Set Caching

C.2.6 Mapping Between Built-in SQL and Java Types

The SQL built-in types are those types with system-defined names, such as NUMBER, and CHAR, as opposed to the Oracle objects, varray, and nested table types, which have userdefined names. In JDBC programs that access data of built-in SQL types, all type conversions are unambiguous, because the program context determines the Java type to which a SQL datum will be converted.

SQL Data Type	ORACLE Mapping - Java Classes Representing SQL Data Types
CHAR	oracle.sql.CHAR
VARCHAR2	oracle.sql.CHAR
DATE	oracle.sql.DATE
DECIMAL	oracle.sql.NUMBER
DOUBLE PRECISION	oracle.sql.NUMBER
FLOAT	oracle.sql.NUMBER
INTEGER	oracle.sql.NUMBER
REAL	oracle.sql.NUMBER
RAW	oracle.sql.RAW
LONG RAW	oracle.sql.RAW
REF CURSOR	java.sql.ResultSet
CLOB LOCATOR	oracle.jdbc.OracleClob ¹
BLOB LOCATOR	oracle.jdbc.OracleBlob ²
BFILE	oracle.sql.BFILE
nested table	oracle.jdbc.OracleArray ³
varray	oracle.jdbc.OracleArray
SQL object value	If there is no entry for the object value in the type map:
	 oracle.jdbc.OracleStruct⁴
	If there is an entry for the object value in the type map:
	customized Java class
REF to SQL object type	<pre>class that implements oracle.sql.SQLRef, typically by implementing oracle.jdbc.OracleRef⁵</pre>

Table C-1Mapping of SQL Data Types to Java Classes that Represent SQL DataTypes

¹ Starting from Oracle Database 12c Release 1 (12.1), the oracle.sql.CLOB class is deprecated and replaced with the oracle.jdbc.OracleClob interface.



- ² Starting from Oracle Database 12c Release 1 (12.1), the oracle.sql.BLOB class is deprecated and replaced with the oracle.jdbc.OracleBlob interface.
- ³ Starting from Oracle Database 12c Release 1 (12.1), the oracle.sql.ARRAY class is deprecated and replaced with the oracle.jdbc.OracleArray interface.
- ⁴ Starting from Oracle Database 12c Release 1 (12.1), the oracle.sql.STRUCT class is deprecated and replaced with the oracle.jdbc.OracleStruct interface.
- ⁵ Starting from Oracle Database 12c Release 1 (12.1), the oracle.sql.REF class is deprecated and replaced with the oracle.jdbc.OracleRef interface.

The most efficient way to access numeric data is to use primitive Java types like int, float, long, and double. However, the range of values of these types do not exactly match the range of values of the SQL NUMBER data type. As a result, there may be some loss of information. If absolute precision is required across the entire value range, then use the BigDecimal type.

All character data is converted to the UCS2 character set of Java. The most efficient way to access character data is as java.lang.String. In worst case, this can cause a loss of information when two or more characters in the database character set map to a single UCS2 character. Since Oracle Database 11g, all characters in the character set map to the characters in the UCS2 character set. However, some characters do map to surrogate pairs.

C.3 Transaction Isolation Levels and Access Modes in JDBC

Read-only connections are supported by Oracle JDBC drivers, but not by the Oracle server.

For transactions, the Oracle server supports only the TRANSACTION_READ_COMMITTED and TRANSACTION_SERIALIZABLE transaction isolation levels. The default is TRANSACTION_READ_COMMITTED. Use the following methods of the oracle.jdbc.OracleConnection interface to get and set the level:

- getTransactionIsolation: Gets the current transaction isolation level of the connection.
- setTransactionIsolation: Changes the transaction isolation level, using either the TRANSACTION READ COMMITTED or the TRANSACTION SERIALIZABLE value.



D JDBC Error Messages

This appendix briefly discusses the general structure of Java Database Connectivity (JDBC) error messages, then lists general JDBC error messages and TTC error messages that Oracle JDBC drivers can return. The appendix is organized as follows:

- General Structure of JDBC Error Messages
- General JDBC Messages
- Native XA Messages
- TTC Messages

Each of the message lists is first sorted by ORA number, and then alphabetically.

D.1 General Structure of JDBC Error Messages

The general JDBC error message structure allows run-time information to be appended to the end of a message, following a colon, as follows:

```
<error message>:<extra info>
```

For example, a "closed statement" error might be displayed as follows:

Closed Statement:next

This indicates that the exception was thrown during a call to the next method (of a result set object).

In some cases, the user can find the same information in a stack trace.

D.2 General JDBC Messages

This section lists general JDBC error messages, first sorted by the ORA number, and then in alphabetic order in the following subsections:

- JDBC Messages Sorted by ORA Number
- JDBC Messages Sorted in Alphabetic Order

Note:

The ORA-17033 and ORA-17034 error messages use the term SQL92. The JDBC escape syntax was previously known as SQL92 Syntax or SQL92 escape syntax.

D.2.1 JDBC Messages Sorted by ORA Number

The following table lists the JDBC error messages sorted by the ORA number:



ORA Number	Message
ORA-17001	Internal Error
ORA-17002	lo exception
ORA-17003	Invalid column index
ORA-17004	Invalid column type
ORA-17005	Unsupported column type
ORA-17006	Invalid column name
ORA-17007	Invalid dynamic column
ORA-17008	Closed Connection
ORA-17009	Closed Statement
ORA-17010	Closed Resultset
ORA-17011	Exhausted Resultset
ORA-17012	Parameter Type Conflict
ORA-17014	ResultSet.next was not called
ORA-17015	Statement was cancelled
ORA-17016	Statement timed out
ORA-17017	Cursor already initialized
ORA-17018	Invalid cursor
ORA-17019	Can only describe a query
ORA-17020	Invalid row prefetch
ORA-17021	Missing defines
ORA-17022	Missing defines at index
ORA-17023	Unsupported feature
ORA-17024	No data read
ORA-17025	Error in defines.isNull ()
ORA-17026	Numeric Overflow
ORA-17027	Stream has already been closed
ORA-17028	Can not do new defines until the current ResultSet is closed
ORA-17029	setReadOnly: Read-only connections not supported
ORA-17030	READ_COMMITTED and SERIALIZABLE are the only valid transaction levels
ORA-17031	setAutoClose: Only support auto close mode on
ORA-17032	cannot set row prefetch to zero
ORA-17033	Malformed SQL92 string at position
ORA-17034	Non supported SQL92 token at position
ORA-17035	Character Set Not Supported !!
ORA-17036	exception in OracleNumber
ORA-17037	Fail to convert between UTF8 and UCS2
ORA-17038	Byte array not long enough

Table D-1 JDBC Messages Sorted by ORA Number



ORA Number	Message
ORA-17039	Char array not long enough
ORA-17040	Sub Protocol must be specified in connection URL
ORA-17041	Missing IN or OUT parameter at index:
ORA-17042	Invalid Batch Value
ORA-17043	Invalid stream maximum size
ORA-17044	Internal error: Data array not allocated
ORA-17045	Internal error: Attempt to access bind values beyond the batch value
ORA-17046	Internal error: Invalid index for data access
ORA-17047	Error in Type Descriptor parse
ORA-17048	Undefined type
ORA-17049	Inconsistent java and sql object types
ORA-17050	no such element in vector
ORA-17051	This API cannot be be used for non-UDT types
ORA-17052	This ref is not valid
ORA-17053	The size is not valid
ORA-17054	The LOB locator is not valid
ORA-17055	Invalid character encountered in
ORA-17056	Non supported character set (add orai18n.jar in your classpath
ORA-17057	Closed LOB
ORA-17058	Internal error: Invalid NLS Conversion ratio
ORA-17059	Fail to convert to internal representation
ORA-17060	Fail to construct descriptor
ORA-17061	Missing descriptor
ORA-17062	Ref cursor is invalid
ORA-17063	Not in a transaction
ORA-17064	Invalid Sytnax or Database name is null
ORA-17065	Conversion class is null
ORA-17066	Access layer specific implementation needed
ORA-17067	Invalid Oracle URL specified
ORA-17068	Invalid argument(s) in call
ORA-17069	Use explicit XA call
ORA-17070	Data size bigger than max size for this type
ORA-17071	Exceeded maximum VARRAY limit
ORA-17072	Inserted value too large for column
ORA-17074	invalid name pattern
ORA-17075	Invalid operation for forward only resultset
ORA-17076	Invalid operation for read only resultset

