Oracle® Database

SQL Translation and Migration Guide

12*c* Release 2 (12.2)

E85805-01

October 2017

ORACLE®

Oracle Database SQL Translation and Migration Guide, 12*c* Release 2 (12.2)

E85805-01

Primary Author: Tanmay Choudhury

Contributors: Tulika Das, Peter Castro, Christopher Jones, Shoaib Lari, Tom Laszewski, Aman Manglik, Robert Pang, Rajendra Pingte, Jeff D. Smith, Andrei Souleimanian

Contents

Preface

1   Introduction to Tools and Products that Support Migration

2　SQL Translation Framework Overview

3　SQL Translation Framework Configuration

4　SQL Translation of JDBC and ODBC Applications

5   Example: Application Migration Using SQL Translation Framework

6   MySQL Client Library Driver for Oracle

7   API Reference for Oracle MySQL Client Library Driver

8   API Reference for SQL Translation of JDBC Applications

Glossary

Index

List of Tables

Preface

This guide describes the installation, configuration, and administration tasks for all activities related to migrating applications developed for non-Oracle databases, such as DB2, MySQL, Sybase, and legacy applications, to Oracle Database. This guide also provides migration scenarios that users may implement in sequence.

Audience

This guide is for database administrators and application developers who are interested in migrating from databases other than Oracle to an Oracle Database.

Related Documents

For more information, see the following documents in the Oracle Database Release 12*c* documentation set:

- *Oracle Database SQL Language Reference*
- *Oracle Database Administrator's Guide*
- *Oracle Database Development Guide*
- *Oracle Database Reference*

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.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.

Convention	Meaning
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Introduction to Tools and Products that Support Migration

Before migrating your application to Oracle Database, you must be aware of several key points that are described in *Oracle Database Concepts*.

When discussing the migration of a database-centered enterprise, it is useful to keep in mind that the actual migration of database schema and data is only a part of the process. The migration of a core business solution often involves several databases and applications that work together to deliver the product and services that drive the revenue of an organization. For more information about preparing a migration plan, see *Oracle SQL Developer User's Guide*.

1.1 Oracle Database Features for Migration Support

Oracle Database 12*c* introduced a large set of features that collectively enhance the migration process of non-Oracle database applications to Oracle Database.

1.1.1 SQL Translation Framework

A key part of migrating non-Oracle databases to Oracle Database involves the conversion of non-Oracle SQL statements to SQL statements that are acceptable to Oracle Database. The conversion of the non-Oracle SQL statements of the applications is a manual and tedious process. To minimize the effort, or to eliminate the necessity for converting these statements, Oracle Database 12*c* introduces a new feature called SQL Translation Framework. SQL Translation Framework receives these SQL statements from client applications, and then translates them at run-time.

The SQL Translation Profile registers the SQL Translater inside the database so it can handle the SQL translation for non-Oracle client application. If an error occurs while a SQL statement is executed, then the SQL Translator can translate the Oracle error code and the `ANSI SQLSTATE` into the vendor-specific values expected by the application. The translated statements are then saved in the SQL Translation Profile, to be examined and edited at the user's discretion.

The advantages of SQL Translation Framework follow:

- The translation of SQL statements, Oracle error codes, and `ANSI SQLSTATE` is automatic.
- The translations are centralized and examinable.
- The user has the option to extract translations and insert them back into the application.

1.1.2 Support for MySQL Applications

Oracle Database driver for MySQL eases migration of applications initially developed to work with MySQL database. This feature has two key benefits:

- It enables the enterprise to reuse the same application to use data stored in both MySQL Database and Oracle Database

- It reduces the cost and complexity of migrating MySQL applications to Oracle Database

Oracle Database supports all MySQL functions in the client interface with the same semantics.

1.1.2.1 Restrictions on SQL Statement Translation

SQL Translation has the following limitations when translating SQL statements:

- SQL Translation ignores the following SQL constructs:

  - The `ENGINE` specification for a table is not used; there is only one storage engine, namely `Oracle`.

  - The `ENUM` and `SET` types are used as `VARCHAR2`. These values are not converted to their index value if they are retrieved in a numeric context.

- SQL Translation generates an error when attempting to handle the following SQL constructs; the application must be recoded.

  - Oracle does not support spatial datatypes, such as `GEOMETRY`, `POINT`, `LINESTRING`, `POLYGON`, `GEOMETRYCOLLECTION`, `MULTILINESTRING`, `MULTIPOINT`, and `MULTIPOLYGON`.

    Oracle does not support MySQL-specific NLS commands.

- The following SQL commands give Oracle-specific output or have Oracle-specific effect:

  - `SHOW DATABASES` shows only one database, namely `oracle`.

  - `SHOW ENGINES` shows the `Oracle` engine only.

  - `CREATE PROCEDURE` must follow Oracle PL/SQL specification in Oracle Database 12*c*.

- The following data types have different behavior In Oracle Database than what is expected in the native database:

  - Columns of `ENUM` data types are created as `VARCHAR2(4000)`. No validation is performed for insertion.

  - Columns of `SET` data types are created as `VARCHAR2(64)`. No validation is performed for insertion.

For further details, see MySQL Client Library Driver for Oracle and API Reference for Oracle MySQL Client Library Driver .

1.1.3 Support for Identity Columns

Oracle Database 12*c* implements ANSI-compliant `IDENTITY` columns. Migration from database systems that use identity columns is simplified and can take advantage of this new functionality.

This feature implements auto increment by enhancing `DEFAULT` or `DEFAULT ON NULL` semantics for use by `SEQUENCE.NEXTVAL` and `SYS_GUID`, supports built-in functions and implicit return of default values.

1.1.3.1 Creating Identity Columns

Example 1-1 creates a table with an identity column, which is generated by default. When explicit `null`s are inserted into the identity column, the sequence generator creates values by default. For further details, see *Oracle Database SQL Language Reference*.

Example 1-1   How to create an identity column

```
CREATE TABLE t1 (c1 NUMBER GENERATED BY DEFAULT ON NULL AS IDENTITY,
                 c2 VARCHAR2(10));
INSERT INTO t1(c2) VALUES ('abc');
INSERT INTO t1 (c1, c2) VALUES (null, 'xyz');
SELECT c1, c2 FROM t1;
```

1.1.4 Implicit Statement Results

Starting with Oracle Database 12*c* Release 2 (12.2), Oracle implicitly returns to the client application the results of SQL statements executed within a stored procedure, bypassing the explicit use `REF CURSOR`s. This feature eliminates the overhead of re-writing the client-side code.

Implicit statement results enable the user to write a stored procedure, where each intended query (the statement after the `FOR` keyword) is part of the `OPEN` cursor variable. When code is migrated to Oracle Database from other vendors environments, the PL/SQL layer adds the equivalent capability and enables `SELECT` statements to pass the results to the client. The stored procedures can then return the results directly to the client with the `DBMS_SQL.RETURN_RESULT` procedure. The SQL*Plus `FORMAT` command and its variations may be invoked to customize the output.

For information about the `DBMS_SQL` package, see *Oracle Database PL/SQL Packages and Types Reference*. For information about how to use format output, *SQL*Plus User's Guide and Reference*.

1.1.4.1 JDBC Support for Implicit Results

Starting with Oracle Database 12*c* Release 2 (12.2), JDBC applications provide support for implicit results through the following new functions:

- `getMoreResults`
- `getMoreResults(int)`
- `getResultSet`

You can use these methods to retrieve and process the implicit results returned by PL/SQL procedures or blocks, as demonstrated in Example 1-2.

For more information, see *Oracle Database JDBC Developer's Guide*

1.1.4.1.1 Processing Implicit Results in JDBC

Example 1-2   Retrieving and Processing Implicit Results from PL/SQL Blocks

Suppose you have a procedure called `foo`:

```
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 demonstrates how to retrieve the implicit results returned by
PL/SQL procedures using the JDBC `getMoreResults` methods:

```java
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 */
            }
         }
       }
```

1.1.4.2 OCI Support for Implicit Results

Starting with Oracle Database 12*c* Release 2 (12.2), Oracle Call Interface (OCI)
provides support for implicit results through a new function, `OCIStmtGetNextResult()`. It
is called iteratively by C applications to retrieve each implicit result from stored
procedures and anonymous blocks. Implicit results consume rows directly from a
stored procedure without going through a `RefCursor`.

> ✎ **See Also:**
>
> *Oracle Call Interface Programmer's Guide*

1.1.4.2.1 Processing Implicit Results in OCI

Example 1-3 shows how to use the `OCIStmtGetNextResult()` function to retrieve and
process the implicit results returned by either a PL/SQL stored procedure or an
anonymous block:

Example 1-3   Using OCIStmtGetNextResult() to Process Implicit Results

```c
OCIStmt *stmthp;
  ub4      rsetcnt;
  void    *result;
  ub4      rtype;
  char    *sql = "begin foo; end;";
```

```
OCIHandleAlloc((void *)envhp, (void **)&stmthp,
               OCI_HTYPE_STMT, 0, (void **)0);

/* Prepare and execute the PL/SQL procedure. */
OCIStmtPrepare(stmthp, errhp, (oratext *)sql, strlen(sql),
               OCI_NTV_SYNTAX, OCI_DEFAULT);
OCIStmtExecute(svchp, stmthp, errhp, 1, 0,
               (const OCISnapshot *)0,
               (OCISnapshot *)0, OCI_DEFAULT);

/* Now check if any implicit results are available. */
OCIAttrGet((void *)stmthp, OCI_HTYPE_STMT, &rsetcnt, 0,
           OCI_ATTR_IMPLICIT_RESULT_COUNT, errhp);

/* Loop and retrieve the implicit result-sets.
 * ResultSets are returned in the same order as in the PL/SQL
 * procedure/block.
 */
while (OCIStmtGetNextResult(stmthp, errhp, &result, &rtype,
                           OCI_DEFAULT) == OCI_SUCCESS)
{   /* Check the type of implicit ResultSet, currently
    * only supported type is OCI_RESULT_TYPE_SELECT
*/   if (rtype == OCI_RESULT_TYPE_SELECT)
{     OCIStmt *rsethp = (OCIStmt *)result;
      /* Perform normal OCI actions to define and fetch rows. */
   }    else
     printf("unknown result type %d\n", rtype);
    /* The result set handle should not be freed by the user. */
 }   OCIHandleFree(stmthp, OCI_HTYPE_STMT); /* All implicit result-sets are also
freed. */
```

1.1.4.3 ODBC Support for Implicit Results

Starting with Oracle Database 12*c*, ODBC applications provide support for implicit results through a new function, `SQLMoreResults()`. ODBC driver is enhanced to make use of the following new OCI APIs that enhance the migration process:

- `OCIStmtGetNextResult()` function

- `OCI_ATTR_IMPLICIT_RESULT_COUNT` attribute

- `OCI_RESULT_TYPE_SELECT` attribute

ODBC support for implicit results enables the migration of Sybase and SQL Server applications that use multiple result sets bundled in the stored procedures. Oracle achieves this by sending the statements or procedures to the server, where the non-Oracle SQL is translated to Oracle syntax.

1.1.4.3.1 Processing Implicit Results in ODBC

Example 1-4 and Example 1-5 demonstrate how to retrieve implicit results in ODBC.

Example 1-4   Using ODBC to return implicit results with DBMS_SQL.RETURN_RESULT

```
create or replace procedure foo
is
c1 sys_refcursor;
c2 sys_refcursor;
begin
```

```
  open c1 for select employee_id, first_name from employees where employee_id=7369;
  dbms_sql.return_result(c1);
  open c2 for select department_id, department_name from departments where rownum
<=2;
  dbms_sql.return_result(c2);
end;
/
```

Example 1-5   Using ODBC to return implicit results with SQLMoreResults

```
SQLLEN enind,jind;
SQLUINTEGER eno = 0;
SQLCHAR empname[STR_LEN] = "";
//Allocate HENV, HDBC, HSTMT handles
rc = SQLPrepare(hstmt, "begin foo(); end;", SQL_NTS);
rc = SQLExecute(hstmt);
//Bind columns for the first SELECT query in the procedure foo( )
rc = SQLBindCol (hstmt, 1, SQL_C_ULONG, &eno, 0, &jind);
rc = SQLBindCol (hstmt, 2, SQL_C_CHAR, empname, sizeof (empname),
&enind);
…
//so on for all the columns that needs to be fetched as per the SELECT
//query in the procedure.
//Fetch all results for first SELECT query
while ((rc = SQLFetch (hstmt)) != SQL_NO_DATA)
{
//do something
}
//Again check if there are any results available by calling
//SQLMoreResults. SQLMoreResults will return SQL_SUCCESS if any
//results are available else returns errors appropriately as explained
//in MSDN ODBC spec.
rc = SQLMoreResults ( hstmt );
if( rc == SQL_SUCCESS)
{
//If the columns for the second SELECT query are different the rebind
//the columns for the second SELECT SQL statement.
rc = SQLBindCol (hstmt, 1,…);
rc = SQLBindCol (hstmt, 2,…);
…
//Fetch the second result set
while ((rc = SQLFetch (hstmt)) != SQL_NO_DATA)
//do something
}
SQLFreeStmt(hstmt,SQL_DROP);
SQLDisconnect (hdbc);

SQLFreeConnect (hdbc);
SQLFreeEnv (henv);
```

1.1.5 Enhanced SQL to PL/SQL Bind Handling

In earlier releases of Oracle Database, a SQL expression could not invoke a PL/SQL function that had a formal parameter or return type that was not a SQL data type.

Starting with Oracle Database 12*c*, a PL/SQL anonymous block, a SQL CALL statement, or a SQL query can invoke a PL/SQL function that has parameters of the following types:

- Boolean

- Record declared in a package specification
- Collection declared in a package specification

The SQL TABLE operator is also enhanced, so that you can query on PL/SQL collections of locally scoped types as an argument to TABLE operator. Here, the collections can be of nested table types, VARRAY, or PL/SQL index table that are indexed by PLS_INTEGER.

This feature extends the flexibility of the TABLE operator, and enables easy migration of non-Oracle stored procedure code to PL/SQL.

1.1.5.1 Invoking a Subprogram with a Nested Table Parameter

Example 1-6 shows how to dynamically call a subprogram with a nested table formal parameter. See *Oracle Database PL/SQL Language Reference* for more information on this topic.

Example 1-6   Invoking a subprogram with a nested table formal parameter

```
CREATE OR REPLACE PACKAGE pkg AUTHID CURRENT_USER AS

  TYPE names IS TABLE OF VARCHAR2(10);

  PROCEDURE print_names (x names);
END pkg;
/
CREATE OR REPLACE PACKAGE BODY pkg AS
  PROCEDURE print_names (x names) IS
  BEGIN
    FOR i IN x.FIRST .. x.LAST LOOP
      DBMS_OUTPUT.PUT_LINE(x(i));
    END LOOP;
  END;
END pkg;
/
DECLARE
  fruits   pkg.names;
  dyn_stmt VARCHAR2(3000);
BEGIN
  fruits := pkg.names('apple', 'banana', 'cherry');

  dyn_stmt := 'BEGIN print_names(:x); END;';
  EXECUTE IMMEDIATE dyn_stmt USING fruits;
END;
```

1.1.6 Native SQL Support for Query Row Limits and Row Offsets

Starting with Oracle Database 12*c*, Oracle provides a row limiting clause that enables native SQL support for query row limits and row offsets. If your application has queries that limit the number of rows returned or offset the starting row of the results, this feature significantly reduces SQL complexity for such queries.

1.1.6.1 Limiting Bulk Selection

Example 1-7 shows how to limit bulk selection with the FETCH FIRST clause. See *Oracle Database SQL Language Reference* for more information on this topic.

Example 1-7   How to limit bulk selection

```
DECLARE
  TYPE SalList IS TABLE OF employees.salary%TYPE;
  sals SalList;
BEGIN
  SELECT salary BULK COLLECT INTO sals FROM employees
    WHERE ROWNUM <= 50;

  SELECT salary BULK COLLECT INTO sals FROM employees
    SAMPLE (10);