Table D-1 (Cont.) JDBC Messages Sorted by ORA Number



ORA Number	Message
ORA-17077	Fail to set REF value
ORA-17078	Cannot do the operation as connections are already opened
ORA-17079	User credentials doesn't match the existing ones
ORA-17080	invalid batch command
ORA-17081	error occurred during batching
ORA-17082	No current row
ORA-17083	Not on the insert row
ORA-17084	Called on the insert row
ORA-17085	Value conflicts occurs
ORA-17086	Undefined column value on the insert row
ORA-17087	Ignored performance hint: setFetchDirection()
ORA-17088	Unsupported syntax for requested resultset type and concurrency level
ORA-17089	internal error
ORA-17090	operation not allowed
ORA-17091	Unable to create resultset at the requested type and/or concurrency level
ORA-17092	JDBC statements cannot be created or executed at end of call processing
ORA-17093	OCI operation returned OCI_SUCCESS_WITH_INFO
ORA-17094	Object type version mismatched
ORA-17095	Statement cache size has not been set
ORA-17096	Statement Caching cannot be enabled for this logical connection.
ORA-17097	Invalid PL/SQL Index Table element type
ORA-17098	Invalid empty lob operation
ORA-17099	Invalid PL/SQL Index Table array length
ORA-17100	Invalid database Java Object
ORA-17101	Invalid properties in OCI Connection Pool Object
ORA-17102	Bfile is read only
ORA-17103	invalid connection type to return via getConnection. Use getJavaSqlConnection instead
ORA-17104	SQL statement to execute cannot be empty or null
ORA-17105	connection session time zone was not set
ORA-17106	invalid JDBC-OCI driver connection pool configuration specified
ORA-17107	invalid proxy type specified
ORA-17108	No max length specified in defineColumnType
ORA-17109	standard Java character encoding not found
ORA-17110	execution completed with warning
ORA-17111	Invalid connection cache TTL timeout specified

Table D-1 (Cont.) JDBC Messages Sorted by ORA Number



ORA Number	Message
ORA-17112	Invalid thread interval specified
ORA-17113	Thread interval value is more than the cache timeout value
ORA-17114	could not use local transaction commit in a global transaction
ORA-17115	could not use local transaction rollback in a global transaction
ORA-17116	could not turn on auto-commit in an active global transaction
ORA-17117	could not set savepoint in an active global transaction
ORA-17118	could not obtain ID for a named Savepoint
ORA-17119	could not obtain name for an un-named Savepoint
ORA-17120	could not set a Savepoint with auto-commit on
ORA-17121	could not rollback to a Savepoint with auto-commit on
ORA-17122	could not rollback to a local txn Savepoint in a global transaction
ORA-17123	Invalid statement cache size specified
ORA-17124	Invalid connection cache Inactivity timeout specified
ORA-17125	Improper statement type returned by explicit cache
ORA-17126	Fixed Wait timeout elapsed
ORA-17127	Invalid Fixed Wait timeout specified
ORA-17128	SQL string is not Query
ORA-17129	SQL string is not a DML Statement
ORA-17132	Invalid conversion requested
ORA-17133	UNUSED
ORA-17134	Length of named parameter in SQL exceeded 32 characters
ORA-17135	Parameter name used in setXXXStream appears more than once in SQL
ORA-17136	Malformed DATALINK URL, try getString() instead
ORA-17137	Connection Caching Not Enabled or Not a Valid Cache Enabled DataSource
ORA-17138	Invalid Connection Cache Name. Must be a valid String and Unique
ORA-17139	Invalid Connection Cache Properties
ORA-17140	Connection Cache with this Cache Name already exists
ORA-17141	Connection Cache with this Cache Name does not exist
ORA-17142	Connection Cache with this Cache Name is Disabled
ORA-17143	Invalid or Stale Connection found in the Connection Cache
ORA-17144	statement handle not executed
ORA-17145	Invalid ONS Event received
ORA-17146	Invalid ONS Event Version received
ORA-17147	Attempt to set a parameter name that does not occur in the SQL
ORA-17148	Method only implemented in thin
ORA-17149	This is already a proxy session

Table D-1 (Cont.) JDBC Messages Sorted by ORA Number



ORA Number	Message
ORA-17150	Wrong arguments for proxy session
ORA-17151	Clob is too large to be stored in a Java String
ORA-17152	This method is only implemented in logical connections
ORA-17153	This method is only implemented in physical connections
ORA-17154	Cannot map Oracle character to Unicode
ORA-17155	Cannot map Unicode to Oracle character
ORA-17156	Invalid array size for End-to-End metrics values
ORA-17157	setString can only process strings of less than 32766 chararacters
ORA-17158	duration is invalid for this function
ORA-17159	metric value for end-to-end tracing is too long
ORA-17160	execution context id sequence number out of range
ORA-17161	Invalid transaction mode used
ORA-17162	Unsupported holdability value
ORA-17163	Can not use getXAConnection() when connection caching is enabled
ORA-17164	Can not call getXAResource() from physical connection with caching on
ORA-17165	DBMS_JDBC package not preset in server for this connection
ORA-17166	Cannot perform fetch on a PLSQL statement
ORA-17167	PKI classes not found. To use 'connect /' functionality, oraclepki.jar must be in the classpath
ORA-17168	encountered a problem with the Secret Store. Check the wallet location for the presence of an open wallet (cwallet.sso) and ensure that this wallet contains the correct credentials using the mkstore utility
ORA-17169	Cannot bind stream to a ScrollableResultSet or UpdatableResultSet
ORA-17170	The Namespace cannot be empty
ORA-17171	The attribute length cannot exceed 30 chars
ORA-17172	That value of the attribute cannot exceed 400 chars
ORA-17173	Not all return parameters registered
ORA-17174	The only supported namespace is CLIENTCONTEXT
ORA-17175	Error during remote ONS configuration
ORA-17259	SQLXML cannot find the XML support jar file in the classpath
ORA-17260	Attempt to read an empty SQLXML
ORA-17261	Attempt to read a SQLXML that is not readable
ORA-17262	Attempt to write a SQLXML that is not writeable
ORA-17263	SQLXML cannot create a Result of that type
ORA_17264	SQLXML cannoct create a Source of that type

Table D-1 (Cont.) JDBC Messages Sorted by ORA Number



D.2.2 JDBC Messages Sorted in Alphabetic Order

The following table lists the JDBC error messages sorted in alphabetic order:

 Table D-2
 JDBC Messages Sorted in Alphabetic Order

ORA Number	Message
ORA-17066	Access layer specific implementation needed
ORA-17261	Attempt to read a SQLXML that is not readable
ORA-17260	Attempt to read an empty SQLXML
ORA-17147	Attempt to set a parameter name that does not occur in the SQL
ORA-17262	Attempt to write a SQLXML that is not writeable
ORA-17102	Bfile is read only
ORA-17038	Byte array not long enough
ORA-17084	Called on the insert row
ORA-17164	Can not call getXAResource() from physical connection with caching on
ORA-17028	Can not do new defines until the current ResultSet is closed
ORA-17163	Can not use getXAConnection() when connection caching is enabled
ORA-17019	Can only describe a query
ORA-17169	Cannot bind stream to a ScrollableResultSet or UpdatableResultSet
ORA-17078	Cannot do the operation as connections are already opened
ORA-17154	Cannot map Oracle character to Unicode
ORA-17155	Cannot map Unicode to Oracle character
ORA-17166	Cannot perform fetch on a PLSQL statement
ORA-17032	Cannot set row prefetch to zero
ORA-17039	Char array not long enough
ORA-17035	Character Set Not Supported !!
ORA-17151	Clob is too large to be stored in a Java String
ORA-17008	Closed Connection
ORA-17057	Closed LOB
ORA-17010	Closed Resultset
ORA-17009	Closed Statement
ORA-17140	Connection Cache with this Cache Name already exists
ORA-17141	Connection Cache with this Cache Name does not exist
ORA-17142	Connection Cache with this Cache Name is Disabled
ORA-17137	Connection Caching Not Enabled or Not a Valid Cache Enabled DataSource
ORA-17105	Connection session time zone was not set
ORA-17065	Conversion class is null
ORA-17118	Could not obtain ID for a named Savepoint
ORA-17119	Could not obtain name for an un-named Savepoint



ORA Number	Message
ORA-17122	Could not rollback to a local txn Savepoint in a global transaction
ORA-17121	Could not rollback to a Savepoint with auto-commit on
ORA-17120	Could not set a Savepoint with auto-commit on
ORA-17117	Could not set savepoint in an active global transaction
ORA-17116	Could not turn on auto-commit in an active global transaction
ORA-17114	Could not use local transaction commit in a global transaction
ORA-17115	Could not use local transaction rollback in a global transaction
ORA-17017	Cursor already initialized
ORA-17070	Data size bigger than max size for this type
ORA-17165	DBMS_JDBC package not preset in server for this connection
ORA-17158	Duration is invalid for this function
ORA-17168	Encountered a problem with the Secret Store. Check the wallet location for the presence of an open wallet (cwallet.sso) and ensure that this wallet contains the correct credentials using the mkstore utility
ORA-17175	Error during remote ONS configuration
ORA-17025	Error in defines.isNull ()
ORA-17047	Error in Type Descriptor parse
ORA-17081	error occurred during batching
ORA-17071	Exceeded maximum VARRAY limit
ORA-17036	Exception in OracleNumber
ORA-17110	Execution completed with warning
ORA-17160	Execution context id sequence number out of range
ORA-17011	Exhausted Resultset
ORA-17060	Fail to construct descriptor
ORA-17037	Fail to convert between UTF8 and UCS2
ORA-17059	Fail to convert to internal representation
ORA-17077	Fail to set REF value
ORA-17126	Fixed Wait timeout elapsed
ORA-17087	Ignored performance hint: setFetchDirection()
ORA-17125	Improper statement type returned by explicit cache
ORA-17049	Inconsistent java and sql object types
ORA-17072	Inserted value too large for column
ORA-17001	Internal Error
ORA-17089	internal error
ORA-17045	Internal error: Attempt to access bind values beyond the batch value
ORA-17044	Internal error: Data array not allocated
ORA-17046	Internal error: Invalid index for data access
ORA-17058	Internal error: Invalid NLS Conversion ratio

Table D-2 (Cont.) JDBC Messages Sorted in Alphabetic Order



ORA Number	Message
ORA-17068	Invalid argument(s) in call
ORA-17156	Invalid array size for End-to-End metrics values
ORA-17080	invalid batch command
ORA-17042	Invalid Batch Value
ORA-17055	Invalid character encountered in
ORA-17003	Invalid column index
ORA-17006	Invalid column name
ORA-17004	Invalid column type
ORA-17124	Invalid connection cache Inactivity timeout specified
ORA-17138	Invalid Connection Cache Name. Must be a valid String and Unique
ORA-17139	Invalid Connection Cache Properties
ORA-17111	Invalid connection cache TTL timeout specified
ORA-17103	invalid connection type to return via getConnection. Use getJavaSqlConnection instead
ORA-17132	Invalid conversion requested
ORA-17018	Invalid cursor
ORA-17100	Invalid database Java Object
ORA-17007	Invalid dynamic column
ORA-17098	Invalid empty lob operation
ORA-17127	Invalid Fixed Wait timeout specified
ORA-17106	invalid JDBC-OCI driver connection pool configuration specified
ORA-17074	invalid name pattern
ORA-17145	Invalid ONS Event received
ORA-17146	Invalid ONS Event Version received
ORA-17075	Invalid operation for forward only resultset
ORA-17076	Invalid operation for read only resultset
ORA-17143	Invalid or Stale Connection found in the Connection Cache
ORA-17067	Invalid Oracle URL specified
ORA-17099	Invalid PL/SQL Index Table array length
ORA-17097	Invalid PL/SQL Index Table element type
ORA-17101	Invalid properties in OCI Connection Pool Object
ORA-17107	invalid proxy type specified
ORA-17020	Invalid row prefetch
ORA-17123	Invalid statement cache size specified
ORA-17043	Invalid stream maximum size
ORA-17064	Invalid Sytnax or Database name is null
ORA-17112	Invalid thread interval specified
ORA-17161	Invalid transaction mode used

Table D-2 (Cont.) JDBC Messages Sorted in Alphabetic Order



ORA Number	Message
ORA-17002	lo exception
ORA-17092	JDBC statements cannot be created or executed at end of call processing
ORA-17134	Length of named parameter in SQL exceeded 32 characters
ORA-17136	Malformed DATALINK URL, try getString() instead
ORA-17033	Malformed SQL92 string at position
ORA-17148	Method only implemented in thin
ORA-17159	metric value for end-to-end tracing is too long
ORA-17021	Missing defines
ORA-17022	Missing defines at index
ORA-17061	Missing descriptor
ORA-17041	Missing IN or OUT parameter at index:
ORA-17082	No current row
ORA-17024	No data read
ORA-17108	No max length specified in defineColumnType
ORA-17050	no such element in vector
ORA-17056	Non supported character set (add orai18n.jar in your classpath)
ORA-17034	Non supported SQL92 token at position
ORA-17173	Not all return parameters registered
ORA-17063	Not in a transaction
ORA-17083	Not on the insert row
ORA-17026	Numeric Overflow
ORA-17094	Object type version mismatched
ORA-17093	OCI operation returned OCI_SUCCESS_WITH_INFO
ORA-17090	operation not allowed
ORA-17135	Parameter name used in setXXXStream appears more than once in SQL
ORA-17012	Parameter Type Conflict
ORA-17167	PKI classes not found. To use 'connect /' functionality, oraclepki.jar must be in the classpath
ORA-17030	READ_COMMITTED and SERIALIZABLE are the only valid transaction levels
ORA-17062	Ref cursor is invalid
ORA-17014	ResultSet.next was not called
ORA-17031	setAutoClose: Only support auto close mode on
ORA-17029	setReadOnly: Read-only connections not supported
ORA-17157	setString can only process strings of less than 32766 chararacters
ORA-17104	SQL statement to execute cannot be empty or null
ORA-17129	SQL string is not a DML Statement

Table D 2	(Cont.) IDBC Massages Serted in Alphabetic Order
Table D-2	(Cont.) JDBC Messages Sorted in Alphabetic Order



ORA Number	Message
ORA-17128	SQL string is not Query
ORA-17263	SQLXML cannot create a Result of that type
ORA_17264	SQLXML cannoct create a Source of that type
ORA-17259	SQLXML cannot find the XML support jar file in the classpath
ORA-17109	standard Java character encoding not found
ORA-17095	Statement cache size has not been set
ORA-17096	Statement Caching cannot be enabled for this logical connection.
ORA-17144	statement handle not executed
ORA-17016	Statement timed out
ORA-17015	Statement was cancelled
ORA-17027	Stream has already been closed
ORA-17040	Sub Protocol must be specified in connection URL
ORA-17172	That value of the attribute cannot exceed 400 chars
ORA-17171	The attribute length cannot exceed 30 chars
ORA-17054	The LOB locator is not valid
ORA-17170	The Namespace cannot be empty
ORA-17174	The only supported namespace is CLIENTCONTEXT
ORA-17053	The size is not valid
ORA-17051	This API cannot be be used for non-UDT types
ORA-17149	This is already a proxy session
ORA-17152	This method is only implemented in logical connections
ORA-17153	This method is only implemented in physical connections
ORA-17052	This ref is not valid
ORA-17113	Thread interval value is more than the cache timeout value
ORA-17091	Unable to create resultset at the requested type and/or concurrency level
ORA-17086	Undefined column value on the insert row
ORA-17048	Undefined type
ORA-17005	Unsupported column type
ORA-17023	Unsupported feature
ORA-17162	Unsupported holdability value
ORA-17088	Unsupported syntax for requested resultset type and concurrency level
ORA-17133	UNUSED
ORA-17069	Use explicit XA call
ORA-17079	User credentials doesn't match the existing ones
ORA-17085	Value conflicts occurs
ORA-17150	Wrong arguments for proxy session

 Table D-2
 (Cont.) JDBC Messages Sorted in Alphabetic Order



D.3 Native XA Messages

The following sections cover the JDBC error messages that are specific to the Native XA feature:

- Native XA Messages Sorted by ORA Number
- Native XA Messages Sorted in Alphabetic Order

D.3.1 Native XA Messages Sorted by ORA Number

The following table lists the Native XA messages sorted by the ORA number:

Table D-3 Native XA Messages Sorted by ORA Number

ORA Number	Message
ORA-17200	Unable to properly convert XA open string from Java to C
ORA-17201	Unable to properly convert XA close string from Java to C
ORA-17202	Unable to properly convert RM name from Java to C
ORA-17203	Could not casting pointer type to jlong
ORA-17204	Input array too short to hold OCI handles
ORA-17205	Failed to obtain OCISvcCtx handle from C-XA using xaoSvcCtx
ORA-17206	Failed to obtain OCIEnv handle from C-XA using xaoEnv
ORA-17207	The tnsEntry property was not set in DataSource
ORA-17213	C-XA returned XAER_RMERR during xa_open
ORA-17215	C-XA returned XAER_INVAL during xa_open
ORA-17216	C-XA returned XAER_PROTO during xa_open
ORA-17233	C-XA returned XAER_RMERR during xa_close
ORA-17235	C-XA returned XAER_INVAL during xa_close
ORA-17236	C-XA returned XAER_PROTO during xa_close