  SELECT salary BULK COLLECT INTO sals FROM employees
    FETCH FIRST 50 ROWS ONLY;
END;
/
```

1.1.7 JDBC Driver Support for Application Migration

Many applications that you want to migrate to Oracle Database from other databases have Java applications that use JDBC to connect to the database. To facilitate SQL translation, Oracle Database 12*c* introduces a new set of JDBC APIs that are specific to SQL translation.

> **See Also:**
>
> - "SQL Translation of JDBC Applications"
> - API Reference for SQL Translation of JDBC Applications
> - Complete documentation of the `oracle.jdbc` package in *Oracle Database JDBC Java API Reference*
> - `http://www.oracle.com/technetwork/database/enterprise-edition/jdbc-112010-090769.html` for an updated list of JDBC drivers

1.1.8 ODBC Driver Support for Application Migration

ODBC driver supports the migration of third-party applications to Oracle Databases by using the SQL Translation Framework. This enables non-Oracle database SQL statements to run against Oracle Database. See "How to Use SQL Translation Framework" before beginning to migrate third-party ODBC application to Oracle Database.

To use this feature with an ODBC application, you must specify the service name, which was created as part of SQL Translation Framework setup, as the `ServerName=` entry in the `.odbc.ini` file.

If you require support for translation of Oracle errors (ORA errors) to your the native database, once your application starts running against Oracle Database, then you must enable the `SQLTranslateErrors=T` entry in the `.odbc.ini` file. See "SQL Translation of ODBC Applications" for more information on this topic.

1.2 Other Oracle Products that Enable Migration

Oracle recommends the use of several Oracle products as part of an overall migration strategy.

1.2.1 OEM Tuning and Performance Packs

For every type of migration, a few of the SQL statements used in the application must change, and some indexes must be re-built. Oracle SQL Tuning and Performance Packs provide guidance for the optimization step of the application migration.

1.2.2 Oracle GoldenGate

Oracle GoldenGate is a comprehensive software package for enabling the replication of data in heterogeneous data environments. The product set enables high availability solutions, real-time data integration, transactional change data capture, data replication, transformations, and verification between operational and analytical enterprise systems.

Oracle GoldenGate enables the exchange and manipulation of data at the transaction level among multiple, heterogeneous platforms across the enterprise. Its modular architecture provides the flexibility to extract and replicate selected data records, transactional changes, and changes to DDL (data definition language) across a variety of topologies.

When you migrate very large databases, the actual process of copying data from one database to another is time-consuming. During this time, the enterprise must continue delivering services using the old solution, which changes some of the data. These run-time changes must be captured and propagated to Oracle Database. Oracle GoldenGate captures these changes and enables side-by-side testing to ensure that the new solution performs as planned.

1.2.3 Oracle Database Gateways

Oracle Database Gateways address the needs of disparate data access. In a heterogeneously distributed environment, Gateways make it possible to integrate with any number of non-Oracle systems from an Oracle application. They enable integration with data stores such as IBM DB2, Microsoft SQL Server and Excel, transaction managers like IBM CICS and message queuing systems like IBM WebSphere MQ.

For more information about Oracle Database Gateways, see `http://www.oracle.com/technetwork/database/gateways/index.html`

1.2.4 Oracle SQL Developer

Oracle SQL Developer, as described in *Oracle SQL Developer User's Guide*, has a large suite of features that enable migration, including the following features:

- Support for database migration, such as schema, data, and server-side objects, from non-Oracle databases to Oracle Database (Migration Wizard)

- Support for application migration, including SQL statement pre-processing and data type translation support (Application Migration Assistant)

1.3 Migration Support for Other Database Vendors

Oracle provides migration support for applications running on various databases.

1.3.1 Application Support in Third-Party Databases

Table 1-1 provides information about the applications supported in several third-party databases. Note that while translation framework is available for DB2 LUW, a translator for DB2 is not available.

Table 1-1   Supported Applications in Databases

Application	SQL Server	DB2 LUW	DB2 AS400	Sybase ASE	Teradata	Informix
Oracle SQL Developer	Yes	Yes	No	Yes	Yes	No
Oracle Migration Workbench	No	No	Yes	No	No	Yes
SQL Translation Framework (SQL Translation Profile)	Yes	Yes	Yes	Yes	Yes	Yes
SQL Translation Framework (SQL Translator)	yes	Partial	No	Yes	No	No

1.3.2 Third-Party Database Version Support

Table 1-2 lists the supported database versions for migration using Oracle SQL Developer; this is not a comprehensive list. SQL translation may not work properly for every database.

Table 1-2   Supported Database Versions for Migration Using Oracle SQL Developer

RDBMS	Supported Versions
SQL Server	7.0, 2000, 2005,2008
Sybase Adaptive Server (ASE)	12, 15
Access	97, 2000, 2002 and 2003
MySQL	3,4,5
DB2	AS400 V4R3, V4R5
DB2 LUW	8, 9
Teradata	12
Informix	7.3, 9.1, 9.2, 9.3, 9.4

2
SQL Translation Framework Overview

Various client-side applications, designed to work with non-Oracle Databases, cannot be used with Oracle Database without significant alterations. This is because SQL dialect varies among vendors of database technologies and different vendors use different syntaxes to express SQL queries and statements.

Starting with Oracle Database 12*c*, there is a new mechanism called SQL Translation Framework. It translates the SQL statements of a client program from a foreign (non-Oracle) SQL dialect into the SQL dialect used by the Oracle Database SQL compiler.

In addition to translating non-Oracle SQL statements, the SQL Translation Framework may be used to substitute an Oracle SQL statement with another Oracle statement to address a semantic or performance issue. In this way, you can address an application issue without patching the client application.

The SQL translation framework consists of two basic components: SQL Translator, and SQL Translation Profile.

The SQL Translator

The SQL Translator is a software component, provided by Oracle or third-party vendors, which can be installed in Oracle Database. It translates the SQL statements of a client program before they are processed by the Oracle Database SQL compiler. If an error results from translated SQL statement execution, then Oracle Database SQL compiler generates an Oracle error message.

The SQL Translator automatically translates non-Oracle SQL to Oracle SQL, thereby enabling the existing client-side application code to run largely unchanged against an Oracle Database. This reduces the cost of migration to Oracle Database storage significantly. As a corollary, the translation feature may be used in other scenarios, where it may be expedient to intervene between the original SQL statement submitted by the client and its actual execution.

The SQL Translation Profile

The SQL Translation Profile is a database object that contains the set of captured non-Oracle SQL statements, and their translations or translation errors. The SQL Translation Profile is used to review, approve, and modify translations. A profile is associated to a single translator. However, a translator can be used in one or more SQL Translation Profiles. Typically, there is one SQL Translation Profile per application, otherwise applications can share translated queries. You can export profiles among various databases.

The following figure illustrates the run-time overview the SQL Translation Framework.

Figure 2-1   SQL Translation Framework at Runtime
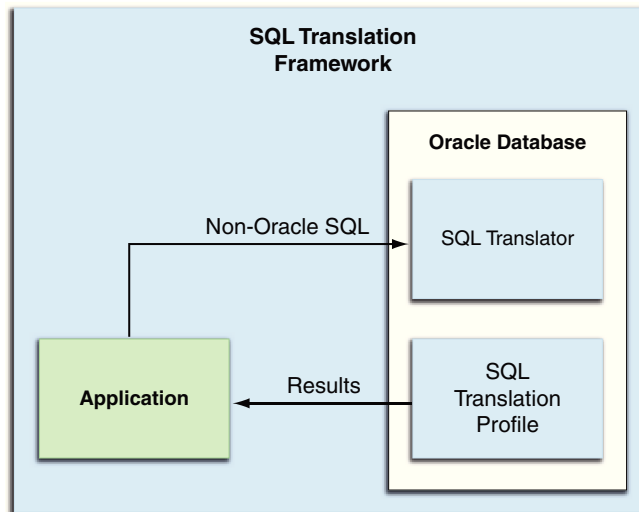
2.1 Architecture of SQL Translation Framework

The key component of SQL Translation Framework is the SQL Translation Profile. The profile is a collection of non-Oracle statements that are processed through the translator. The application determines which profile to use when connecting to the Oracle Database. The translator handles the actual translation work.

In most cases, the non-Oracle SQL statements and errors are translated by a SQL Translator registered in the profile. The translator may be supplied by Oracle or by a third-party vendor. If the translator does not have a translation for a particular SQL statement or error, then you may register your own custom translation. You may also wish to register your own custom translation to override the default translator and to customize your translation results.

2.2 How to Use SQL Translation Framework

Perform the following steps to use SQL Translation Framework:

1. Install a SQL Translator, either from Oracle or a third-party vendor, in Oracle Database.

2. Create a SQL Translation Profile and register the SQL Translator with the profile.

3. Create a Database service and specify the SQL Translation Profile as a service attribute to which the application can connect.

   Note that setting the SQL Translation Profile at the service level ensures that everything running through that listener service is translated automatically.

   The translator can also be activated at connection level by using the `ALTER SESSION` statement or the `LOGON` triggers.

4. Link the application with an Oracle driver to connect the application to Oracle Database. You must also change the connection settings to connect to the Database service with the SQL Translation Profile.

5. Test all functionality of the application against Oracle Database. As the application runs, the SQL Translation Profile translates SQL statements of the application from the third-party SQL dialect to semantically-equivalent Oracle syntax and register them in the profile.

   If the translator does not have a translation for a particular SQL statement or error, then you may register your own translation to fill its place.

6. Verify the custom translations and edit them, if required. Alternatively, register new ones to ensure that the application performs as intended, until testing is complete.

   Oracle recommends establishing a test environment and rigorously testing the application, ideally through a regression test suite.

7. Set up the server-side application objects and data in the production Oracle Database for deployment to a production environment.

8. Create a database service with the profile set as a service attribute and change the connection settings of the application, so that it connects to the database service in the production database. The application is expected to run as tested.

Oracle recommends that the application be monitored to guard against the possibility of errors due to unavailability of translation of any SQL statement. You must first disable the automatic translation of new and unseen SQL statements in the profile; when any such statement is encountered, it raises an error that is logged. In cases of alerts for mis-translation, you must make adjustments to the profile.

> ✎ **See Also:**
>
> - The new `DBMS_SQL_TRANSLATOR` PL/SQL package and updated `DBMS_SQL` and `DBMS_SERVICE` PL/SQL packages in the *Oracle Database PL/SQL Packages and Types Reference*.
>
> - Updated `GRANT` and `REVOKE` statements and new system privileges in the *Oracle Database SQL Language Reference*.
>
> - *Oracle Database PL/SQL Packages and Types Reference*
>
> - *Oracle Database Administrator's Guide*

2.3 When to Use SQL Translation Framework

Use SQL Translation to migrate a client application that uses SQL statements with vendor-proprietary SQL syntax.

Currently, SQL Translators are available only for Sybase and SQL Server, and there is limited support for DB2.

SQL Translation Framework is designed for use with open API applications, such as ODBC or JDBC, and applications that use SQL statements that may be translated into semantically-equivalent Oracle syntax. These applications must relink to the Oracle ODBC or JDBC driver and then execute through the translation service.

Following are the possible scenarios for the connection mechanism:

- If the application uses ODBC, JDBC, OLE DB or .NET driver, or data provider to connect to the database, then the driver or data provider for Oracle must be replaced.

- If the application uses MySQL client library to connect to MySQL, then the library with Mysql Client Library Driver for Oracle must be replaced.

- No direct translator is available for DB2. For more information, refer to "Migration Support for Other Database Vendors".

  If the application uses IBM DRDA network protocol to connect to DB2, then the database connection settings must be changed to connect to Oracle through DRDA Application Server for Oracle.

- If the application uses a vendor-proprietary C client API (the case of Sybase), then the API calls must be replaced with appropriate Oracle OCI APIs.

3

SQL Translation Framework Configuration

The SQL Translation Framework may be installed and configured using Oracle SQL Developer, or from the command line interface. In either case, the user must have the necessary permissions to install SQL Translator.

3.1 Installing and Configuring SQL Translation Framework with Oracle SQL Developer

You can use the DBA Navigator in Oracle SQL Developer 3.2 to install and manage the translator and translation profile.

3.1.1 Overview of Oracle SQL Developer Migration Support

The SQL Translation framework is installed as part of Oracle Database installation. However, it must be configured to recognize the non-Oracle SQL dialect of the application and you must install at least one translator to fully utilize the framework.

Before using the SQL Translation feature, you must migrate your data, schema, stored procedures, triggers, and views. Oracle implements database schema migration and data migration through Oracle SQL Developer functionality. Oracle SQL Developer simplifies the process of migrating a non-Oracle database to an Oracle Database through the use of Migration Wizard. The Migration wizard provides convenient and comprehensive guidance through the phases involved in migrating a database.

Oracle SQL Developer captures information from the source non-Oracle database and displays it in a captured model, which is a representation of the structure of the source database. This representation is stored in a migration repository, which is a collection of schema objects that Oracle SQL Developer uses to store migration information.

The information in the repository is used to generate the converted model, which is a representation of the structure of the destination database as it will be implemented in the Oracle database. You can then use the information in the captured model and the converted model to compare database objects, identify conflicts with Oracle reserved words, and manage the migration progress. When you are ready to migrate, generate the Oracle schema objects, and then migrate the data.

This section describes how to perform the subsequent tasks that enable automatic run-time migration. These examples use SQL Translator with a JDBC application that runs against a Sybase database; they can be easily adapted for other client/database configurations. Note that Oracle SQL Developer is shipped with an installed Sybase translator.

See *Oracle SQL Developer User's Guide* for more information.

3.1.2 Setting Up Oracle SQL Developer 3.2 for Windows

Oracle SQL Developer 3.2 is shipped with Oracle Database 11*g* JDBC drivers and there is no client for Windows in this release. If you are using a Windows system, then you must enable Oracle SQL Developer 3.2 to use Oracle Database 12*c* JDBC driver, so that all the features of the current release are enabled. Perform the following steps to achieve this:

- Rename the `sqldeveloper\jdbc\lib` folder to `sqldeveloper\jdbc\lib_11g`.

- Create a new empty folder as `sqldeveloper\jdbc\lib`.

- Unzip Oracle Database 12*c* JDBC JAR files into the new `sqldeveloper\jdbc\lib` folder.

  See *Oracle Database JDBC Developer's Guide* for more information about Oracle Database 12*c* JDBC files.

3.1.2.1 Setting Up Oracle SQL Developer 3.2 Startup

Oracle SQL Developer automatically uses JDBC drivers found in any `ORACLE_HOME` `\client` directory. To override this behavior and make Oracle SQL Developer use JDBC drivers in the `sqldeveloper\jdbc\lib` directory, create a new `sqldeveloper.bat` file in the `sqldeveloper` directory:

```
set ORACLE_HOME=%CD%
start sqldeveloper.exe
```

3.1.2.2 Starting Oracle SQL Developer

Run the `sqldeveloper.bat` file to run Oracle SQL Developer.

To check the JDBC driver configuration:

1. Select **About** from **Help** menu.

2. Select **Properties**. It must display the configuration as shown in Figure 3-1:

Figure 3-1   Checking JDBC Configuration for Oracle SQL Developer
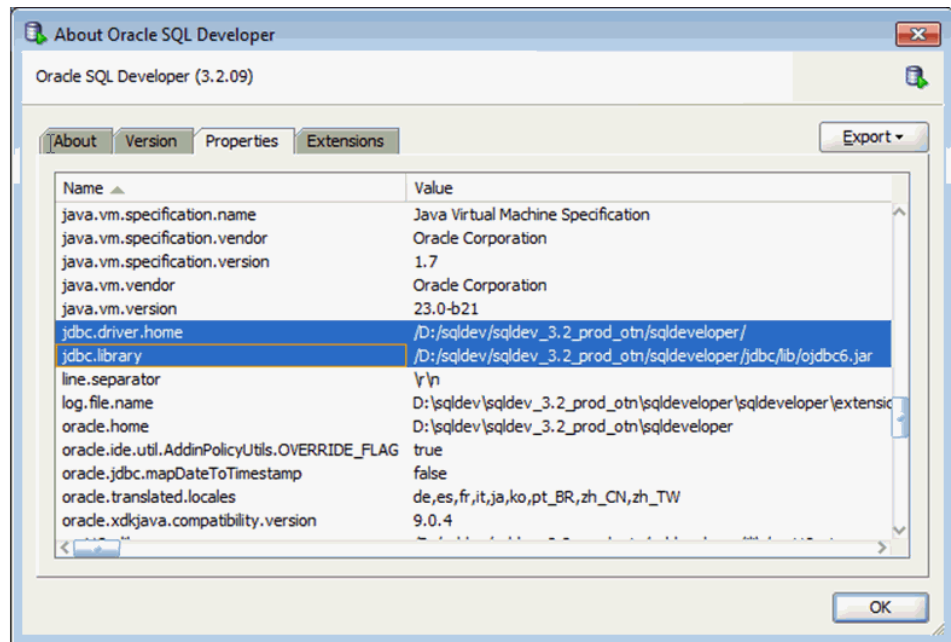
3.1.3 Creating a Connection to Oracle Database

Create a connection to the Database with the credentials as shown in Figure 3-2:

Figure 3-2   Creating an Oracle Database Connection
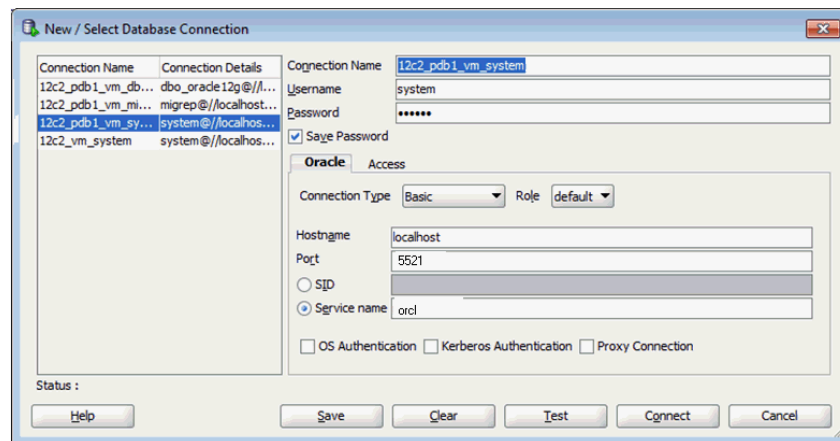
You can use the following command to check the database you are connected to and the JDBC driver being used:

```
show jdbc
```

Setting Up Migration Preferences

You must set up the migration preferences in the following way:

1. Select **Preferences** from the Tools menu.

2. Select **Generation Options** from **Migration** option on the left panel, as shown in Figure 3-3.

Figure 3-3   Setting Up Migration Preferences in Oracle SQL Developer
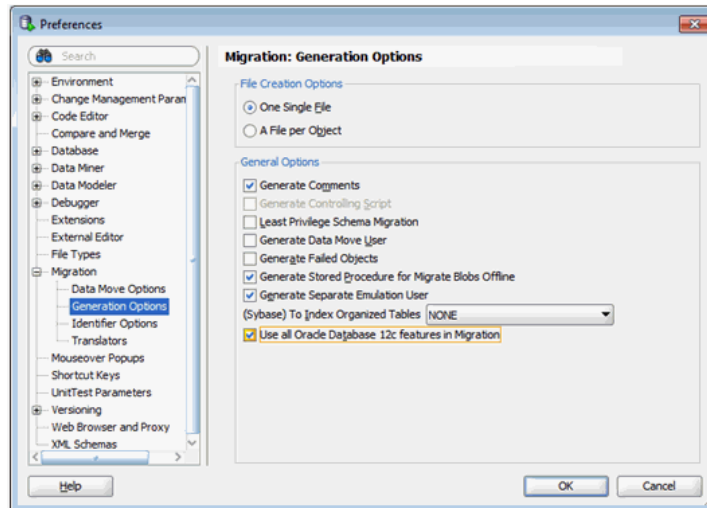
3.1.4 Testing SQL Translation

Perform the following steps to determine whether Sybase SQL Translator is properly installed or not:

1. Open Oracle SQL Developer.

2. From the **Tools** menu, select **Migration**, and then select **Translation Scratch Editor**.
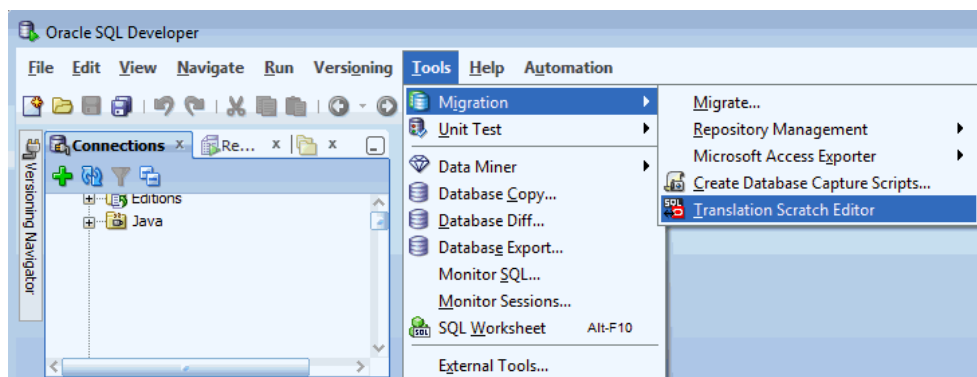
3. In the Scratch Editor toolbar, select **Sybase T_SQL To PL/SQL** option, which is the Sybase translator.
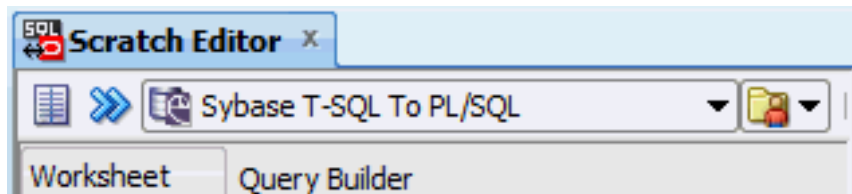
4. In the left panel of the **Scratch Editor**, enter the following query in Sybase SQL dialect:

```
select top 10 * from dual
```

5. Click the **Translate** icon.

   The translated query text is displayed in the right panel of the editor.
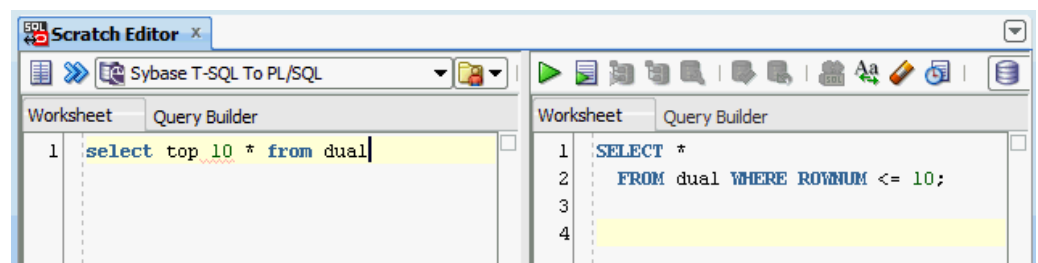
3.1.5 Creating a Translation Profile and Installing SQL Translator

Oracle SQL Developer is installed with Oracle Database 12*c*. It loads Java classes of the Sybase Translator, approximately 15 MB, into Oracle Database. Due to the size and the number of Java classes loaded, Oracle recommends you to install the translator locally, and not over a WAN.

If the translator is installed under a user profile that has a pre-existing migration repository, the translator picks up the context of the database, such as name changes. Therefore, you must create a new user with the following specifications:

- `CONNECT`, `RESOURCE`, and `CREATE VIEW` privileges
- Access to storage in the `SYSTEM` and/or `USER` tablespace

3.1.5.1 Installing SQL Translator

To install SQL Translator:

1. Log into the database using `ADMIN` privileges.

2. At the command line, enter the following commands.

   ```
   GRANT CONNECT, RESOURCE, CREATE VIEW TO TranslUser identified by TranslUser;
   ALTER USER TranslUser QUOTA UNLIMITED ON SYSTEM;
   ```

3. From the View menu, select **DBA**.
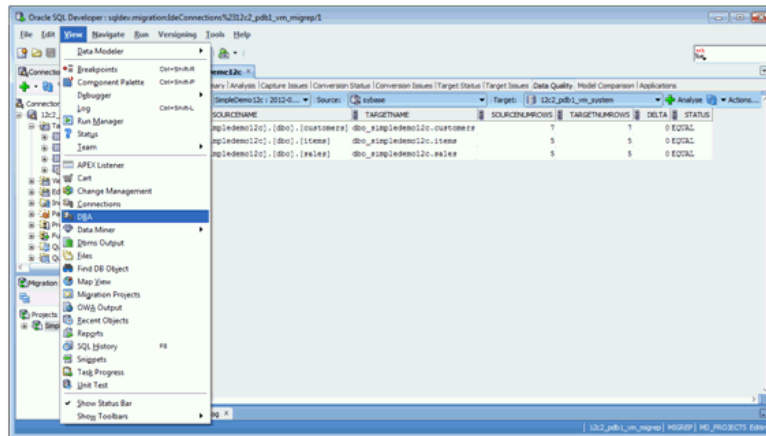
4. In the DBA Navigator, right-click **Connections** and select **Add Connection.**
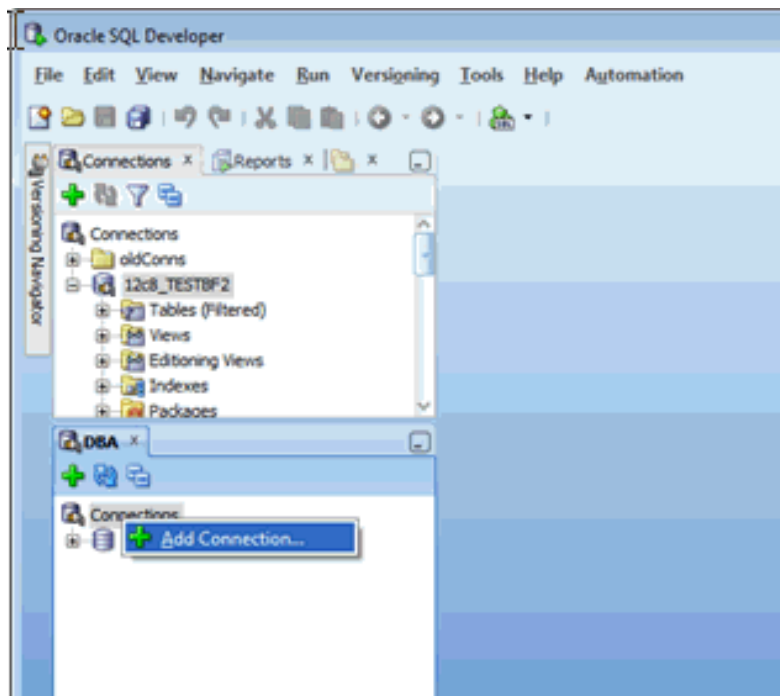
5. In the **Select Connection** box, select the connection if you want to use an existing connection. If you want to create a new connection, then add the information for `transluser` discussed in step 2.
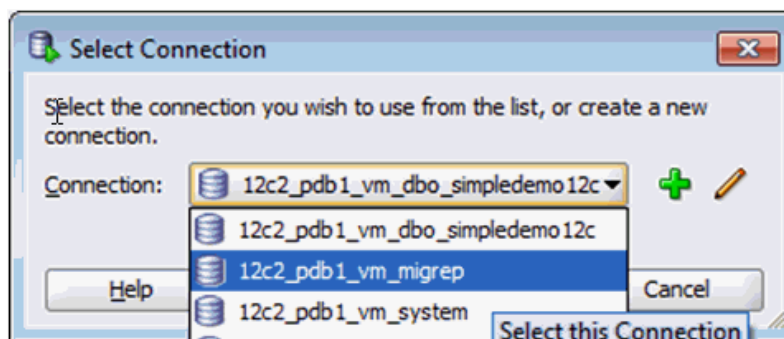
6. Click **Connect**.

7. In the DBA navigator, right-click the connection created in the preceding steps, and select **Install SQL Translator**.
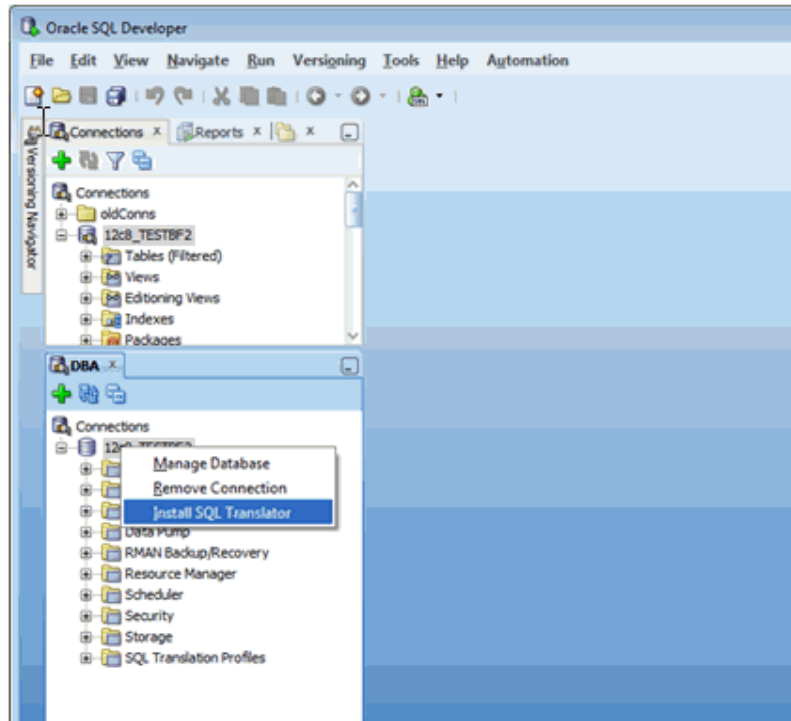
The Install SQL Translator dialog box opens.

You must have special permissions to install the SQL Translator and create a SQL Translation Profile. You will be prompted to provide the sys password, so that these privileges can be granted. Refer to "Granting Necessary Permissions for Installing the SQL Translator" for more information about these privileges.

8. Create a SQL Translation Profile, following steps described in "Creating a Translation Profile ".

9. Verify that the user has sufficient privileges to run the translation profile.

You may have to login as sys user to grant additional privileges.
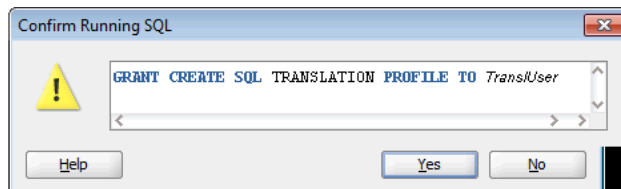
10. Install SQL Translator.
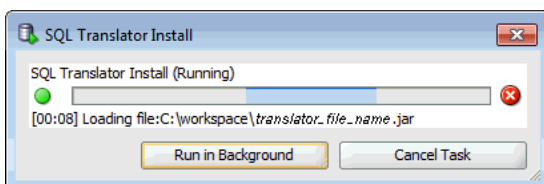
11. To ensure that both the Profile and Translator are properly installed, verify whether the appropriate package and Java class files are present or not in the Connections pane.
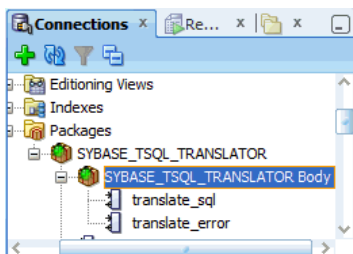
3.1.5.2 Creating a Translation Profile

To create a translation profile:

1. From the SQL Translator drop-down box, select **Sybase** or **SQL Translator**.

2. Check **Create New Profile**.

3. Enter *SYBASE_PROFILE* in Profile Name field.

4. In Profile Schema, select the name of the user created in section "Creating a Translation Profile and Installing SQL Translator".

5. Click **Apply**.
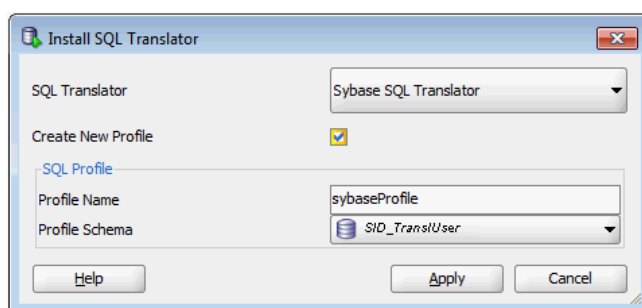
3.1.6 Using the SQL Translator Profile

To test the SQL Translation Profile, use SQL Worksheet:

1. Right-click the *SYBASE_PROFILE* node.

2. Select **Open SQL Worksheet with Profile**.

3. Enter a T-SQL statement that you want to translate.
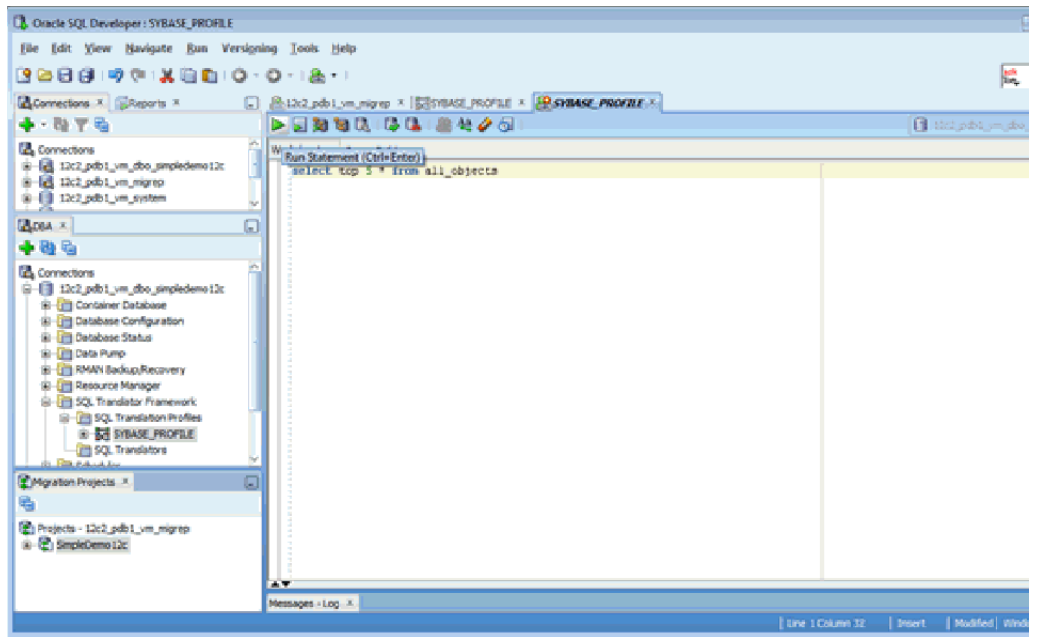
4. Click `SYBASE_PROFILE` and select the SQL Translation tab to inspect the profile and view the translated statement.
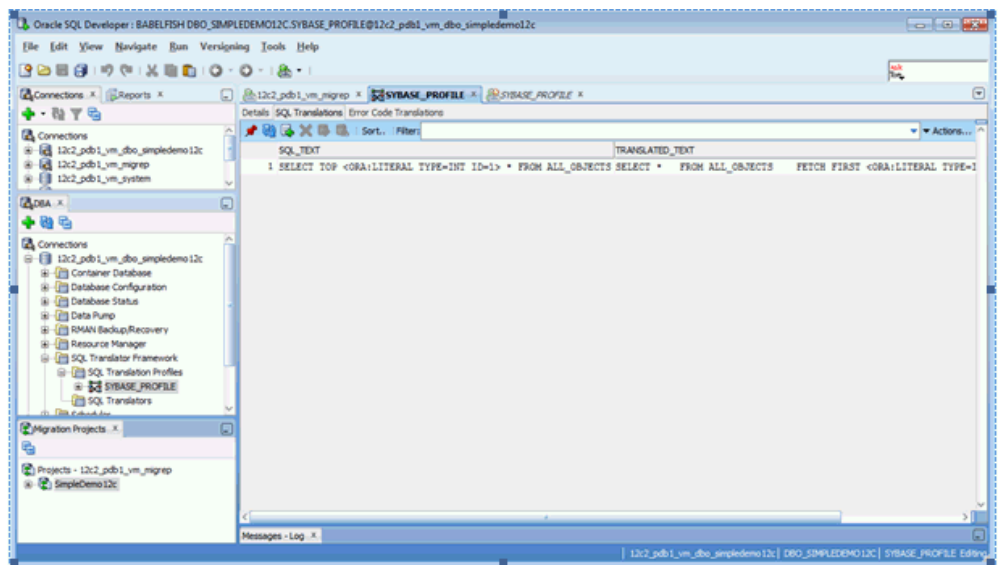
An alternative way to view the profile SQL in a better way when you double-click on it, the fingerprint and template open in a Translation Scratch Editor as shown in the following images:
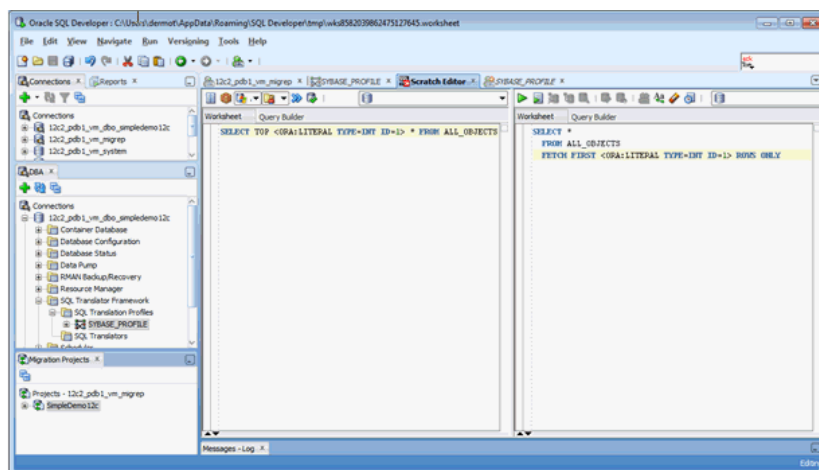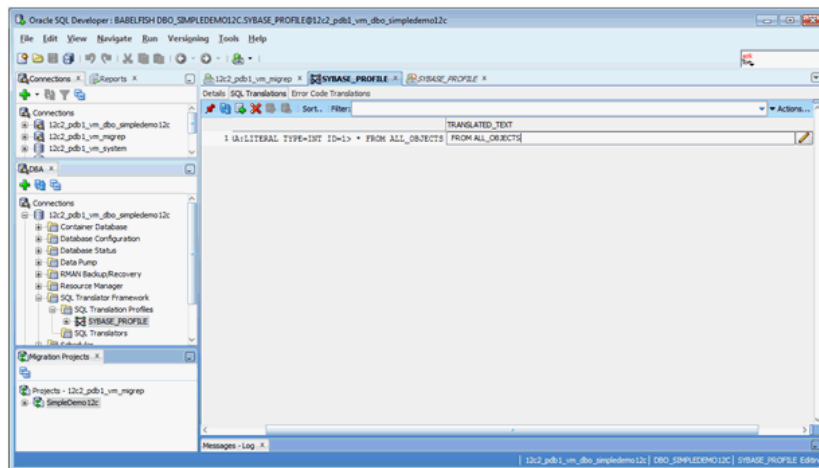
3.2 Installing and Configuring SQL Translation Framework from Command Line

There are several processes that you must complete to successfully install and configure the SQL Translation Framework from command line interface.

3.2.1 Installing Oracle Sybase Translator

To install Oracle Sybase Translator, Use Oracle SQL Developer as described in "Installing and Configuring SQL Translation Framework with Oracle SQL Developer".

3.2.2 Setting up a SQL Translation Profile

Perform the following steps to set up a SQL Translation Profile through a command-line interface:

1. Login as a `system` user.

   ```
   > sqlplus system/<password>
   ```

2. Grant create privileges to the standard user.

This allows the standard user to create a SQL Translation Profile.