D.3.2 Native XA Messages Sorted in Alphabetic Order

The following table lists the Native XA messages sorted in the alphabetic order:

ORA Number	Message
ORA-17203	Could not casting pointer type to jlong
ORA-17235	C-XA returned XAER_INVAL during xa_close
ORA-17215	C-XA returned XAER_INVAL during xa_open
ORA-17236	C-XA returned XAER_PROTO during xa_close
ORA-17216	C-XA returned XAER_PROTO during xa_open
ORA-17233	C-XA returned XAER_RMERR during xa_close
ORA-17213	C-XA returned XAER_RMERR during xa_open

Table D-4 Native XA Messages Sorted in Alphabetic Order



ORA Number	Message
ORA-17206	Failed to obtain OCIEnv handle from C-XA using xaoEnv
ORA-17205	Failed to obtain OCISvcCtx handle from C-XA using xaoSvcCtx
ORA-17204	Input array too short to hold OCI handles
ORA-17207	The tnsEntry property was not set in DataSource
ORA-17202	Unable to properly convert RM name from Java to C
ORA-17201	Unable to properly convert XA close string from Java to C
ORA-17200	Unable to properly convert XA open string from Java to C

Table D-4 (Cont.) Native XA Messages Sorted in Alphabetic Order

D.4 TTC Messages

This section lists TTC error messages, first sorted by the ORA number and then in alphabetic order in the following subsections:

- TTC Messages Sorted by ORA Number
- TTC Messages Sorted in Alphabetic Order

D.4.1 TTC Messages Sorted by ORA Number

The following table lists the TTC messages sorted by the ORA number:

ORA Number	Message
ORA-17401	Protocol violation
ORA-17402	Only one RPA message is expected
ORA-17403	Only one RXH message is expected
ORA-17404	Received more RXDs than expected
ORA-17405	UAC length is not zero
ORA-17406	Exceeding maximum buffer length
ORA-17407	invalid Type Representation(setRep)
ORA-17408	invalid Type Representation(getRep)
ORA-17409	invalid buffer length
ORA-17410	No more data to read from socket
ORA-17411	Data Type representations mismatch
ORA-17412	Bigger type length than Maximum
ORA-17413	Exceding key size
ORA-17414	Insufficient Buffer size to store Columns Names
ORA-17415	This type hasn't been handled
ORA-17416	FATAL
ORA-17417	NLS Problem, failed to decode column names

Table D-5 TTC Messages Sorted by ORA Number



ORA Number	Message
ORA-17418	Internal structure's field length error
ORA-17419	Invalid number of columns returned
ORA-17420	Oracle Version not defined
ORA-17421	Types or Connection not defined
ORA-17422	Invalid class in factory
ORA-17423	Using a PLSQL block without an IOV defined
ORA-17424	Attempting different marshaling operation
ORA-17425	Returning a stream in PLSQL block
ORA-17426	Both IN and OUT binds are NULL
ORA-17427	Using Uninitialized OAC
ORA-17428	Logon must be called after connect
ORA-17429	Must be at least connected to server
ORA-17430	Must be logged on to server
ORA-17431	SQL Statement to parse is null
ORA-17432	invalid options in all7
ORA-17433	invalid arguments in call
ORA-17434	not in streaming mode
ORA-17435	invalid number of in_out_binds in IOV
ORA-17436	invalid number of outbinds
ORA-17437	Error in PLSQL block IN/OUT argument(s)
ORA-17438	Internal - Unexpected value
ORA-17439	Invalid SQL type
ORA-17440	DBItem/DBType is null
ORA-17441	Oracle Version not supported. Minimum supported version is 7.2.3.
ORA-17442	Refcursor value is invalid
ORA-17443	Null user or password not supported in THIN driver
ORA-17444	TTC Protocol version received from server not supported
ORA-17445	LOB already opened in the same transaction
ORA-17446	LOB already closed in the same transaction
ORA-17447	OALL8 is in an inconsistent state

Table D-5 (Cont.) TTC Messages Sorted by ORA Number

D.4.2 TTC Messages Sorted in Alphabetic Order

The following table lists the TTC messages in the alphabetic order:

Table D-6 TTC Messages Sorted in Alphabetic Order

ORA Number	Message
ORA-17424	Attempting different marshaling operation



ORA Number	Message
ORA-17412	Bigger type length than Maximum
ORA-17426	Both IN and OUT binds are NULL
ORA-17411	Data Type representations mismatch
ORA-17440	DBItem/DBType is null
ORA-17437	Error in PLSQL block IN/OUT argument(s)
ORA-17413	Exceding key size
ORA-17406	Exceeding maximum buffer length
ORA-17416	FATAL
ORA-17414	Insufficient Buffer size to store Columns Names
ORA-17438	Internal - Unexpected value
ORA-17418	Internal structure's field length error
ORA-17433	invalid arguments in call
ORA-17409	invalid buffer length
ORA-17422	Invalid class in factory
ORA-17419	Invalid number of columns returned
ORA-17435	invalid number of in_out_binds in IOV
ORA-17436	invalid number of outbinds
ORA-17432	invalid options in all7
ORA-17439	Invalid SQL type
ORA-17408	invalid Type Representation(getRep)
ORA-17407	invalid Type Representation(setRep)
ORA-17446	LOB already closed in the same transaction
ORA-17445	LOB already opened in the same transaction
ORA-17428	Logon must be called after connect
ORA-17429	Must be at least connected to server
ORA-17430	Must be logged on to server
ORA-17417	NLS Problem, failed to decode column names
ORA-17410	No more data to read from socket
ORA-17434	not in streaming mode
ORA-17443	Null user or password not supported in THIN driver
ORA-17447	OALL8 is in an inconsistent state
ORA-17402	Only one RPA message is expected
ORA-17403	Only one RXH message is expected
ORA-17420	Oracle Version not defined
ORA-17441	Oracle Version not supported. Minimum supported version is 7.2.3.
ORA-17401	Protocol violation
ORA-17404	Received more RXDs than expected

Table D-6 (Cont.) TTC Messages Sorted in Alphabetic Order



ORA Number	Message
ORA-17442	Refcursor value is invalid
ORA-17425	Returning a stream in PLSQL block
ORA-17431	SQL Statement to parse is null
ORA-17415	This type hasn't been handled
ORA-17444	TTC Protocol version received from server not supported
ORA-17421	Types or Connection not defined
ORA-17405	UAC length is not zero
ORA-17423	Using a PLSQL block without an IOV defined
ORA-17427	Using Uninitialized OAC

 Table D-6
 (Cont.) TTC Messages Sorted in Alphabetic Order

E Troubleshooting

This appendix describes how to troubleshoot a Java Database Connectivity (JDBC) application in the following topics:

- Common Problems
- Basic Debugging Procedures

E.1 Common Problems

This section describes some common problems that you might encounter while using Oracle JDBC drivers. These problems include:

- Memory Consumption for CHAR Columns Defined as OUT or IN/OUT Variables
- Memory Leaks and Running Out of Cursors
- Opening More than 16 OCI Connections for a Process
- Using statement.cancel
- Using JDBC with Firewalls
- Frequent Abrupt Disconnection from Server
- Network Adapter Cannot Establish Connection

E.1.1 Memory Consumption for CHAR Columns Defined as OUT or IN/OUT Variables

In PL/SQL, when a CHAR or a VARCHAR2 column is defined as a OUT or IN/OUT variable, the driver allocates a CHAR array of 32512 chars. This can cause a memory consumption problem. JDBC Thin driver does not allocate memory when using VARCHAR2 output type. But JDBC OCI driver allocates memory for both CHAR and VARCHAR2 types. So, CPU load in OCI driver is higher than Thin driver.

At previous releases, the solution to the problem was to invoke the Statement.setMaxFieldSize method. A better solution is to useOracleCallableStatement.registerOutParameter. Oracle encourages you always to call registerOutParameter (int paramIndex, int sqlType, int scale, int maxLength) on each CHAR or VARCHAR2 column. This method is defined in

oracle.jdbc.OracleCallableStatement. Use the fourth argument, maxLength, to limit the memory consumption. This parameter tells the driver how many characters are necessary to store this column. The column is truncated if the character array cannot hold the column data. The third argument, scale, is ignored by the driver.

E.1.2 Memory Leaks and Running Out of Cursors

If you receive messages that you are running out of cursors or that you are running out of memory, make sure that all your Statement and ResultSet objects are explicitly closed.



Oracle JDBC drivers do not have finalizer methods. They perform cleanup routines by using the close method of the ResultSet and Statement classes. If you do not explicitly close your result set and statement objects, significant memory leaks can occur. You could also run out of cursors in the database. Closing a statement releases the corresponding cursor in the database.

Similarly, you must explicitly close Connection objects to avoid leaking and running out of cursors on the server-side. When you close the connection, the JDBC driver closes any open statement objects associated with it, thus releasing the cursor on the server-side.

E.1.3 Opening More than 16 OCI Connections for a Process

You may find that you are unable to open more than approximately 16 JDBC-OCI connections for a process at any given time. The most likely reasons for this would be either that the number of processes on the server exceeded the limit specified in the initialization file, or that the per-process file descriptors limit was exceeded. It is important to note that one JDBC-OCI connection can use more than one file descriptor (it might use anywhere between 3 and 4 file descriptors).

If the server allows more than 16 processes, then the problem could be with the perprocess file descriptor limit. The possible solution would be to increase this limit.

E.1.4 Using statement.cancel

The JDBC standard method <code>Statement.cancel</code> attempts to cleanly stop the execution of a SQL statement by sending a message to the database. In response, the database stops execution and replies with an error message. The Java thread that invoked <code>Statement.execute</code> waits on the server, and continues execution only when it receives the error reply message invoked by the call of the other thread to <code>Statement.cancel</code> method.

As a result, the Statement.cancel method relies on the correct functioning of the network and the database. If either the network connection is broken or the database server is hung, the client does not receive the error reply to the cancel message. Frequently, when the server process dies, JDBC receives an IOException that frees the thread that invoked Statement.execute. In some circumstances, the server is hung, but JDBC does not receive an IOException. The Statement.cancel method does not free the thread that initiated the Statement.execute method.



Note:

Remember the following points while working with the Statement.cancel method:

- Distinguish between Connection-level and Statement-level cancel. If a nonstatement execution, for example, a ROLLBACK is cancelled by a statement.cancel method, then we replay the command (only if it is ROLLBACK, COMMIT, autoCommit ON, autoCommit OFF, VERSION). To guarantee data integrity, we do not replay statement executions.
- Synchronize statement execution and statement cancel, so that the execution does not return until the cancel call is sent to the Database. This provides a better chance for the executing statement to be cancelled.
- Synchronize cancel calls, so that any new cancel request is ignored until the cancel in progress has completed the full protocol, that is, after the database receives an interrupt, act on it, and notify JDBC.

When JDBC does not receive an IOException, Oracle Net may eventually time out and close the connection. This causes an IOException and frees the thread. This process can take many minutes. For information about how to control this time-out, see the description of the readTimeout property for OracleDatasource.setConnectionProperties. You can also tune this time-out with certain Oracle Net settings.

The JDBC standard method <code>Statement.setQueryTimeout</code> relies on the <code>Statement.cancel</code> method. If execution continues longer than the specified time-out interval, then the monitor thread calls the <code>Statement.cancel</code> method. This is subject to all the same limitations described previously. As a result, there are cases when the time-out does not free the thread that invoked the <code>Statement.execute</code> method.

The length of time between execution and cancellation is not precise. This interval is no less than the specified time-out interval but can be several seconds longer. If the application has active threads running at high priority, then the interval can be arbitrarily longer. The monitor thread runs at high priority, but other high priority threads may keep it from running indefinitely. Note that the monitor thread is started only if there are statements executed with non zero time-out. There is only one monitor thread that monitors all Oracle JDBC statement execution.

Note:

The Statement.cancel method and the Statement.setQueryTimeout method are not supported in the server-side internal driver. The server-side internal driver runs in the single-threaded server process and the Oracle JVM implements Java threads within this single-threaded process. If the server-side internal driver is executing a SQL statement, then no Java thread can call the Statement.cancel method. This also applies to the Oracle JDBC monitor thread.



E.1.5 Using JDBC with Firewalls

Firewall timeout for idle-connections may sever a connection. This can cause JDBC applications to hang while waiting for a connection. You can perform one or more of the following actions to avoid connections from being severed due to firewall timeout:

- If you are using connection caching or connection pooling, then always set the inactivity timeout value on the connection cache to be shorter than the firewall idle timeout value.
- Pass oracle.jdbc.ReadTimeout as connection property to enable read timeout on socket. The timeout value is in milliseconds.
- For both JDBC OCI and JDBC Thin drivers, use net descriptor to connect to the database and specify the ENABLE=BROKEN parameter in the DESCRIPTION clause in the connect descriptor. Also, set a lower value for TCP KEEPALIVE INTERVAL.
- Enable Oracle Net DCD by setting SQLNET.EXPIRE_TIME=1 in the sqlnet.ora file on the server-side.

E.1.6 Frequent Abrupt Disconnection from Server

If the network is not reliable, then it is difficult for a client to detect the frequent disconnections when the server is abruptly disconnected. By default, a client running on Linux takes 7200 seconds (2 hours) to sense the abrupt disconnections. This value is equal to the value of the tcp_keepalive_time property. If you want your application to detect the disconnections faster, then you must set the value of the tcp_keepalive_time, tcp_keepalive_interval, and tcp_keepalive_probes properties to a lower value at the operating system level.

Note:

Setting a low value for the $tcp_keepalive_interval$ property leads to frequent probe packets on the network, which can make the system slower. So, the value of this property should be set appropriately based on the system requirements.

Also, you must specify the ENABLE=BROKEN parameter in the DESCRIPTION clause in the connection descriptor. For example:

jdbc:oracle:thin:@(DESCRIPTION=(ENABLE=BROKEN)(ADDRESS=(PROTOCOL=tcp)(PORT=5221) (HOST=myhost))(CONNECT DATA=(SERVICE NAME=orcl)))

E.1.7 Network Adapter Cannot Establish Connection

You may receive the following error while trying to establish a connection from a JDBC application to an Oracle instance:

java.sql.SQLException: Io exception: The Network Adapter could not establish connection

```
SQLException: SQLState (null) vendor code (17002)
```



This error may occur even if all or any of the following conditions is true:

- You are able to establish a SQL*Plus connection from the same client to the same Oracle instance.
- You are able to establish a JDBC OCI connection, but not a JDBC Thin connection from the same client to the same Oracle instance.
- The same JDBC application is able to connect from a different client to the same Oracle instance.
- The same behavior applies whether the initial JDBC connection string specifies a host name or an IP address.

One or more of the following reasons can cause this error:

- The host name to which you are trying to establish the connection is incorrect.
- The port number you are using to establish the connection is wrong.
- The NIC card supports both IPv4 and IPv6.
- The Oracle instance is configured for MTS, but the JDBC connection uses a shared server instead of a dedicated server.

You can quickly diagnose these above-mentioned reasons by using SQL*Plus, except for the issue with the NIC card. The following sections specify how to resolve this error, and also contains a sample application:

- Oracle Instance Configured with MTS Server Uses Shared Server
- JDBC Thin Driver with NIC Card Supporting Both IPv4 and IPv6
- Sample Application

E.1.7.1 Oracle Instance Configured with MTS Server Uses Shared Server

For resolving this error, you must verify whether the Oracle instance is configured for Multithreaded Server (MTS) or not. If the Oracle instance is not configured for MTS, then it must be configured.

If the Oracle instance is configured for MTS, then you must force the JDBC connection to use a dedicated server instead of a shared server. You can achieve this by reconfiguring the server to use dedicated connections only. If it is not feasible to configure your server to use only dedicated connections, then you perform the following steps to set it from the client side:

For JDBC OCI Client

- **1.** Add the (SERVER=DEDICATED) property to the TNS connection string stored in the tnsnames.ora file on the client.
- 2. Set the USER DEDICATED SERVER=ON in the sqlnet.ora file on the client.