```
 SQL> grant create sql translation profile to <user>;
```

3. Login as a standard user.

```
sqlplus <user>/<password>
```

4. Invoke the methods of DBMS_SQL_TRANSLATOR PL/SQL package to create and configure the translation profile.

```
SQL> exec dbms_sql_translator.create_profile('sybase_profile')
SQL> exec dbms_sql_translator.set_attribute('sybase_profile',
     dbms_sql_translator.attr_translator,
     'migration_repo.sybase_tsql_translator')
```

5. Grant all privileges for the SQL Translation Profile to Oracle Sybase translation schema.

```
SQL> grant all on sql translation profile sybase_profile to migration_repo;
```

3.2.3 Setting Up a Database Service to Use the SQL Translation Profile

This section describes how to add a database service in a standard environment and in an Oracle Real Application Clusters environment.

Setting Up a Database Service in a Standard Environment

To set up a database service in a standard environment:

1. Login as a DBA

2. Issue the following commands to use the DBMS_SERVICE PL/SQL package to create and invoke the database service:

```
SQL> declare
  params dbms_service.svc_parameter_array;
begin
  params('SQL_TRANSLATION_PROFILE') := 'user.sybase_profile';
  dbms_service.create_service('sybase_service', 'network_name', params);
  dbms_service.start_service('sybase_service');
end;
/
```

3.2.3.1 Setting Up a Database Service in Oracle Real Application Clusters

To set up a database service in Oracle Real Application Clusters:

1. Add the database service:

```
srvctl add service -db db_name -service sybase_service
-sql_translation_profile user.sybase_profile
```

2. Start the database service:

```
srvctl start service -db db_name -service sybase_service
```

3.2.4 Testing Sybase SQL Translation Using the SQL Translation Profile

Perform the following steps to test the translation:

1. Login as a standard user:

   ```
   sqlplus user/password
   ```

2. Specify the SQL Translation Profile at the SQL prompt:

   ```
   SQL> alter session set sql_translation_profile = sybase_profile;
   ```

3. Force the database to treat SQL*Plus as a foreign SQL application:

   ```
   SQL> alter session set events = '10601 trace name context forever, level 32';
   ```

4. Run a SQL query that uses Sybase SQL dialect. For example:

   ```
   select top 3 * from emp;
   ```

5. The query returns the following results:

   ```
   EMPNO   ENAME    JOB         MGR   HIREDATE      SAL    COMM    DEPTNO
   ------------------------------------------------------------------
   7369    SMITH    CLERK      7902   17-DEC-80     800              20
   7499    ALLEN    SALESMAN   7698   20-FEB-81    1600    300       30
   7521    WARD     SALESMAN   7698   22-FEB-81    1250    500       30
   ```

3.3 Granting Necessary Permissions for Installing the SQL Translator

This section discusses the privileges that you must have to install the SQL Translator. The SYBASE_PROFILE created here has the following two users:

- MIGREP, where the translator is installed
- TARGET_USER, where the profile is installed

To grant privileges necessary for installing the SQL Translator:

1. Connect as SYS to grant the required privileges:

   ```
   connect sys/oracle as sysdba
   ```

2. Allow MIGREP to create a view and have access to unlimited quota:

   ```
   GRANT connect, resource, create view to MIGREP;
   ALTER USER MIGREP QUOTA UNLIMITED ON USERS;
   ```

3. Allow TARGET_USER to create a view and have access to unlimited quota:

   ```
   GRANT connect, resource, create view to TARGET_USER;
   ALTER USER MIGREP QUOTA UNLIMITED ON TARGET_USER;
   ```

4. Allow MIGREP to load a SQL Translator:

   ```
   BEGIN
     DBMS_JAVA.GRANT_PERMISSION(UPPER('MIGREP'), 'SYS:java.lang.RuntimePermission',
   'getClassLoader', '');
   END;
   /
   ```

5. Allow `TARGET_USER` to create profiles:

```
GRANT CREATE SQL TRANSLATION PROFILE TO TARGET_USER;
```

6. Allow `TARGET_USER` to explicitly alter the session to use a profile:

```
GRANT ALTER SESSION TO TARGET_USER;
```

This privilege is not granted in SQL Developer by default.

7. Allow the translator to make reference to the profile:

```
CONNECT TARGET_USER/TARGET_USER;
GRANT ALL ON SQL TRANSLATION PROFILE SYBASE_PROFILE TO MIGREP;
```

8. Allow the profile to make reference to the translator:

```
CONNECT MIGREP/MIGREP;
GRANT EXECUTE ON SYBASE_TSQL_TRANSLATOR TO TARGET_USER;
```

4

SQL Translation of JDBC and ODBC Applications

Oracle provides SQL Translation mechanisms for use with JDBC and ODBC applications.

4.1 SQL Translation of JDBC Applications

Consider the concepts necessary to understanding how to use SQL Translator with a JDBC application.

4.1.1 SQL Translation Profile

A SQL Translation Profile is a database schema object that directs how SQL statements in non-Oracle dialects are translated into Oracle SQL dialects. It also directs how Oracle error codes and `SQLSTATES` are translated into the SQL dialect of other vendors.

When you want to migrate a client application written for a non-Oracle SQL database to Oracle, you can create a SQL Translation Profile and configure it to translate the SQL statements and errors for the application. At runtime, the application sets the profile for the connection in Oracle Database to translate its SQL statements and errors. This profile is set using the `oracle.jdbc.sqlTranslationProfile` property.

When necessary, you can register custom translations of SQL statements and errors with the SQL Translation Profile on the Server. When a SQL statement or error is translated, then first, the custom translation is looked up and then, the translator is invoked only if no match is found.

See "Architecture of SQL Translation Framework" and "Setting up a SQL Translation Profile".

4.1.2 Error Message Translation

You may prefer receiving error messages in the form of messages that used to be thrown by the native database. You must then use the error message translation file, which translates error messages when there is no valid connection to the database. Once a connection to the database is established, the JDBC driver bypasses this file completely and all errors are handled by the translator on the server. Similar to query translation, you can also register custom error translations on the server.

The error message translation file is not written by a specific component. You must provide the file for translation and specify the name of the file. You can also provide the file path as the value of the corresponding connection property.

The error message translation file is in XML format; it contains a series of error translations. Each error translation contains the following information:

Translation Error	Type
ORA error number	positive integer
Oracle error message	String
Translated error code	positive integer
Translated SQL State	positive integer

4.1.3 Converting JDBC Standard Parameter Markers

Before submitting the SQL statements for translation., the JDBC driver internally converts the JDBC standard parameter markers (?) into Oracle style parameter markers of the format `:b<n>`.

Here, the naming format for the parameter markers is `:b<n>`, where `n` is an incremental number to specify the position of the (?) marker in the JDBC `PreparedStatement`.

Consider the `UPDATE employees SET salary = salary * ? WHERE employee_id = ?` `PreparedStatement` statement, where, the first parameter marker (?) will become `:b1` and the second parameter marker (?) will become `:b2`.

After conversion, the driver sends the following query to the server for translation:

```
UPDATE employees SET salary = salary * :b1 WHERE employee_id = :b2
```

Note that any query that contains "?" as a parameter marker fails during the connection translation phase if you change the value of the `processEscapes` property to `FALSE`. For a successful translation, you must retain the default value of the `processEscapes` property.

Converting parameter markers helps the driver to automatically reorder any parameter changes that occurred at translation. At the time of conversion, any custom translation that must be registered on the server should be registered from the Oracle style parameter marker version; the server receives the statements. Note that, the custom translation must have the same number of parameter markers in the Oracle style as in the original query.

For more information about supported JDBC APIs, API Reference for SQL Translation of JDBC Applications .

4.1.4 Executing the Translated Oracle Dialect Query

After the JDBC standard parameter markers are converted into Oracle style parameter markers, the driver makes a round-trip to the server for translating the query into Oracle dialect. Once the translated query is received by the server, any reordering in the parameters in handled transparently by the driver, and the query is executed as a normal query.

If a query cannot be translated due to the unavailability of translation, then the server can either raise an error or return a `NULL`, based on the value of the `DBMS_SQL_TRANSLATOR.ATTR_RAISE_TRANSLATION_ERROR` profile attribute. If the server returns a `NULL`, then the original untranslated query is assumed to be the query translated by the driver and executed.

The driver keeps the translation in the local caches to save the future round-trip.

Note that the JDBC driver can support the translation errors (when the query cannot be translated due to the unavailability of translation) set by either value of the `DBMS_SQL_TRANSLATOR.ATTR_RAISE_TRANSLATION_ERROR` attribute. However, the value must be set on the server before the connection is established. Because a change in the value of this attribute in the middle of a session may result in inconsistent behavior, Oracle recommends that you do not flip the value of this attribute during a session. See *Oracle Database PL/SQL Packages and Types Reference* for more information about the `TRANSLATE_SQL` procedure.

4.1.5 Error Translation

If any `SQLException` is thrown during the query execution, the driver transparently makes a trip to the server and translates the exception from Oracle codes to the original vendor-specific code. So, the resulting `SQLException` has both vendor-specific code and `SQLSTATE` along with the Oracle-specific `SQLException` as the cause.

Similar to query translation, custom error translations can also be registered on the server and given priority over standard translation. The `DBMS_SQL_TRANSLATOR.ATTR_RAISE_TRANSLATION_ERROR` attribute has the same effect on custom error translation as on query translation.

Note that the errors are translated only after a connection to the server is established. So, for errors that occur before the connection to the server is established, Error Message Translation is used.

4.1.6 Using JDBC Driver for SQL Translation

Example 4-1 demonstrates how to use a JDBC driver for SQL translation. You must first grant the `CREATE SQL TRANSLATION PROFILE` privilege to `HR` as follows:

```
conn system/manager;
grant create sql translation profile to HR;
exit
```

Now, connect to the database as `HR` and execute the following SQL statements:

```
drop table sample_tab;
create table sample_tab (c1 number, c2 varchar2(100));
insert into sample_tab values (1, 'A');
insert into sample_tab values (1, 'A');
insert into sample_tab values (1, 'A');
commit;
exec dbms_sql_translator.drop_profile('FOO');
exec  dbms_sql_translator.create_profile('FOO');
exec  dbms_sql_translator.register_sql_translation('FOO','select row of select c1,
c2 from sample_tab
where c1=:b1 and c2=:b2','select c1, c2 from sample_tab where c1=:b1 and c2=:b2');
```

Now, you can run the following program that translates SQL statements that use JDBC standard parameter markers.

Example 4-1   Translating Non-Oracle SQL Statements to Oracle SQL Dialect Using JDBC Driver

```
public class SQLTransPstmt
{
  static String url="jdbc:oracle:thin:@localhost:5521:jvx1";
  static String user="HR", pwd="hr";
```

```
  static String PROFILE = "FOO";
  static String primitiveSql = "select row of select c1, c2 from sample_tab where
c1=? and c2=?";

// Note that this query contains JDBC style parameter markers
// But the preceding custom translation registered in SQL is using Oracle style
markers

  public static void main(String[] args) throws Exception
  {
    OracleDataSource ods = new OracleDataSource();
    ods.setURL(url);

    Properties props = new Properties();
    props.put("user", user);
    props.put("password", pwd);

    // The Following connection property makes the connection translating
    props.put(OracleConnection.CONNECTION_PROPERTY_SQL_TRANSLATION_PROFILE, PROFILE);
    ods.setConnectionProperties(props);
    Connection conn = ods.getConnection();
    System.out.println("connection for SQL translation: "+conn);

    try{
      // Any statements created using a translating connection are
      // automatically translating. If you want to get a non-translating
      // statement out of a translating connection please have a look at
      // the oracle.jdbc.OracleTranslatingConnection Interface.
      // Refer to "OracleTranslatingConnection Interface"
      // for more information
      PreparedStatement trStmt = conn.prepareStatement(primitiveSql);
      trStmt.setInt(1, 1);
      trStmt.setString(2, "A");
      System.out.println("executeQuery for: "+primitiveSql);
      ResultSet trRs = trStmt.executeQuery();
      while (trRs.next())
        System.out.println("C1:"+trRs.getInt(1)+", C2:"+trRs.getString(2));
      trRs.close();

      trStmt.close();
    }catch (Exception e) {
      e.printStackTrace();
    }


    conn.close();
  }
}
```

4.2 SQL Translation of ODBC Applications

Consider the concepts necessary to understanding how to use SQL Translator with an ODBC application.

4.2.1 SQL Translation profile

For ODBC applications, the SQL Translation Profile is set at the service level. So, you do not require to set it in the `.odbc.ini` file.

4.2.2 Error Message Translation

You may prefer receiving error messages in the form of messages that used to be thrown by the native database. In such cases, when the application is set to run on Oracle Database, you must set the `SQLTranslateErrors=T` entry in the `.odbc.ini` file to translate the `ORA` errors to their native form.

4.2.3 Translating Error Messages

Example 4-2 demonstrates how to use the ODBC driver in SQL translation. The SQL statement used in the example uses Sybase `TOP N` syntax.

Note that you must set the `ServerName=` entry in the `.odbc.ini` file with the Database service name created in "How to Use SQL Translation Framework" section. Also, set the '`SQLTranslateErros=T` entry in the `.odbc.ini` file, if you require translation of Oracle errors to native database errors.

Example 4-2   Translating Non-Oracle SQL to Oracle SQL Dialect Using ODBC Driver