For JDBC Thin:

You must specify a full name-value pair connection string (the same as it may appear in the tnsnames.ora file) instead of the short JDBC Thin syntax. For example, instead of the "jdbc:oracle:thin:@host:port:sid" connection string, you must use a connection string of the following form:

```
"jdbc:oracle:thin:@(DESCRIPTION=" +
"(ADDRESS_LIST=" +
"(ADDRESS=(PROTOCOL=TCP)" +
```



```
"(HOST=host)" +

"(PORT=port)" +

")" +

")" +

"(CONNECT_DATA=" +

"(SERVICE_NAME=sid)" +

"(SERVER=DEDICATED)" +

")" +
```

E.1.7.2 JDBC Thin Driver with NIC Card Supporting Both IPv4 and IPv6

If the Network Interface Controller (NIC) card of the server is configured to support both IPv4 and IPv6, then some services may start with IPv6. Any client application that tries to connect using IPv4 to the service that is running with IPv6 (or the other way round) receives a connection refused error. If a JDBC thin client application tries to connect to the Database server, then the application may stop responding or fail with the following error:

java.sql.SQLException: Io exception: The Network Adapter could not establish the connection Error Code: 17002

Use any of the following solutions to resolve this error:

Indicate the Java Virtual Machine (JVM) to use IP protocol version 4. Launch the JVM, where the JDBC application is running, with the –
 Djava.net.preferIPv4Stack parameter as true. For example, suppose you are running a JDBC application named jdbcTest. Then execute the application in the following way:

java -Djava.net.preferIPv4Stack=true jdbcTest

Use the OCI JDBC driver.

E.1.7.3 Sample Application

Example E-1 shows a basic JDBC program that connects to a Database and can be used to test your connection. It enables to try all forms of connection using Oracle JDBC drivers.

Example E-1 Basic JDBC Program to Connect to a Database in Five Different Ways

```
import java.sql.*;
public class Jdbctest
{
   public static void main (String args[])
   {
         try
         {
                   /* Uncomment the next line for more connection information */
                   // DriverManager.setLogStream(System.out);
                   /* Set the host, port, and sid below to match the entries in
the listener.ora */
                   String host = "myhost.oracle.com";
                   String port = "5221";
                   String sid = "orcl";
                   // or pass on command line arguments for all three
items
                   if ( args.length >= 3 )
```

```
{
                              host = args[0];
                              port = args[1];
                              sid = args[2];
                   }
                   String s1 = "jdbc:oracle:thin:@" + host + ":" + port + ":" + sid ;
                   if (args.length == 1)
                   {
                              s1 = "jdbc:oracle:oci8:@" + args[0];
                   }
                   if (args.length == 4)
                   {
                               s1 = "jdbc:oracle:" + args[3] + ":@" +
                                    "(description=(address=(host=" + host+ ")
(protocol=tcp) (port=" + port+ ")) (connect data=(sid="+ sid + ")))";
                   System.out.println( "Connecting with: " );
                   System.out.println( s1 );
                   DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
                   Connection conn =
DriverManager.getConnection( s1,"hr","hr");
                   DatabaseMetaData dmd = conn.getMetaData();
                   System.out.println("DriverVersion:["+dmd.getDriverVersion()+"]");
                   System.out.println("DriverMajorVersion:
["+dmd.getDriverMajorVersion()+"]");
                   System.out.println("DriverMinorVersion:
["+dmd.getDriverMinorVersion()+"]");
                   System.out.println("DriverName:["+dmd.getDriverName()+"]");
                   if ( conn!=null )
                      conn.close();
                   System.out.println("Done.");
           }
           catch ( SQLException e )
           {
                   System.out.println ("\n*** Java Stack Trace ***\n");
                   e.printStackTrace();
                   System.out.println ("\n*** SQLException caught ***\n");
                   while ( e != null )
                   {
                               System.out.println ("SQLState: " + e.getSQLState
());
                               System.out.println ("Message: " + e.getMessage
());
                               System.out.println ("Error Code: " + e.getErrorCode
());
                               e = e.getNextException ();
                               System.out.println ("");
                    }
             }
 }
}
```

E.2 Basic Debugging Procedures

This section describes strategies for debugging a JDBC program:

- Oracle Net Tracing to Trap Network Events
- Third Party Debugging Tools



Related Topics

About Processing SQL Exceptions

E.2.1 Oracle Net Tracing to Trap Network Events

You can enable client and server Oracle-Net trace to trap the packets sent over Oracle Net. You can use client-side tracing only for the JDBC OCI driver; it is not supported for the JDBC Thin driver.

The trace facility produces a detailed sequence of statements that describe network events as they execute. "Tracing" an operation lets you obtain more information about the internal operations of the event. This information is printed to a readable file that identifies the events that led to the error. Several Oracle Net parameters in the SQLNET.ORA file control the gathering of trace information. After setting the parameters in SQLNET.ORA, you must make a new connection for tracing to be performed.

The higher the trace level, the more detail is captured in the trace file. Because the trace file can be hard to understand, start with a trace level of 4 when enabling tracing. The first part of the trace file contains connection handshake information, so look beyond this for the SQL statements and error messages related to your JDBC program.



The trace facility uses a large amount of disk space and might have significant impact upon system performance. Therefore, enable tracing only when necessary.

Related Topics

Oracle Call Interface Programmer's Guide

E.2.1.1 Client-Side Tracing

Set the following parameters in the SQLNET.ORA file on the client system.

Note:

Starting from Oracle Database 12c Release 1 (12.1), Oracle recommends you to use the configuration parameters present in the new XML configuration file oraaccess.xml instead of the OCI-specific configuration parameters present in the sqlnet.ora file. However, the configuration parameters present in the sqlnet.ora file are still supported.

E.2.1.1.1 TRACE_LEVEL_CLIENT

Purpose:

Turns tracing on or off to a certain specified level.



Default Value:

0 or OFF

Available Values:

- 0 or OFF No trace output
- 4 or USER User trace information
- 10 or ADMIN Administration trace information
- 16 or SUPPORT WorldWide Customer Support trace information

Example:

TRACE_LEVEL_CLIENT=10

E.2.1.1.2 TRACE_DIRECTORY_CLIENT

Purpose:

Specifies the destination directory of the trace file.

Default Value:

ORACLE_HOME/network/trace

Example:

UNIX: TRACE DIRECTORY CLIENT=/oracle/traces

Windows: TRACE_DIRECTORY_CLIENT=C:\ORACLE\TRACES

E.2.1.1.3 TRACE_FILE_CLIENT

Purpose:

Specifies the name of the client trace file.

Default Value:

SQLNET.TRC

Example:

TRACE_FILE_CLIENT=cli_Connection1.trc

Note:

Ensure that the name you choose for the TRACE_FILE_CLIENT file is different from the name you choose for the TRACE_FILE_SERVER file.

E.2.1.1.4 TRACE_UNIQUE_CLIENT

Purpose:

Gives each client-side trace a unique name to prevent each trace file from being overwritten with the next occurrence of a client trace. The PID is attached to the end of the file name.



Default Value:

OFF

Example:

TRACE UNIQUE CLIENT = ON

E.2.1.2 Server-Side Tracing

Set the following parameters in the SQLNET.ORA file on the server system. Each connection will generate a separate file with a unique file name.

Note:

Starting from Oracle Database 12c Release 1 (12.1), Oracle recommends you to use the configuration parameters present in the new XML configuration file oraaccess.xml instead of the OCI-specific configuration parameters present in the sqlnet.ora file. However, the configuration parameters present in the sqlnet.ora file are still supported.

E.2.1.2.1 TRACE_LEVEL_SERVER

Purpose:

Turns tracing on or off to a certain specified level.

Default Value:

 $0 \mbox{ or } \mbox{OFF}$

Available Values:

- **0** or OFF No trace output
- 4 or USER User trace information
- 10 or ADMIN Administration trace information
- 16 or SUPPORT WorldWide Customer Support trace information

Example:

TRACE_LEVEL_SERVER=10

E.2.1.2.2 TRACE_DIRECTORY_SERVER

Purpose:

Specifies the destination directory of the trace file.

Default Value:

ORACLE_HOME/network/trace

Example:

TRACE_DIRECTORY_SERVER=/oracle/traces



E.2.1.2.3 TRACE_FILE_SERVER

Purpose:

Specifies the name of the server trace file.

Default Value:

SERVER.TRC

Example:

TRACE FILE SERVER= svr Connection1.trc

Note:

Ensure that the name you choose for the TRACE_FILE_SERVER file is different from the name you choose for the TRACE_FILE_CLIENT file.

E.2.2 Third Party Debugging Tools

You can use tools such as JDBCSpy and JDBCTest from Intersolv to troubleshoot at the JDBC API level. These tools are similar to ODBC Spy and ODBC Test tools.