```
int main()
{
 HENV m_henv;           /* environment handle */
 HDBC m_hdbc;           /* connection handle */
 HSTMT m_hstmt;         /* statement handle */
 int retCode;           /* return code */
 char dbdsn[100];       /* Initialize with the DSN name of connection */
 const  char szUID[10];/*Initialize with appropriate Username of DB */
 const char szPWD[10]; /* Initialize with appropriate Password */

 char query1[100]="select top 3 col1 from babel_tab3 order by col1";
 SQLLEN paramInd = SQL_NTS;
 SQLUINTEGER   no = 0;

 //Allocate HENV, HDBC, HSTMT handles
 retCode = SQLAllocHandle (SQL_HANDLE_ENV, SQL_NULL_HANDLE, &m_henv);
 if (retCode != SQL_SUCCESS && retCode != SQL_SUCCESS_WITH_INFO)
 {
    printf ("SQLAllocHandle failed \n");
    printSQLError (1, m_henv);
 }

 retCode = SQLSetEnvAttr (m_henv, SQL_ATTR_ODBC_VERSION, (void *) SQL_OV_ODBC3,
           SQL_IS_INTEGER);
 if (retCode != SQL_SUCCESS && retCode != SQL_SUCCESS_WITH_INFO)
 {
    printf ("SQLSetEnvAttr failed\n");
    printSQLError (1, m_henv);
 }

 retCode = SQLAllocHandle (SQL_HANDLE_DBC, m_henv, &m_hdbc);
 if (retCode != SQL_SUCCESS && retCode != SQL_SUCCESS_WITH_INFO)
 {
    printf ("SQLAllocHandle failed\n");
    printSQLError (2, m_hdbc);
 }

 retCode = SQLConnect (m_hdbc, (SQLCHAR *) dbdsn,SQL_NTS,
```

```
                          (SQLCHAR *) szUID, SQL_NTS,
                          (SQLCHAR *) szPWD, SQL_NTS);
if (retCode != SQL_SUCCESS && retCode != SQL_SUCCESS_WITH_INFO)
{
   printf ("SQLConnect failed to connect\n");
   printSQLError (2, m_hdbc);
}

retCode = SQLAllocHandle (SQL_HANDLE_STMT, m_hdbc, &m_hstmt);
if (retCode != SQL_SUCCESS && retCode != SQL_SUCCESS_WITH_INFO)
{
   printf ("SQLAllocHandle with SQL_HANDLE_STMT failed\n");
   printSQLError (3, m_hstmt);
}

/* Prepare and Execute the Sybase Top-N syntax SQL statements */

retCode = SQLPrepare (m_hstmt, (SQLCHAR *) query1, SQL_NTS);
if (retCode != SQL_SUCCESS && retCode != SQL_SUCCESS_WITH_INFO)
{
   printf ("SQLPrepare failed\n");
   printSQLError (3, m_hstmt);
}

retCode=SQLExecute(m_hstmt);
if (retCode != SQL_SUCCESS && retCode != SQL_SUCCESS_WITH_INFO)
{
   printf ("SQLExecute-failed\n");
   printSQLError (3, m_hstmt);
}

while (retCode = SQLFetch(m_hstmt)!=SQL_NO_DATA)
{
   retCode=SQLGetData(m_hstmt,1,SQL_C_ULONG, &no, 0, &paramInd);
   if (retCode != SQL_SUCCESS && retCode != SQL_SUCCESS_WITH_INFO)
   {
      printf ("SQLFetch failed\n");
      printSQLError (3, m_hstmt);
   }
   printf("Value is %d\n",no);
}

retCode = SQLCloseCursor (m_hstmt);
if (retCode != SQL_SUCCESS && retCode != SQL_SUCCESS_WITH_INFO)
   printf ("SQLCloseCursor failed\n");

printf ("cleanup()\n");
retCode = SQLFreeHandle (SQL_HANDLE_STMT, m_hstmt);
if (retCode != SQL_SUCCESS && retCode != SQL_SUCCESS_WITH_INFO)
{
   printf ("SQLFreeHandle failed\n");
   printSQLError (3, m_hstmt);
}

retCode = SQLDisconnect (m_hdbc);
if (retCode != SQL_SUCCESS && retCode != SQL_SUCCESS_WITH_INFO)
{
   printf ("SQLDisconnect failed\n");
   printSQLError (2, m_hdbc);
}
```

```
retCode = SQLFreeHandle (SQL_HANDLE_DBC, m_hdbc);
if (retCode != SQL_SUCCESS && retCode != SQL_SUCCESS_WITH_INFO)
{
   printf ("SQLFreeHandle failed\n");
   printSQLError (2, m_hdbc);
}