Index

A

AC, 32-1 Accessing PL/SQL Associative Arrays, 4-5 annotations, 2-16 ANYDATA, 4-21 ANYTYPE, 4-21 application continuity, 32-1 configuring Oracle database, 32-6 delaying the reconnection, 32-17 disabling replay, 32-22 identifying request boundaries, 32-9 registering a connection callback, 32-12 retaining mutable values, 32-19 ARRAY objects, creating, 17-7 arrays defined, 17-1 getting, 17-11 named, 17-1 passing to callable statement, 17-13 retrieving from a result set, 17-8 retrieving partial arrays, 17-10 using type maps, 17-14 working with, 17-1 authentication (security), 9-13 auto-commit, 2-13 auto-commit mode disabling, C-2 result set behavior, C-2

В

batch jobs, authenticating users in, 9-40 batch updates--see update batching, 22-1 Bequeath Protocol, 2-18 BFILE class, 4-7 defined, 13-8 BFILE locator, selecting, 4-8 BLOB class, 4-7 locators selecting, 4-8 branch qualifier (distributed transactions), 36-13

С

CachedRowSet, 19-6 caching, client-side Oracle use for scrollable result sets, 18-1 callable statement using getOracleObject() method, 12-8 cancelling SOL statements, E-2 casting return values, 12-12 catalog arguments (DatabaseMetaData), A-16 CHAR columns using setFixedCHAR() to match in WHERE, 12-15 character sets. 4-16 CLOB class, 4-7 locators, selecting, 4-8 close method, 21-12 close() method, E-1 collections defined, 17-1 collections (nested tables and arrays), 17-6 column types defining, 22-11 redefining, 22-9 commit a distributed transaction branch, 36-8 commit changes to database, 2-13 connection closing, 2-15 opening, 2-8 connection properties, 8-7 put() method, 8-10 connections read-only, C-6 constants for SQL types, 4-28 CursorName limitations, A-15 cursors, E-1 custom collection classes defined, 17-2 custom Java classes, 4-3 defined, 14-1 custom object classes creating, 14-6

custom object classes *(continued)* defined, *14-1* custom reference classes defined, *16-1*

D

data conversions, 12-4 LONG, 13-3 LONG RAW, 13-3 data sources creating and connecting (with JNDI), 8-5 creating and connecting (without JNDI), 8-5 Oracle implementation, 8-2 properties, 8-2 standard interface, 8-2 data streaming avoiding, 13-6 data type mappings, 12-1 data types Java, 12-1 Java native, 12-1 JDBC, 12-1 Oracle SQL, 12-1 database connecting with server-side internal driver, 7-1 connection testing, 2-5 **Database Resident Connection Pooling** DRCP, 27-1 database specifiers, 8-11 database URL including userid and password, 2-8 database URL, specifying, 2-8 database URLs and database specifiers, 8-11 DatabaseMetaData calls, A-16 datasources. 8-1 and JNDI. 8-5 DATE class, 4-8 debugging JDBC programs, E-7 defaultConnection() method, 7-1 distributed transaction ID component, 36-13 distributed transactions branch qualifier. 36-13 check for same resource manager, 36-8 commit a transaction branch, 36-8 components and scenarios, 36-2 concepts, 36-2 distributed transaction ID component, 36-13 end a transaction branch. 36-8 example of implementation, 36-16 forget, 36-8 global transaction identifier, 36-13 ID format identifier, 36-13

distributed transactions (continued) obtain the list of transaction brances during recoverv. 36-8 Oracle XA connection implementation, 36-7 Oracle XA data source implementation, 36-6 Oracle XA ID implementation, 36-13 Oracle XA optimizations, 36-16 Oracle XA resource implementation. 36-8 overview, 36-1 prepare a transaction branch. 36-8 roll back a transaction branch, 36-8 start a transaction branch, 36-8 transaction branch ID component, 36-13 XA connection interface. 36-7 XA data source interface. 36-6 XA error handling, 36-16 XA exception classes, 36-14 XA ID interface, 36-13 XA resource functionality. 36-8 XA resource interface, 36-8 DML Returning, 4-4, 4-32 example, 4-33 limitations, 4-34 Oracle-specific APIs, 4-32 running statements, 4-33 Double.NaN restrictions on use, 4-8

Е

end a distributed transaction branch, 36-8 Enterprise Java Beans (EJB), 19-8 errors general JDBC message structure, D-1 general JDBC messages, listed, D-1 processing exceptions, 2-30 TTC messages, listed, D-13 explicit Statement caching definition of. 21-3 extensions to JDBC, Oracle, 4-1, 12-1, 14-1, 16-1, 17-1, 22-1 external changes (result set) defined, 18-6 visibility vs. detection, 18-7 external file defined, 13-8

F

fetch direction in result sets, 18-5 fetch size, result sets, 18-4 FilteredRowSet, 19-12 finalizer methods, *E*-1 Firewalls, using with JDBC, *E*-4 Float.NaN restrictions on use, 4-8 floating-point compliance, A-16 format identifier, transaction ID, 36-13 function call syntax, JDBC escape syntax, A-14

G

getBinaryStream() method, 13-4 getBytes() method, 13-5 getColumns, 2-19 getConnection() method, 7-1 getCursorName() method limitations, A-15 getLogicalTransactionId method, 31-3 getMoreResultSet(int), 2-22 getObject() method for ORAData objects, 14-13 return types, 12-8 getOracleObject() method return types, 12-7, 12-8 using in callable statement, 12-8 using in result set, 12-8 getStatementCacheSize() method code example, 21-6 getXXX() methods casting return values, 12-12 for specific data types, 12-10 global transaction identifier (distributed transactions), 36-13 global transactions, 36-1 globalization, 20-1 using, 20-1

I

IEEE 754 floating-point compliance, A-16 implicit Statement caching definition of, 21-2 Least Recently Used (LRU) algorithm, 21-3 internal changes (result set) defined, 18-6 isColumnInvisible, 2-19 isSameRM() (distributed transactions), 36-8

J

Java compiling and running, 2-4 data types, 12-1 native data types, 12-1 stored procedures, 2-30 stream data, 13-1 Java Naming and Directory Interface (JNDI), 8-1 Java Sockets, 1-1 Java Virtual Machine (JVM), 7-1 java.sql.Connection interface close method. 21-12 java.sgl.Statement interface close method, 21-12 java.util.Properties, 26-5 JDBC and IDEs. 1-7 basic program, 2-7 data types, 12-1 importing packages, 2-7 limitations of Oracle extensions, A-15 sample files, 2-4 testing. 2-5 JDBC drivers choosing a driver for your needs, 1-3 common problems, E-1 introduction, 1-1 JDBC escape syntax. A-9 JDBC escape syntax, A-9 function call syntax, A-14 LIKE escape characters, A-13 outer joins, A-14 scalar functions, A-12 time and date literals, A-10 translating to SQL example, A-14 JDBC PIPELINING, 25-1 JdbcCheckup program, 2-5 JDBCRowSet, 19-9 JDBCSpv. E-11 JDBCTest, E-11 JDeveloper, 1-7 JNDI and datasources, 8-5 looking up data source, 8-5 overview of Oracle support, 8-1 registering data source, 8-5 JoinRowSet, 19-14 JVM, 7-1

Κ

Kerberos Authentication, 9-33, 9-34 Kerberos Authentication Enhancements, 9-33 KPRB driver overview, 1-2 relation to the SQL engine, 7-1 session context, 7-3 testing, 7-4 transaction context, 7-3 URL for, 7-3

L

Least Recently Used (LRU) algorithm, 21-3, 26-6

LIKE escape characters, JDBC escape syntax, *A-13* limitations on setBytes() and setString(), use of streams to avoid, *13-11* LOB defined, *13-7* LONG data conversions, *13-3* LONG RAW data conversions, *13-3* LRU algorithm, *21-3*

Μ

memory leaks, *E-1* Multiple Pool Support, 27-4

Ν

named arrays, 17-1 defined, 17-6 nativeXA, 8-4 network events, trapping, E-8 NLS. See globalization, 20-1 NULL testing for, 12-6 NUMBER class, 4-8

0

object references accessing object values, 16-3, 16-4 described, 16-1 passing to prepared statements, 16-3 retrieving, 16-2 retrieving from callable statement, 16-3 updating object values, 16-3, 16-4 OCI driver described, 1-2 ODBCSpy, E-11 ODBCTest, E-11 optimization, performance, C-1 Oracle Advanced Security support by JDBC, 9-11 Oracle data types using, 12-1 Oracle extensions, 4-1 data type support, 4-2 limitations, A-15 catalog arguments to DatabaseMetaData calls, A-16 CursorName, A-15 IEEE 754 floating-point compliance, A-16 JDBC outer join escapes, A-15