retCode = SQLFreeHandle (SQL_HANDLE_ENV, m_henv);
if (retCode != SQL_SUCCESS && retCode != SQL_SUCCESS_WITH_INFO)
{
   printf ("SQLFreeHandle failed\n");
   printSQLError (1, m_henv);
}
}
```

5

Example: Application Migration Using SQL Translation Framework

Consider an example of migrating a Sybase JDBC Application, and the information contained in the migration reports: how it may be used to tune the migration for optimal results.

5.1 Migrating a Sybase JDBC Application

Figure 5-1 illustrates how an application that is coded to query a Sybase database may use SQL Translation Framework to query information stored in Oracle Database instead.

Figure 5-1   Sybase Application Running Against Oracle Database
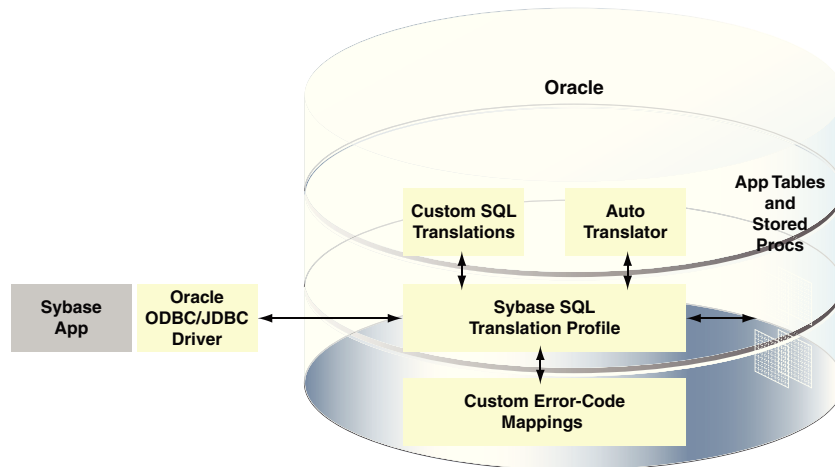
5.1.1 Application Overview

The Sybase database used in this example has three tables and five procedures and includes the following features:

- `IDENTITY` columns

- `INSERT` statements into tables with `IDENTITY` columns

- `VARCHAR` columns with size greater than `4000` characters

- Multiple implicit result sets returned from procedures

A Java application connects to this Sybase database using JDBC.

5.1.2 Setting Up Migration

The migration process has four phases - Capture, Convert, Generate, and Data Move. It is best practice to complete each phase of the migration process, review any issues on the Summary page, and then continue to the next phase. The Migration Wizard enables you to complete each step in turn and then return back to the wizard to complete further steps. To do this, after completing each phase, select the **Proceed to Summary Page** check box and click **Next**.

Perform the following steps to set up migration:

1. Download the JDBC driver JTDS 1.2.

2. Add JTDS as a third-party JDBC driver as follows:

    a. Select **Preferences** from the **Tools** menu.

    b. Select **Third Party JDBC Driver** from the **Database** option on the right panel, as shown in Figure 5-2.

Figure 5-2   Setting JTDS JDBC Driver
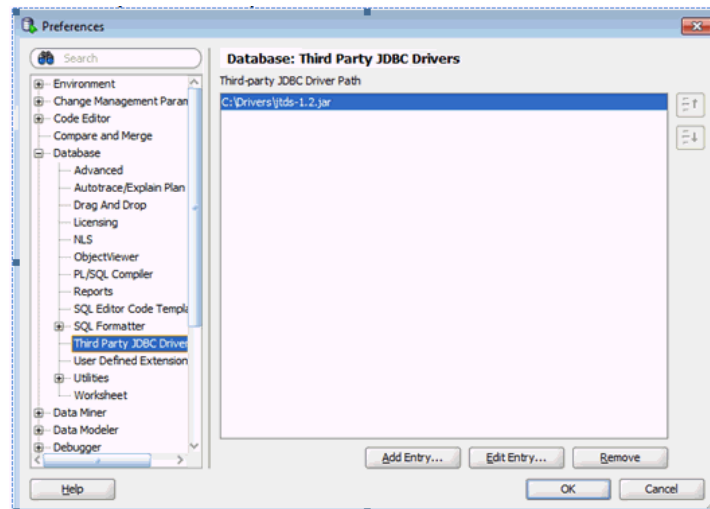
3. Click **Add Entry**.

    The Select Path Entry box is displayed.

4. Select the `jtds-1.2.jar` file and click **Select**.

5. Click **OK**.

6. Connect to the Oracle Database where you want to migrate the information.

7. Verify that the connection is using Oracle Database 12c JDBC drivers, with the following command:

    ```
    show jdbc
    ```

8. Create a new user `migrep` in Oracle database, for the migration repository, with the following command:

    ```
    GRANT CONNECT,RESOURCE,CREATE VIEW to migrep INDENTIFIED BY migrep;
    ALTER USER migrep QUOTA UNLIMITED to users;
    ```

9. Connect to the database as the `migrep` user and associate the migration repository with the user, as shown in Figure 5-3.

Figure 5-3   Associating a User with Migration Repository
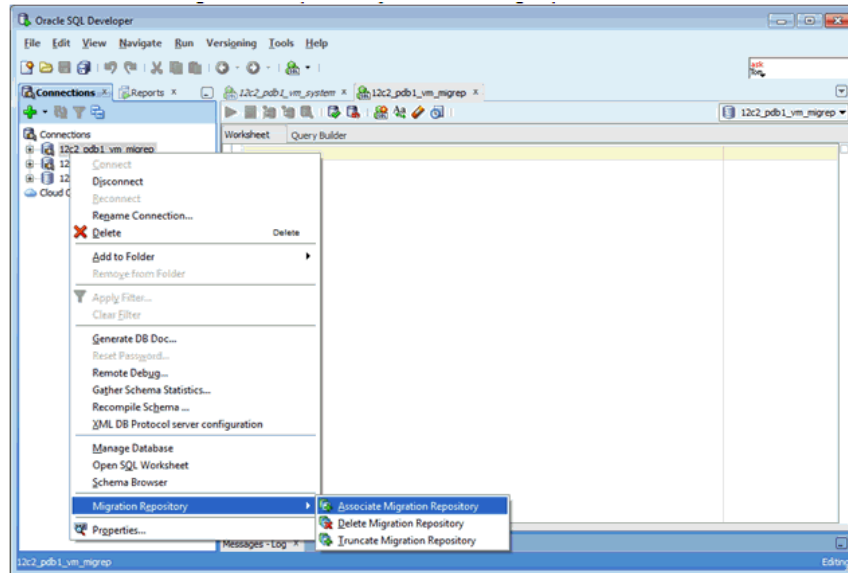
10. Create a connection to the Sybase database, in this example, `simpledemo12c`, as shown in Figure 5-4.

Figure 5-4   Creating a Connection to the Sybase Database
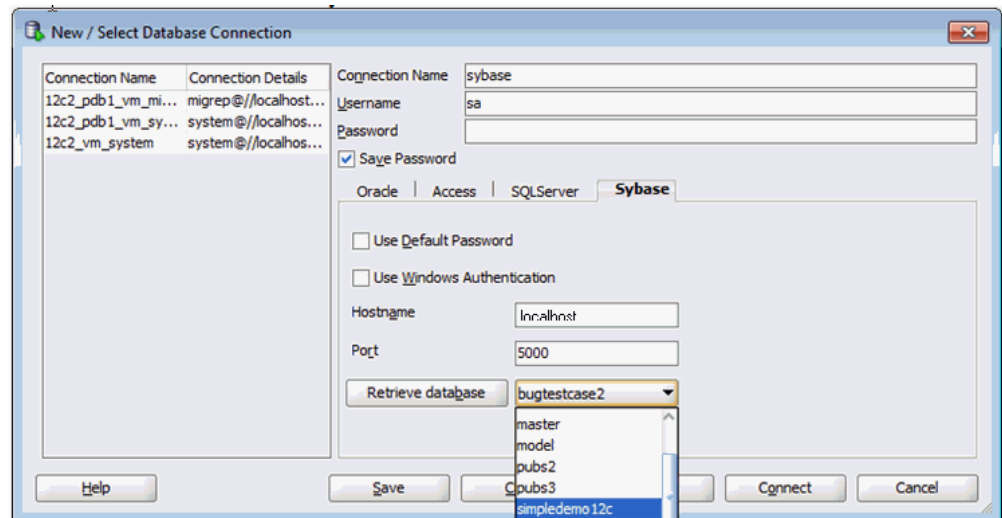
5.1.3 Capturing Migration

Perform the following steps to capture migration:

1. Right-click on the `simpledemo12c Sybase` database and select the **Migrate to Oracle** option, as shown in Figure 5-5.

Figure 5-5   Starting Capture Phase of Migration Process
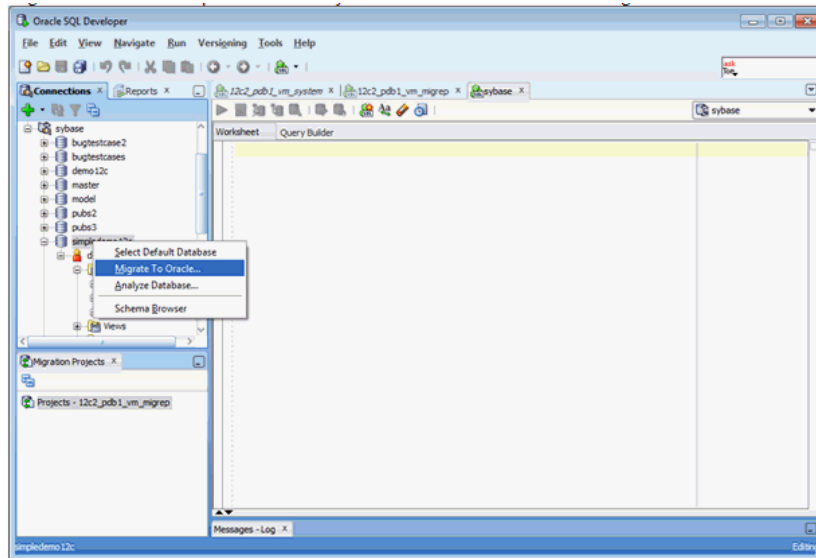
This opens the Migration Wizard, as shown in Figure 5-6.

Click **Next**.

Figure 5-6   Migration Wizard Introduction Screen
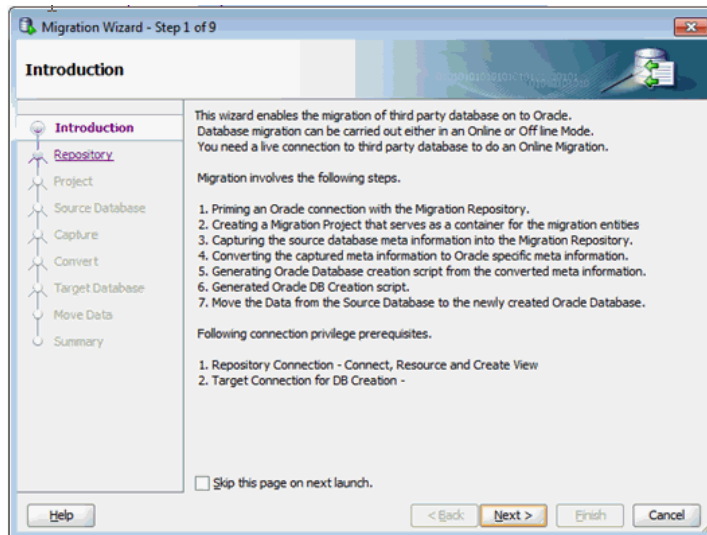
2. Choose the Migration Repository, as shown in Figure 5-7.

Click **Next**.

Figure 5-7   Choosing the Migration Repository
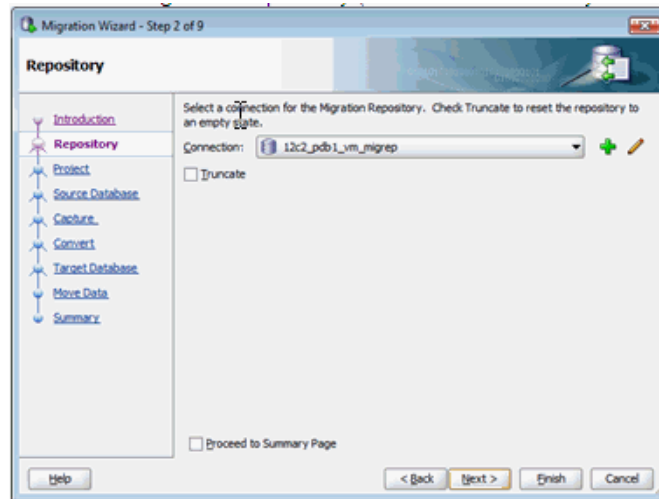
3. Enter a project name and specify an output directory to place files, as shown in Figure 5-8.

    Click **Next**.

Figure 5-8   Specifying Project Name and Output Directory
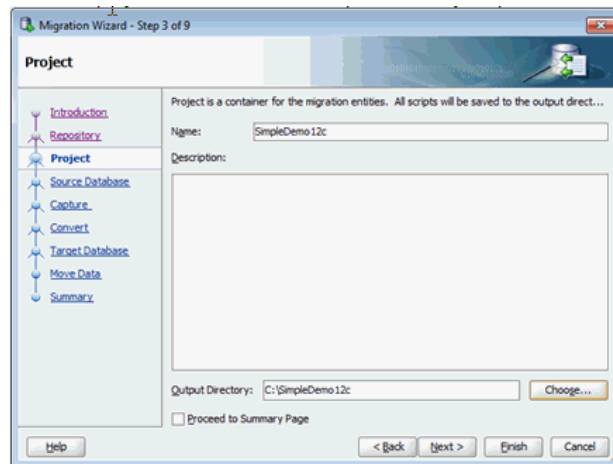
4. Select the database connection and the mode, as shown in Figure 5-9.

    Click **Next**.

Figure 5-9   Selecting the Database Connection and Mode
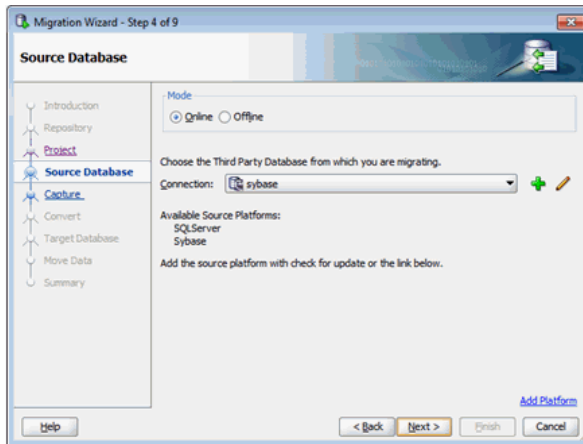
5. Select the database, in this case, `simpledemo12c`, by moving it from **Available Databases** to **Selected Databases**, as shown in Figure 5-10.

   Click **Proceed to Summary Page** to review the Capture phase before moving to the next phase of the migration process.

   Click **Next**.

Figure 5-10   Selecting the Database to be Migrated
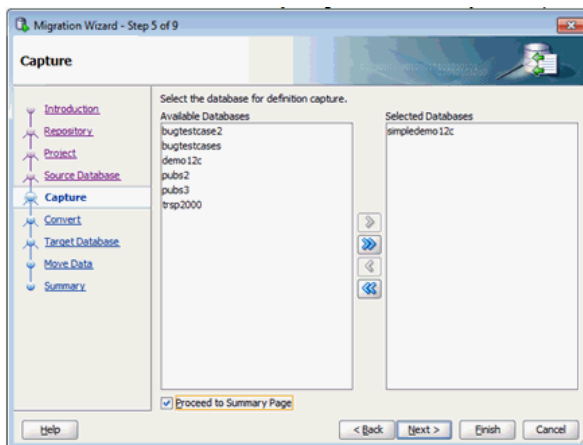
The capture phase saves a snapshot of the selected database at this point of time. Only the object definitions are captured, not the actual table data. This captured snapshot can be viewed in the Migration Projects navigator.

Note that the snapshot is not a connection to the database, and it only enables you to browse through the information saved in the Migration Repository.

5.1.4 Setting Migration Preferences

Before starting the conversion phase, you must set the migration preferences. Perform the following steps to achieve this:

1. From the Tools menu, select **Preferences**, then **Migration**, and then **Translators**. Select the **Generate Compound Triggers** option.

Figure 5-11   Setting Migration Preferences
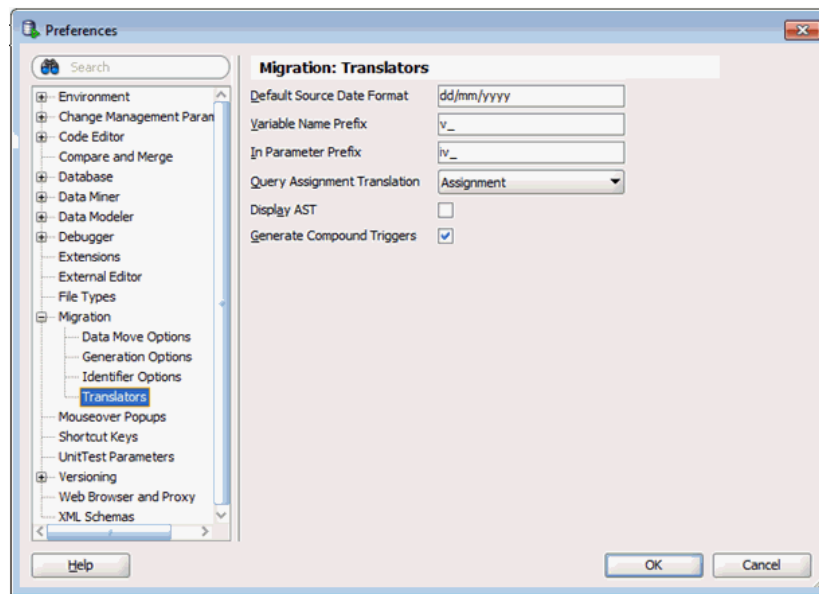
2. From the Tools menu, select **Preferences**, then **Migration**, and then **Generation Options**. Select the **Use all Oracle Database 12c features in Migration** option.

Figure 5-12   Setting Migration Preferences
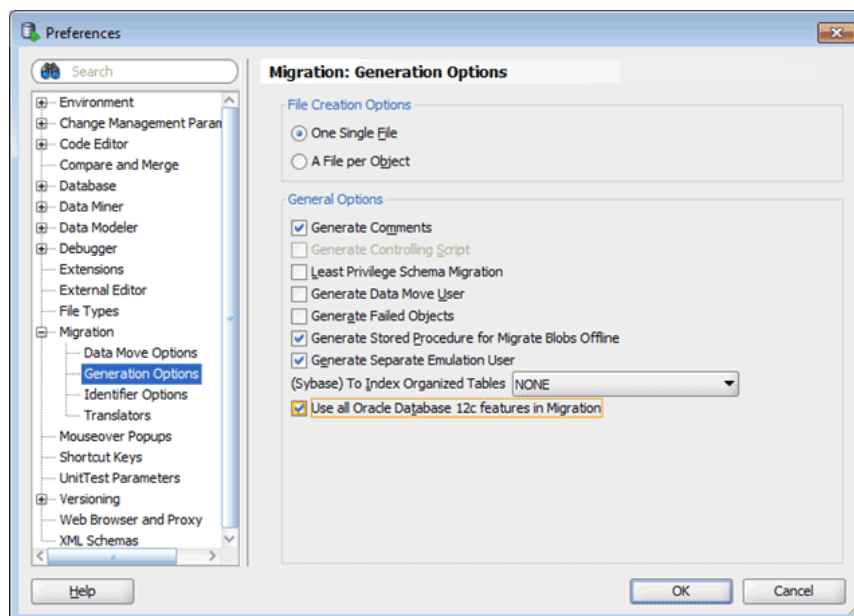
5.1.5 Converting Migration

Perform the following steps to start convert phase of the migration process:

1. Right-click the Capture Model node and choose **Convert**, as shown in Figure 5-13.

Figure 5-13   Starting Convert Phase of Migration Process
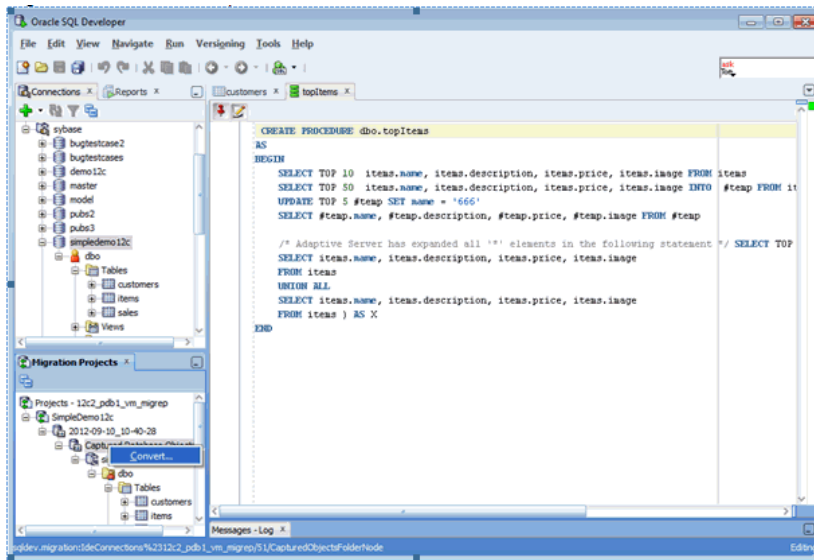
The Migration Wizard is opened at the Convert phase, as shown in Figure 5-14.

Figure 5-14   Converting the Migrated Data
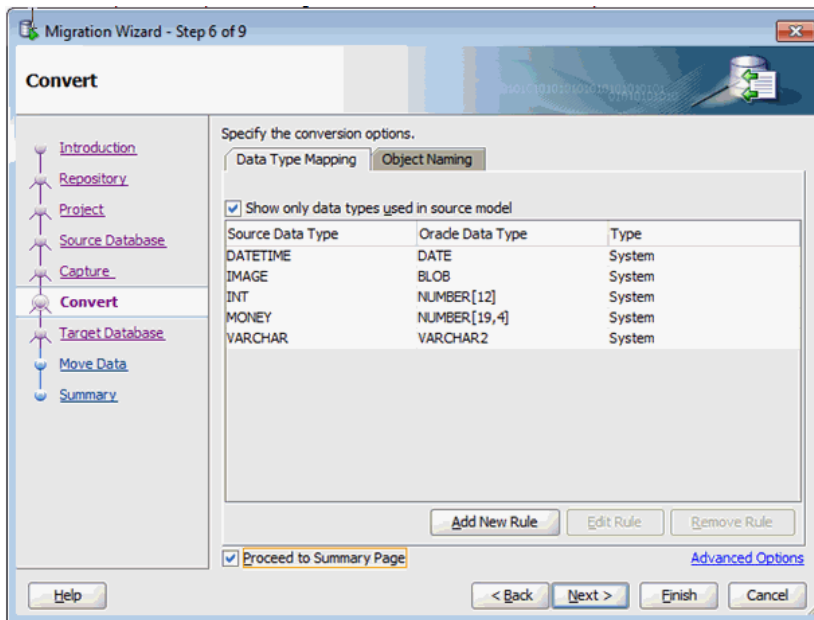
2. Select Proceed to Summary Page and click **Next**.

3. Click **Finish**.

During the convert phase, object names are resolved to valid Oracle names. Data types are converted to Oracle Database types and T-SQL defined objects like stored procedures, views, and so on are converted to Oracle PL/SQL. A converted model is created that can be browsed in the Migration Projects navigator. The converted procedures can be reviewed in the converted model.

Note that the converted model is not an actual Oracle database, but a prototype of an Oracle Database. The information is still stored only in the Migration Repository tables.

5.1.6 Generating a Migration

The migration generation phase creates the objects in the target Oracle Database. A script is created and it is run against a selected Oracle connection in the following two ways:

- In `offline` mode, the script is opened in a SQL Worksheet and you have to select the connection and run it manually.

- In `online` mode, you must provide the target connection in the wizard and the wizard runs the script automatically.

The following steps demonstrate how to perform the generate phase of the migration process in `offline` mode:

1. Right-click on **Converted Database Objects** in the Migration Projects panel and select **Generate Target**.

2. Select `offline` as the database mode in the Migration Wizard, as shown in Figure 5-15.

   Click **Next**.

Figure 5-15   Selecting the Database Mode
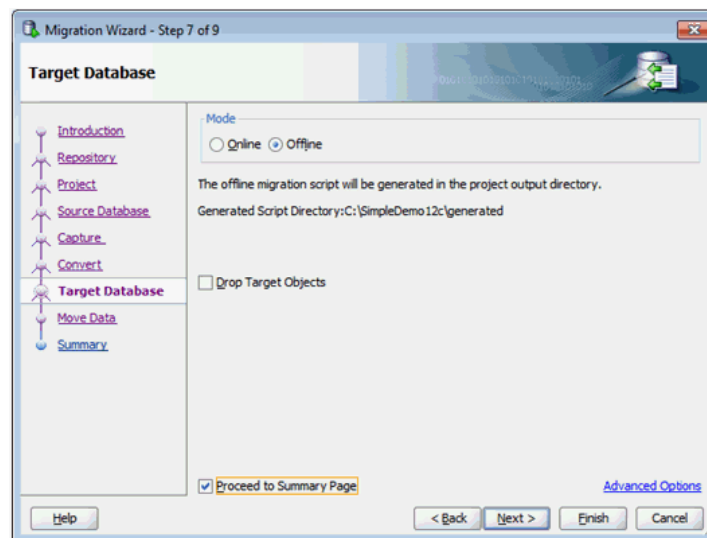
3. Choose a connection in the target Oracle Database, as shown in Figure 5-16.

Figure 5-16   Creating Oracle Database Connection for Target User dbo_simpledemo12c
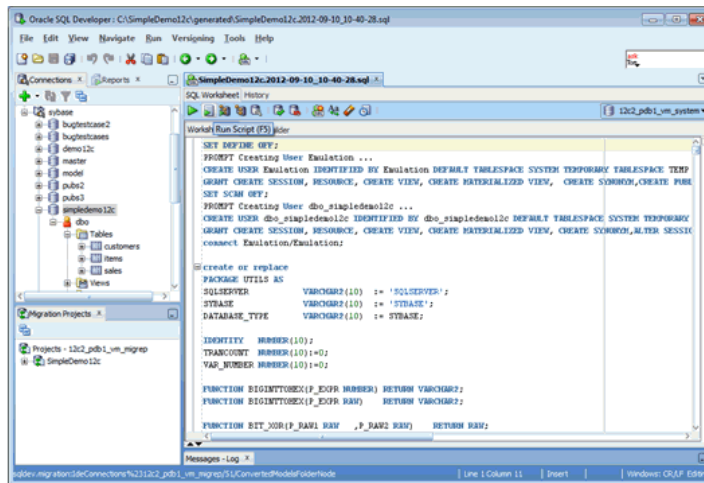
The database objects are not created under the connection selected in this step. However, this connection *must* have enough privileges to create other users and objects.

5.1.6.1 Creating a Target Oracle User

Create a connection to the newly created user (described in step 3), as shown in Figure 5-17. At this point, the Sybase database objects are migrated to Oracle Database, but the data is not migrated till now.

Figure 5-17   Targeting an Oracle User
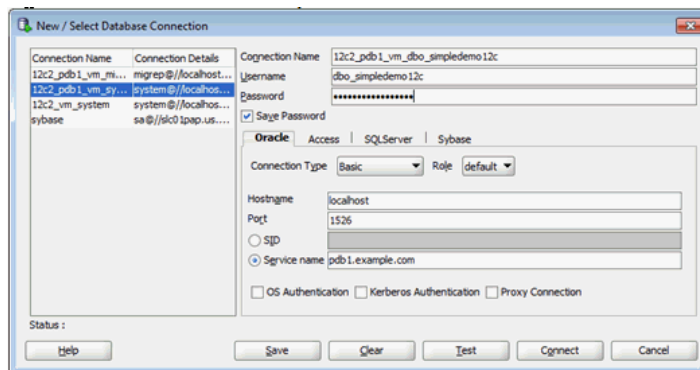
5.1.7 Moving the Data

Perform the following steps to move the data to Oracle Database:

1. Right-click the **Converted Database Objects** node and select **Move Data**, as shown in Figure 5-18.

   Click **Next**.

Figure 5-18   Moving the Data from Sybase Database to Oracle Database
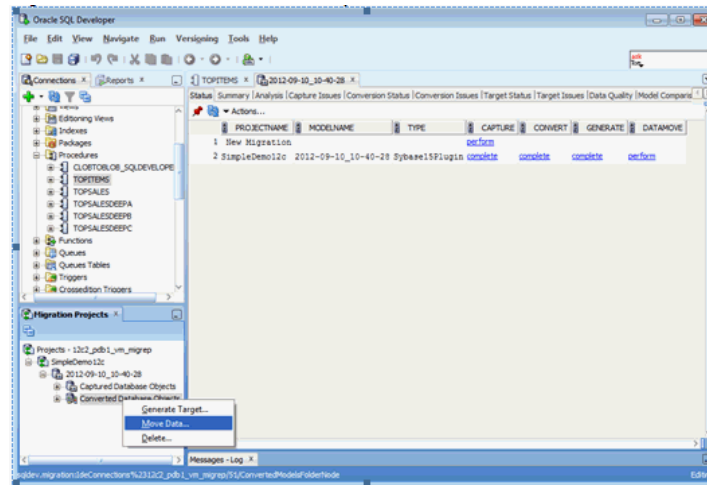
2. Select `online` as the data move mode in the Move Data screen.

   You can select `offline` as the data move mode if the migration process involves large amount of data.

3. Click **Next**. The Summary screen appears.

4. Click **Finish**.

   You can browse the database objects to verify the data is moved to Oracle database.

> ✎ **See Also:**
>
> *Oracle SQL Developer User's Guide*

5.2 Generating Migration Reports

Oracle SQL Developer provides a number of reports on the migration process to help identify tasks and issues to resolve. Click or double-click on the migrated project in the Migration Projects navigator. A report will appear on the right panel with a number of tabs and children reports, as shown in Figure 5-19.

Figure 5-19   Generating Migration Reports
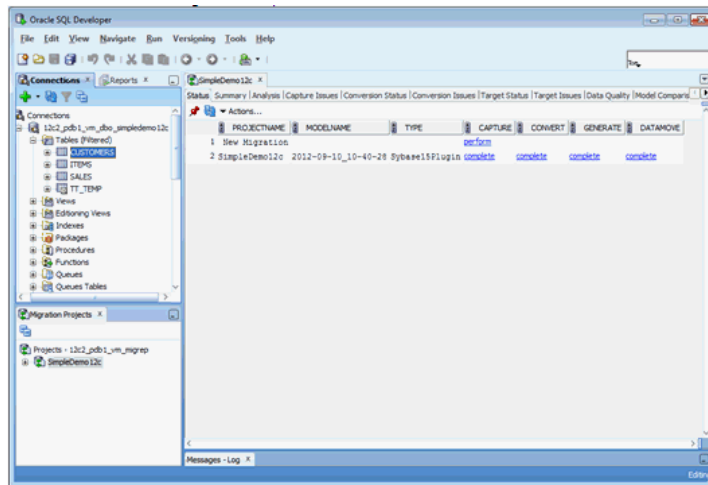
The Analysis report provides information about the size of the migrated database like the number of objects, line sizes, and so on, as shown in Figure 5-20.

Figure 5-20   Migration Analysis Report
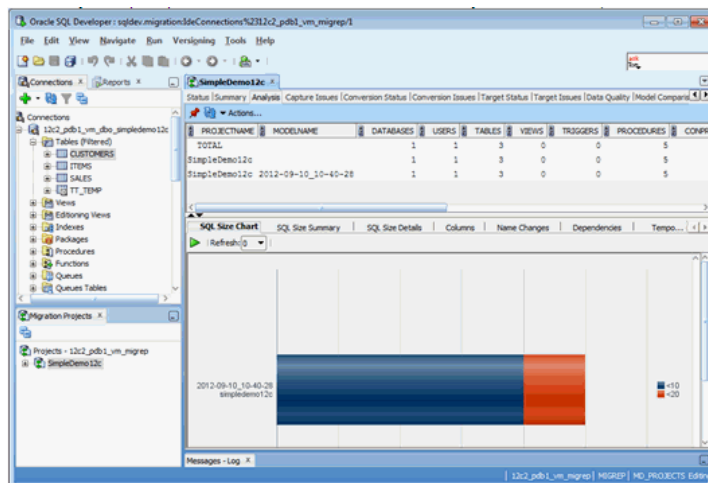
The Target Status report provides information about the status of the migrated objects in the Target database. First, select a target connection with enough privileges to view the status of other schema objects and then select **refresh**. Objects that are present in the converted model, but are missing from the target Oracle Database, are listed as missing. These objects can be either valid or invalid.

Figure 5-21   Target Status Report

The Data Quality tab provides information about the number of rows in the target Oracle Database compared with the source database. Perform the following steps to compare the databases:

1. Select a converted model, a source connection, and a target connection.

2. Click **Analyse**.

3. Click **Refresh**.

   This performs a count(*) function on each table in the source and the target database. So, it is advisable not to perform this operation on production data.

ORACLE

6

MySQL Client Library Driver for Oracle

Consider the specifics of MySQL Client Library Driver for Oracle Database, and its use in migrating applications from MySQL to Oracle.

> ✎ **See Also:**
>
> API Reference for Oracle MySQL Client Library Driver for more information about MySQL programmatic support

6.1 Introduction to MySQL Client Library Driver for Oracle

MySQL Client Library Driver for Oracle Database 12*c*, `liboramysql`, is a drop-in replacement for MySQL Commercial Connector/C 6.0 client library. The `liboramysql` driver implements a similar API, enabling C-based applications and tools developed for MySQL to connect to Oracle Database. The driver may be used to migrate applications from MySQL to Oracle Database with minimal changes to the application code.

The `liboramysql` driver uses Oracle Call Interface (OCI) to connect to Oracle Database.

Figure 6-1   MySQL Application Code Using liboramysql Driver to Connect to Oracle
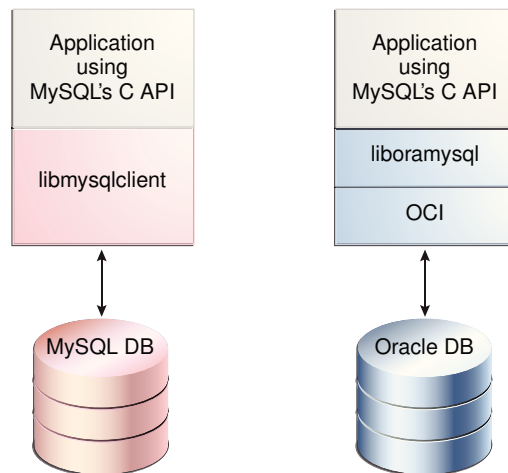
The C code snippet in Example 6-1 demonstrates how to connect to MySQL and how to insert a row into a table. After updating the connection credentials, this code can run unchanged against Oracle Database when the executable is linked using the `liboramysql` library, instead of the `libmysqlclient` library.

Although the database schema and data must be migrated to Oracle separately, and although the `liboramysql` library does not translate SQL statements, considerable amount of effort is conserved when migrating to Oracle Database because no changes have to be made to the application code.

Custom C applications can use the `liboramysql` library to easily migrate to Oracle Database.

Additionally, you can migrate applications using programming languages that abstract the use of the `libmysqlclient` library and provide MySQL extensions or adapters. These languages include PHP, Perl, Python, and Ruby. Although native Oracle adapters already exist for many programming languages implemented in C, migrating an application to a native Oracle adapter often requires extensive application code changes.

6.1.1 Connecting to MySQL

Example 6-1   Connecting to MySQL and Inserting a New Row