Oracle extensions (continued) limitations (continued) read-only connection, C-6 SQLWarning class, A-16 object support, 4-3 result sets, 12-6 statements, 12-6 to JDBC, 4-1, 12-1, 14-1, 16-1, 17-1, 22-1 Oracle objects and JDBC. 14-1 Java classes which support, 14-3 mapping to custom object classes, 14-6 reading data by using SQLData interface, 14-10 working with, 14-1 writing data by using SQLData interface, 14-11 Oracle SQL data types, 12-1 oracle.idbc.. Oracle JDBC extensions. 2-8 oracle.jdbc.LogicalTransactionIdEventListener interface. 31-4 oracle.jdbc.OracleCallableStatement interface, 4-27 oracle.jdbc.OracleConnection interface, 4-25 oracle.jdbc.OraclePreparedStatement interface, 4-26 oracle.jdbc.OracleResultSet interface, 4-27 oracle.jdbc.OracleResultSetMetaData interface, 4-28 oracle.idbc.OracleSgl class. A-14 oracle.jdbc.OracleStatement interface, 4-26 oracle.jdbc.OracleTypes class, 4-28 oracle.jdbc.xa package and subpackages, 36-5 oracle.sql.ARRAY class methods for Java primitive types, 17-5 oracle.sgl.BFILE class, 4-7 oracle.sgl.BLOB class, 4-7 oracle.sgl.CLOB class, 4-7 oracle.sql.data types support, 4-6 oracle.sgl.DATE class, 4-8 oracle.sgl.NUMBER class, 4-8 oracle.sql.RAW class, 4-8 OracleCallableStatement interface, 4-27 OracleCallableStatement object, 21-2, 21-3 OracleConnection class, 4-25 OracleData interface advantages, 14-6 OracleDataSource class, 8-2 OraclePreparedStatement interface, 4-26 OraclePreparedStatement object, 21-2, 21-3 OracleResultSet interface, 4-27 OracleResultSetMetaData interface. 4-28 OracleStatement interface, 4-26 OracleTypes class, 4-28



OracleXAConnection class, 36-7 OracleXADataSource class, 36-6 OracleXAResource class, 36-8 OracleXid class, 36-13 ORAData interface additional uses, 14-16 reading data, 14-14 writing data, 14-15 orai18n.jar file, 20-2 outer joins, JDBC escape syntax, A-14

Ρ

password, specifying, 2-8 PDA, 19-8 performance extensions defining column types, 22-11 performance optimization, C-1 Personal Digital Assistant (PDA), 19-8 pipelining, 25-1 pipelining with Reactive Extensions, 25-2 PL/SQL stored procedures. 2-29 PL/SQL Associative Arrays, 4-35 prefetching rows, 22-9 suggested default, 22-9 prepare a distributed transaction branch, 36-8 put() method for Properties object, 8-10

R

RAW class. 4-8 recover (distributed transactions), 36-8 REF CURSORs, 4-18 refetching rows into a result set, 18-5 registerConnectionInitializationCallback, 32-14 Remote Method Invocation (RMI), 19-8 resource managers, 36-2 result set auto-commit mode, C-2 metadata, 4-28 Oracle extensions, 12-6 using getOracleObject() method, 12-8 result set enhancements downgrade rules, 18-2 fetch size, 18-4 limitations, 18-2 Oracle scrollability requirements, 18-1 Oracle updatability requirements, 18-1 refetching rows, 18-5 summary of visibility of changes, 18-7 visibility vs. detection of external changes, 18-7 result set fetch size, 18-4

result set object closing, 2-10 result set, processing, 2-10 return types for getXXX() methods, 12-11 getObject() method, 12-8 getOracleObject() method, 12-8 return values casting, 12-12 RMI. 19-8 roll back a distributed transaction branch, 36-8 roll back changes to database, 2-13 row prefetching and data streams. 13-11 **ROWID** class defined. 4-17 ROWID, use for result set updates, 18-1 **RowSet** events and event listeners. 19-3 overview. 19-1 properties, 19-2 traversing, 19-4 RSI Modes, 24-3

S

scalar functions, JDBC escape syntax, A-12 SCAN backward compatibility, 35-4 configuring the database, 35-2 connection load balancing, 35-2 maximum availability architecture environment, 35-5 Oracle connection manager. 35-6 overview, 35-1 version. 35-4 Schema Naming, 4-4 scripts, authenticating users in, 9-40 scroll-sensitive result sets limitations, 18-2 scrollable result sets fetch direction, 18-5 implementation of scroll-sensitivity, 18-8 refetching rows, 18-5 visibility vs. detection of external changes, 18-7 security authentication, 9-13 Oracle Advanced Security support, 9-11 server-side internal driver connection to database, 7-1 server-side Thin driver, overview, 1-2 session context for KPRB driver, 7-3

setBytes() limitations, using streams to avoid, 13-11 setCursorName() method, A-15 setDisableStmtCaching() method, 21-7 setEscapeProcessing() method, A-10 setFixedCHAR() method, 12-15 setNull(), 12-6 setObeict() method. 12-12 setObject() method for STRUCT objects, 14-4 setOracleObject() method, 12-12 setString() limitations, using streams to avoid, 13-11 setXXX() methods, for specific data types, 12-13 Solaris shared libraries, 36-22 specifiers database, 8-11 SOL data converting to Java data types, 12-4 types, constants for, 4-28 SQL engine relation to the KPRB driver, 7-1 SQL syntax (Oracle), A-10 SQLData interface advantages, 14-6 reading data from Oracle objects, 14-10 writing data from Oracle objects, 14-11 SQLWarning class, limitations, A-16 start a distributed transaction branch, 36-8 Statement caching explicit definition of, 21-3 implicit definition of, 21-2 Least Recently Used (LRU) algorithm, 21-3 Statement object closing, 2-10 statement.cancel(), E-2 statements Oracle extensions, 12-6 stopping statement execution, E-2 stored procedures Java, 2-30 PL/SQL, 2-29 stream data, 13-1 CHAR columns, 13-7 closing, 13-10 example, 13-4 external files. 13-7 LOBs, 13-7 LONG columns, 13-2 LONG RAW columns, 13-2

stream data (continued) multiple columns, 13-8 precautions, 13-10 RAW columns, 13-7 row prefetching, 13-11 use to avoid setBytes() and setString() limitations, 13-11 VARCHAR columns, 13-7 stream data column bypassing, 13-9 STRUCT object retrieving, 14-3 retrieving attributes as oracle.sql types, 14-3 SYS.ANYDATA, 4-21 SYS.ANYTYPE, 4-21

Т

TAF, definition of, 34-1 TCP/IP protocol, 8-14 testing for NULL values, 12-6 Thin driver overview. 1-1 server-side, overview, 1-2 time and date literals, JDBC escape syntax, A-10 trace facility, E-8 trace parameters client-side. E-8 server-side, E-10 transaction branch, 36-1 transaction branch ID component, 36-13 transaction context for KPRB driver. 7-3 transaction IDs (distributed transactions), 36-3 transaction managers, 36-2 transactions switching between local and global, 36-4 Transparent Application Failover (TAF), definition of. 34-1 True Cache support, 2-17 TTC error messages, listed, D-13 type map, 12-7 adding entries, 14-8 and STRUCTs, 14-9 creating a new map, 14-9 used with arrays, 17-10 using with arrays, 17-14 type map (SQL to Java), 14-6 type maps relationship to database connection, 7-3

U

unicode data, 4-13

unregisterConnectionInitializationCallback method, 32-14 updatable result sets limitations, 18-2 refetching rows, 18-5 update conflicts, 18-4 update batching, 22-1 update batching (standard model) adding to batch, 22-2 clearing the batch, 22-4 committing changes, 22-4 error handling, 22-6 example, 22-5 executing the batch, 22-3 intermixing batched and non-batched, 22-7 overview, 22-2 update counts, 22-5 update counts upon error, 22-6 update conflicts in result sets, 18-4 update counts standard update batching, 22-5 upon error (standard batching), 22-6 URLs for KPRB driver, 7-3

URLs *(continued)* userid, specifying, 2-8

W

WebRowSet, 19-10 window, scroll-sensitive result sets, 18-8

Х

XA

connection implementation, 36-7 connections (definition), 36-3 data source implementation, 36-6 data sources (definition), 36-2 definition, 36-2 error handling, 36-16 example of implementation, 36-16 exception classes, 36-14 Oracle optimizations, 36-16 Oracle transaction ID implementation, 36-13 resource implementation, 36-8 resources (definition), 36-3 transaction ID interface, 36-13