```
c = mysql_init(NULL);
mysql_real_connect(c, "myhost", "myun", "mypw", "mydb", 0, NULL, 0);
mysql_query(c, "insert into mytable values (1,2)");
mysql_close(c);
```

6.2 Installation and First Use of MySQL Client Library Driver for Oracle

The MySQL Client Library Driver for Oracle is provided as a file in the `liboramysql.so` shared library for Linux and as the `oramysql.dll` dynamic link library (DLL) for Windows. The driver is also packaged as part of the Oracle Instant Client `Basic` and `Basic Lite` packages for download from OTN. See http://www.oracle.com/technetwork/topics/linuxsoft-082809.html and http://www.oracle.com/technetwork/topics/winsoft-085727.html.

The driver must be installed in the same directory as the Oracle Client Shared Library, that is, `libclntsh.so` for Linux and `oci.dll` for Windows. Typically, you must set the operating system environment variable (`LD_LIBRARY_PATH` on Linux or `PATH` on Windows) to include this installation directory.

For `ORACLE_HOME` installations, the driver library is installed in the `$ORACLE_HOME/lib` directory for Linux and the `%ORACLE_HOME%\bin` directory for Windows. For Instant Client ZIP files, the library is in the `instantclient_12_1` directory. For Instant Client RPM installations, the library is in the `/usr/lib/oracle/12.1/client/lib` or `/usr/lib/oracle/12.1/client64/lib` directory on 32-bit and 64-bit Linux platforms, respectively.

6.3 Overview of Migration with MySQL Client Library Driver for Oracle

Migrating a C-based MySQL application to Oracle Database involves the following steps:

1. **Confirm that the application runs against MySQL Database.**

    This ensures that the migration process starts at a known baseline of functionality.

2. **Replace the `libmysqlclient` library with the `liboramysql` library**.

   The application must be relinked to use the `liboramysql` library instead of the `libmysqlclient` library.

3. **Migrate the application schema to Oracle Database.**

   The schema must be migrated to use Oracle DDL and types. Oracle SQL Developer assists in this process.

   See *Oracle SQL Developer User's Guide* for further details.

4. **Review all SQL statements used by the application.**

   If necessary, change the SQL statements of the application to use Oracle syntax, or implement a SQL Translator to automatically perform the conversion at application run time. Rewrite any logic that depends on MySQL features that are not supported by Oracle Database.

   See SQL Translation of JDBC and ODBC Applications .

5. **Update the connection string of the application to connect to Oracle Database.**

   Use Oracle Easy Connect syntax or a `tnsnames.ora` connect identifier in the *host* parameter of the connection call.

6. **Test the application with Oracle Database.**

   Verify the application against Oracle Database.

6.4 Using MySQL Client Library Driver for Oracle

The `liboramysql` API is compatible with MySQL Commercial Connector/C 6.0. MySQL Driver for Oracle Database, liboramysql, translates MySQL API calls to Oracle Call Interface (OCI) calls, and between Oracle and MySQL data types.

Existing MySQL-based applications may be relinked to use the `liboramysql` driver, making Oracle Database the new data source. Note that the `liboramysql` driver supports connections only to Oracle Database. Simultaneous connections to both MySQL Database and Oracle Database in the same application are not possible.

See API Reference for Oracle MySQL Client Library Driver for details on data type mapping and API compatibility. Additional information may also be found in *Oracle SQL Developer User's Guide*.

The `liboramysql` driver does not translate SQL statements. You must rewrite the statements that are not valid for Oracle Database. You can do this directly in the application, or by using a SQL Translator. The application schema and data must also be migrated separately. Oracle SQL Developer automates this process.

Whenever cross-version OCI connectivity exists for older versions of Oracle Database, you can use the `liboramysql` driver to connect to these older versions.

6.4.1 Relinking the Application with the liboramysql Driver

The fundamental step of using the `liboramysql` library is to relink the application to use the new library. The `liboramysql` library is compatible with the `libmysqlclient.so` library from MySQL Commercial Connector/C 6.0.2 package, so you must build and verify version-sensitive applications with MySQL Commercial Connector/C 6.0.2 before migrating to Oracle Database.

The installation scripts of public software compiled from source code typically expect MySQL components to follow a predefined system directory structure. You can use the `setuporamysql.sh` script in the `demo` directory of Instant Client SDK to achieve this.

Depending on the application, you can use one or more of the following ways to relink the application with the `liboramysql` library:

- **Build directly with the `liboramysql` library.**

  You can update your build scripts to use the `liboramysql` library and build custom applications directly with this Oracle library.

- **Use the `liboramysql` library to emulate a MySQL Commercial Connector/C directory**

  The `setuporamysql.sh` library in the Instant Client SDK shows how a directory structure emulating a MySQL Commercial Connector/C installation can be created. You may build applications using this emulated directory.

- **Use the `LD_PRELOAD` environment variable**.

  Preconfigured programs may be able to use the `LD_PRELOAD` environment variable to link with the `liboramysql` library. However, changing the value of this environment variable may not work if the program uses the `dlopen()` method.

- **Duplicate the `liboramysql` library.**

  Perform the following steps to rename the `liboramysql` library to the MySQL client library name used by the application:

  1. Use the `ldd` command to identify the MySQL library with which the application is linked:

     ```
     $ ldd yourprogram
     ...
     libmysqlclient.so.16 => /usr/lib/libmysqlclient.so.16 (0x00007f9004e7f000)
     ...
     ```

  2. Create the following symbolic link as the Oracle software owner user:

     ```
     $ ln -s $ORACLE_HOME/lib/liboramysql12.so $ORACLE_HOME/lib/libmysqlclient.so.16
     ```

  3. Add `$ORACLE_HOME/lib` to the `LD_LIBRARY_PATH` environment variable for any application that formerly used the `libmysqlclient` library:

     ```
     $ export LD_LIBRARY_PATH=$ORACLE_HOME/lib
     ```

- **Replace the system MySQL client library.**

  Rename the target system MySQL client library and link the new library in its place. Because this option affects every application on the system that uses MySQL, and should be done only if absolutely necessary.

  ```
  # mv /usr/lib64/libmysqlclient.so.16 /usr/lib64/libmysqlclient.so.16.backup
  # ln -s $ORACLE_HOME/lib/liboramysql12.so /usr/lib64/libmysqlclient.so.16
  ```

If MySQL applications are not rebuilt from the source code, then you must first link the applications against the `libmysqlclient.so` library from MySQL Commercial Connector/C 6.0.2 package. This ensures binary compatibility with the data structures in the `liboramysql` library.

6.4.2 Connecting to Oracle Database

To connect to Oracle Database with the `liboramysql` library, use Oracle Easy Connect syntax or a `tnsnames.ora` connect identifier in the *host* parameter of the connection call:

```
mysql_real_connect(c, "localhost/pdborcl", "myun", "mypw", NULL, 0, NULL, 0);
```

6.4.3 Supported Platforms

MySQL Client Library Driver for Oracle is available on platforms that support the Oracle Instant Client.

See the list of supported platforms on the Oracle Support Certification site: `https://support.oracle.com`

6.4.4 Error Handling

All errors generated by OCI client code or the Oracle server are passed to the application when either the `mysql_errno()` method or the `mysql_error()` method is invoked after an error.

6.4.5 Globalization

The date format expected by the application may be set using `NLS_DATE_FORMAT` environment variable of Oracle Database, or changed with the equivalent `ALTER SESSION` command after connecting. The `NLS_DATE_FORMAT` environment variable is only used if `NLS_LANG` is also set in the environment.

6.4.6 Expected Differences

Some APIs in the `liboramysql` library necessarily return different results because of the underlying differences between MySQL Database and Oracle Database. Existing applications that use these APIs may require logic changes. For details of these differences, see API Reference for Oracle MySQL Client Library Driver .

7

API Reference for Oracle MySQL Client Library Driver

Consider the APIs that support migration from MySQL, the mapping of data types, support for specific MySQL APIs within Oracle, and error handling for migrated applications..

For documentation of MySQL C APIs, refer to MySQL 5.5 documentation.

7.1 Mapping Data Types

Oracle database types are described in the Internal Data Types section of *Oracle Call Interface Programmer's Guide*.

MySQL data types are fully described in MySQL documentation.

MySQL C APIs use `MYSQL_TYPE_` symbols to process data to and from MySQL database. These type symbols are mapped to MySQL data types in the server.

For instance, `MYSQL_TYPE_VAR_STRING` is mapped to `VARCHAR` in the server.

7.1.1 Mapping Oracle Data Types to MySQL Data Types

This table shows the value of the type field in `MYSQL_FIELD` parameter returned from `mysql_fetch_field_*` calls. The Oracle database type is mapped to a MySQL C API data type.

For example: A `VARCHAR2` column is represented by `MYSQL_TYPE_VAR_STRING`.

It is recommended that users use this table when migrating MySQL applications to Oracle. The MySQL Client Library driver for Oracle will perform Data type conversions between MySQL and Oracle.

Table 7-1   Mapping Oracle Data Types to MySQL Data Types

Oracle Data Type	Maps to MySQL Data Type
CHAR	MYSQL_TYPE_VAR_STRING
NCHAR	MYSQL_TYPE_VAR_STRING
NVARCHAR2	MYSQL_TYPE_VAR_STRING
VARCHAR2	MYSQL_TYPE_VAR_STRING
NUMBER	MYSQL_TYPE_NEWDECIMAL
LONG	MYSQL_TYPE_BLOB
CLOB	MYSQL_TYPE_BLOB
NCLOB	MYSQL_TYPE_BLOB
DATE	MYSQL_TYPE_DATETIME

Table 7-1   (Cont.) Mapping Oracle Data Types to MySQL Data Types

Oracle Data Type	Maps to MySQL Data Type
RAW	MYSQL_TYPE_VAR_STRING
BLOB	MYSQL_TYPE_BLOB
LONG RAW	MYSQL_TYPE_BLOB
ROWID	MYSQL_TYPE_VAR_STRING
UROWID	MYSQL_TYPE_VAR_STRING
BINARY FLOAT	MYSQL_TYPE_FLOAT
BINARY DOUBLE	MYSQL_TYPE_DOUBLE
User-defined type (object type, VARRAY, Nested Table)	Not supported
REF	Not supported
BFILE	MYSQL_TYPE_BLOB
TIMESTAMP	MYSQL_TYPE_DATETIME
TIMESTAMP WITH TIME ZONE	MYSQL_TYPE_DATETIME
TIMESTAMP WITH LOCAL TIME ZONE	MYSQL_TYPE_DATETIME
INTERVAL YEAR TO MONTH	MYSQL_TYPE_VAR_STRING
INTERVAL DAY TO SECOND	MYSQL_TYPE_VAR_STRING

7.1.2 Data Type Conversions for MySQL Program Variable Data Types

The calls to mysql_stmt_bind_param() and mysql_stmt_bind_result() may be used to convert between C program variables and database column values. Similarly, OCI provides rich conversion support from server data types to many client data types.

Input conversions from a C program value to a database column value are handled by invoking mysql_stmt_bind_param(). Output to a C program value is handled through a call to mysql_stmt_bind_result().

Table 7-2 summarizes viable conversions between MySQL program variable data types and Oracle column data types. The possible values in the table are:

- I: input conversion is supported
- O: output conversion is supported
- I/O: both input and output conversion is supported
- -: conversion is not supported.

Be sure to read the corresponding notes for each data type before finalizing conversion choices.

Table 7-2   Converting MySQL Program Variable Data Types to Oracle Column Data Types

MySQL Program Variable Data Types	CHAR	VARCHAR2	NUMBER	LONG	ROWID	UROWID	DATE	RAW	LONG RAW
MYSQL_TYPE_TINY	I/O	I/O	I/O	I	-	-	-	-	-
MYSQL_TYPE_SHORT	I/O	I/O	I/O	I	-	-	-	-	-
MYSQL_TYPE_LONG	I/O	I/O	I/O	I	-	-	-	-	-
MYSQL_TYPE_LONGLONG	I/O	I/O	I/O	I	-	-	-	-	-
MYSQL_TYPE_FLOAT	I/O	I/O	I/O	I	-	-	-	-	-
MYSQL_TYPE_DOUBLE	I/O	I/O	I/O	I	-	-	-	-	-
MYSQL_TYPE_DATE	I/O	I/O	-	I	-	-	I/O	-	-
MYSQL_TYPE_TIME	I/O	I/O	-	I	-	-	I/O	-	-
MYSQL_TYPE_DATETIME	I/O	I/O	-	I	-	-	I/O	-	-
MYSQL_TYPE_TIMESTAMP	I/O	I/O	-	I	-	-	I/O	-	-
MYSQL_TYPE_STRING	I/O	I/O	I/O	I/O	I/O	I/O	I/O	I/O	I/O
MYSQL_TYPE_VAR_STRING	O	O	O	O	O	O	O	O	O
MYSQL_TYPE_BLOB	I/O	I/O	-	I/O	-	-	-	I/O	I/O
MYSQL_TYPE_TINY_BLOB	O	O	-	O	-	-	-	O	O
MYSQL_TYPE_MEDIUM_BLOB	O	O	-	O	-	-	-	O	O
MYSQL_TYPE_LONG_BLOB	O	O	-	O	-	-	-	O	O
MYSQL_TYPE_NEWDECIMAL	O	O	O	-	-	-	-	-	-

7.1.2.1 MYSQL_TYPE_BLOB

- `CHAR` and `VARCHAR2`: Conversion is valid for input or output. On input, column value is stored in hexadecimal format.
- `LONG`: Conversion is valid for input or output. On input, column value is stored in hexadecimal format.
- `RAW`: Conversion is valid for input or output.
- `LONG RAW`: Conversion is valid for input or output.
- Conversion is not supported for `NUMBER`, `ROWID`, `UROWID`, and `DATE`.

7.1.2.2 MYSQL_TYPE_DATE

- `CHAR` and `VARCHAR2`: Conversion is valid for input or output. For input, host string must be in Oracle `DATE` character format. For output, column value is returned in Oracle `DATE` format.

- `DATE`: Conversion is valid for input or output.
- `LONG`: Conversion valid for input to database column value.
- Conversion not supported for `NUMBER`, `ROWID`, `UROWID`, `RAW`, and `LONG RAW`.

7.1.2.3 MYSQL_TYPE_DATETIME

- `CHAR` and `VARCHAR2`: Conversion is valid for input or output. For input, host string must be in Oracle `DATE` character format. For output, column value is returned in Oracle `DATE` format.
- `DATE`: Conversion is valid for input or output.
- `LONG`: Conversion valid for input to database column value.
- Conversion not supported for `NUMBER`, `ROWID`, `UROWID`, `RAW`, and `LONG RAW`.

7.1.2.4 MYSQL_TYPE_DOUBLE

- `CHAR` and `VARCHAR2`: Conversion is valid for input or output. For output, column value must represent a valid number.
- `NUMBER`: Conversion is valid for input or output.
- `LONG`: Conversion valid for input to database column value.
- Conversion not supported for `ROWID`, `UROWID`, `DATE`, `RAW`, and `LONG RAW`.

7.1.2.5 MYSQL_TYPE_FLOAT

- `CHAR` and `VARCHAR2`: Conversion is valid for input or output. For output, column value must represent a valid number.
- `NUMBER`: Conversion is valid for input or output.
- `LONG`: Conversion valid for input to database column value.
- Conversion not supported for `ROWID`, `UROWID`, `DATE`, `RAW`, and `LONG RAW`.

7.1.2.6 MYSQL_TYPE_LONG

- `CHAR` and `VARCHAR2`: Conversion is valid for input or output. For output, column value must represent a valid number.
- `NUMBER`: Conversion is valid for input or output.
- `LONG`: Conversion valid for input to database column value.
- Conversion not supported for `ROWID`, `UROWID`, `DATE`, `RAW`, and `LONG RAW`.

7.1.2.7 MYSQL_TYPE_LONG_BLOB

- `CHAR`, `VARCHAR2`, `LONG`, `RAW`, and `LONG RAW`: Conversion is valid for output.
- Conversion is not supported for `NUMBER`, `ROWID`, `UROWID`, and `DATE`.

7.1.2.8 MYSQL_TYPE_LONGLONG

- `CHAR` and `VARCHAR2`: Conversion is valid for input or output. For output, column value must represent a valid number.

- NUMBER: Conversion is valid for input or output.

- LONG: Conversion valid for input to database column value.

- Conversion not supported for ROWID, UROWID, DATE, RAW, and LONG RAW.

7.1.2.9 MYSQL_TYPE_MEDIUM_BLOB

- CHAR, VARCHAR2, LONG, RAW, and LONG RAW: Conversion is valid for output.

- Conversion is not supported for NUMBER, ROWID, UROWID, and DATE.

7.1.2.10 MYSQL_TYPE_NEWDECIMAL

- CHAR and VARCHAR2: Conversion is valid for output. Column value must represent a valid number.

- NUMBER: Conversion is valid for output to C program value.

- Conversion is not supported for LONG, ROWID, UROWID, DATE, RAW, and LONG RAW.

7.1.2.11 MYSQL_TYPE_SHORT

- CHAR and VARCHAR2: Conversion is valid for input or output. For output, column value must represent a valid number.

- NUMBER: Conversion is valid for input or output.

- LONG: Conversion valid for input to database column value.

- Conversion not supported for ROWID, UROWID, DATE, RAW, and LONG RAW.

7.1.2.12 MYSQL_TYPE_STRING

- CHAR and VARCHAR2: Conversion is valid for input or output.

- NUMBER: Conversion is valid for input or output. For input, the host string must represent a valid number.

- LONG: Conversion valid for input or output.

- ROWID: Conversion is valid for input or output. For input, the host string must be in Oracle ROWID format. For output, column value is returned in Oracle ROWID format.

- UROWID: Conversion is valid for input or output. For input, the host string must be in Oracle UROWID format. For output, column value is returned in Oracle UROWID format.

- DATE: Conversion is valid for input or output. For input, host string must be in Oracle DATE character format. For output, column value is returned in Oracle DATE format.

- RAW: Conversion is valid for input or output. For input, host string must be in hexadecimal format.

- LONG RAW: Conversion is valid for input or output. For input, host string must be in hexadecimal format.

7.1.2.13 MYSQL_TYPE_TIME

- `CHAR` and `VARCHAR2`: Conversion is valid for input or output. For input, host string must be in Oracle `DATE` character format. For output, column value is returned in Oracle `DATE` format.

- `DATE`: Conversion is valid for input or output.

- `LONG`: Conversion valid for input to database column value.

- Conversion not supported for `NUMBER`, `ROWID`, `UROWID`, `RAW`, and `LONG RAW`.

7.1.2.14 MYSQL_TYPE_TIMESTAMP

- `CHAR` and `VARCHAR2`: Conversion is valid for input or output. For input, host string must be in Oracle `DATE` character format. For output, column value is returned in Oracle `DATE` format.

- `DATE`: Conversion is valid for input or output.

- `LONG`: Conversion valid for input to database column value.

- Conversion not supported for `NUMBER`, `ROWID`, `UROWID`, `RAW`, and `LONG RAW`.

7.1.2.15 MYSQL_TYPE_TINY

- `CHAR` and `VARCHAR2`: Conversion is valid for input or output. For output, column value must represent a valid number.

- `NUMBER`: Conversion is valid for input or output.

- `LONG`: Conversion valid for input to database column value.

- Conversion not supported for `ROWID`, `UROWID`, `DATE`, `RAW`, and `LONG RAW`.

7.1.2.16 MYSQL_TYPE_TINY_BLOB

- `CHAR`, `VARCHAR2`, `LONG`, `RAW`, and `LONG RAW`: Conversion is valid for output.

- Conversion is not supported for `NUMBER`, `ROWID`, `UROWID`, and `DATE`.

7.1.2.17 MYSQL_TYPE_VAR_STRING

- `CHAR` and `VARCHAR2`: Conversion is valid for output to C program value.

- `NUMBER`: Conversion is valid for output to C program value.

- `LONG`: Conversion is valid for output to C program value.

- `ROWID`: Conversion is valid for output to C program value. On output, column value is returned in Oracle `ROWID` format.

- `UROWID`: Conversion is valid for output to C program value. On output, column value is returned in Oracle `UROWID` format.

- `DATE`: Conversion is valid for output to C program value. On output, column value is returned in Oracle `DATE` format.

- `RAW`: Conversion is valid for output to C program value.

- `LONG RAW`: Conversion is valid for output to C program value.

7.1.3 Data Type Conversions for MySQL External Data Types (LOB Data Type Descriptors)

The external data types Table 7-3 may be converted to the specified Oracle internal data types.

Table 7-3   Data Type Conversions for LOB Data Type Descriptors

MySQL External Data Types	ORACLE INTERNAL CLOB/NCLOB	ORACLE INTERNAL BLOB
MYSQL_TYPE_BIT	I/O	I/O
MYSQL_TYPE_STRING	I/O	I/O
MYSQL_TYPE_VAR_STRING	O	O
MYSQL_TYPE_BLOB	I/O	I/O
MYSQL_TYPE_TINY_BLOB	O	O
MYSQL_TYPE_MEDIUM_BLOB	O	O
MYSQL_TYPE_LONG_BLOB	O	O

7.1.4 Data Type Conversions for Datetime and Interval Data Types

When working with a DATETIME or INTERVAL columns, it is possible to use one of the character data types to define a host variable used in a FETCH or INSERT operation The driver automatically converts between the character data type and DATETIME or INTERVAL data type.

Table 7-4 lists external data types that may be converted to the specified internal Oracle data types.

Table 7-4   Data Conversions for Datetime and Internal Data Type

External/Internal Types	VARCHAR, CHAR	DATE	TS	TSTZ	TSLTZ	INTERVAL YEAR TO MONTH	INTERVAL DAY TO SECOND
MYSQL_TYPE_STRING	I/O	I/O	I/O	I/O	I/O	I/O	I/O
MYSQL_TYPE_VAR_STRING	O	O	O	O	O	O	O
MYSQL_TYPE_DATE	I/O	I/O	I/O	I/O	I/O	-	-
MYSQL_TYPE_TIME	I/O	I/O	I/O	I/O	I/O	-	-
MYSQL_TYPE_DATETIME	I/O	I/O	I/O	I/O	I/O	-	-
MYSQL_TYPE_TIMESTAMP	I/O	I/O	I/O	I/O	I/O	-	-

7.2 Error Handling

All errors generated by OCI or Oracle server pass to the application when methods mysql_errno() or mysql_error() are invoked after an error. The application receives an

Oracle-specific error. Oracle error messages are more specific then MySQL error codes, and are therefore more pertinent to resolving the error condition.

The errors that are generated by the driver itself are in an error range reserved for the MySQL driver in the OCI error space.

The mysql_sqlstate() call attempts to map the error to the appropriate SQLSTATE whenever possible. In most cases, it returns HY000, which corresponds to the general error state.

Possible SQLSTATE values are:

- 00000 success
- HY000 all other errors

However, this also means that client applications that expect more specific SQLSTATE errors must be partially re-written.

7.3 Available Oracle Support for MySQL APIs

Oracle MySQL driver implements the APIs listed in MySQL C API documentation. Please note the following:

- Some MySQL functions have changed behavior, typically due to not having an equivalent behavior in Oracle; the description notes the changed behavior.
- Some MySQL functions are not supported; the description marks them accordingly. The driver returns an error for these functions, and prompts the application to work around the unsupported functionality.

Supported MySQL APIs are grouped functionally here, and here are links to more extensive information. However, we do not provide full documentation of function behavior and parameters, leaving it to the original MySQL C API documentation.

Client Library Initialization and Termination

The following interfaces support client library initialization and termination: mysql_library_end(), mysql_library_init(), mysql_server_end(), and mysql_server_init().

Connection Management

The following interfaces support connection management: my_init(), mysql_change_user(), mysql_close(), mysql_connect(), mysql_get_character_set_info(), mysql_get_ssl_cipher(), mysql_init(), mysql_options(), mysql_ping(), mysql_real_connect(), mysql_select_db(), mysql_set_character_set(), andmysql_ssl_set().

Error Reporting

The following interfaces support error reporting: mysql_errno(), mysql_error(), andmysql_sqlstate()

Statement Construction and Execution

The following interfaces support statement construction and execution: mysql_affected_rows(), mysql_escape_string(), mysql_hex_string(), mysql_kill(), mysql_query(), mysql_real_escape_string(), mysql_real_query(), and mysql_reload().

Result Set Processing

The following interfaces support result set processing: mysql_data_seek(), mysql_eof(), mysql_fetch_field(), mysql_fetch_field_direct(), mysql_fetch_fields(), mysql_fetch_lengths(), mysql_fetch_row(), mysql_field_count(), mysql_field_seek(), mysql_field_tell(), mysql_free_result(), mysql_insert_id(), mysql_list_dbs(), mysql_list_fields(), mysql_list_processes(), mysql_list_tables(), mysql_more_results(), mysql_next_result(), mysql_num_fields(), mysql_num_rows(), mysql_row_seek(), mysql_row_tell(), mysql_store_result(), and mysql_use_result().

Prepared Statements

The following interfaces support statement preparation: mysql_stmt_affected_rows(), mysql_stmt_attr_get(), mysql_stmt_attr_set(), mysql_stmt_bind_param(), mysql_stmt_bind_result(), mysql_stmt_close(), mysql_stmt_data_seek(), mysql_stmt_errno(), mysql_stmt_error(), mysql_stmt_execute(), mysql_stmt_fetch(), mysql_stmt_fetch_column(), mysql_stmt_field_count(), mysql_stmt_free_result(), mysql_stmt_init(), mysql_stmt_insert_id(), mysql_stmt_next_result(), mysql_stmt_num_rows(), mysql_stmt_param_count(), mysql_stmt_param_metadata(), mysql_stmt_prepare(), mysql_stmt_reset(), mysql_stmt_result_metadata(), mysql_stmt_row_seek(), mysql_stmt_row_tell(), mysql_stmt_send_long_data(), mysql_stmt_sqlstate(), and mysql_stmt_store_result().

Transaction Control

The following interfaces support transaction control: mysql_autocommit(), mysql_commit(), and mysql_rollback().

Information Routines

The following interfaces support information routines: mysql_character_set_name(), mysql_get_client_info(), mysql_get_client_version(), mysql_get_host_info(), mysql_get_proto_info(), mysql_get_server_info(), mysql_get_server_version(), mysql_info(), mysql_stat(), mysql_thread_id(), and mysql_warning_count().

Administrative Routines

The following interfaces support administrative routines: mysql_refresh(), mysql_set_server_option(), mysql_set_local_infile_default(), mysql_set_local_infile_handler(), and mysql_shutdown().

Miscellaneous Routines

The following interfaces support all remaining routines: mysql_create_db(), mysql_debug(), mysql_debug_info(), mysql_drop_db(), mysql_dump_debug_info(), mysql_read_query_result(), mysql_send_query(), mysql_thread_end(), mysql_thread_init(), and mysql_thread_safe().

7.3.1 my_init()

This function is a no-op function. It is called by `my_init` macro in `my_sys.h` file. All initializations are done by the `mysql_library_init()`.

Return Value

0

7.3.2 mysql_affected_rows()

Returns the number of rows processed for `INSERT`, `UPDATE`, and `DELETE` statements executed.

For `UPDATE` statements, note that the semantics of MySQL do not report rows where the new value is the same as the old value. In contrast, Oracle reports that rows are affected, even if the new value is the same as the old value. This function implements Oracle semantics. Therefore, existing applications that rely on this call may have to make programmatic changes.

For `SELECT` statement, the return is (`my_ulonglong`) -1.

Return Value

A number of rows that were processed by DML statement; `>0`. `0` indicates no updates were made by the statement. `-1` indicates that the statement was a query (`SELECT`), or an error.

7.3.3 mysql_autocommit()

Sets auto commit mode to `ON` or `OFF`.

Return Value

`0`, if the auto commit mode is changed successfully. Non-zero if an error occurred in the process.

7.3.4 mysql_change_user()

Changes the user, including user name, password, and database on the same or different host. In Oracle Database 12*c*, change of the database is not supported, so the value entered for the `db` parameter is ignored.

A call to `mysql_change_user()` rolls back any active transactions, ends the current session, and then re-establishes a new connection based on information stored in the `host` parameter.

Existing applications must make necessary application logic changes to implement this behavior in Oracle Database 12*c*.

Return Value

`0` if connection can be reestablished with the original host for the supplied user name and password. Non-zero if an error occurred.

7.3.5 mysql_character_set_name()

Not supported in Oracle Database 12*c*. Applications that rely on results of this call must change their application logic.

Return Value

Empty string.

7.3.6 mysql_close()

Closes the connection and frees all associated data structures.

Return Value

7.3.7 mysql_commit()

Commits the transaction currently associated with the service context.

A `mysql_commit()` call supports the default mode in Oracle Database 12*c*. It therefore ignores the `completion type` system variable.

Existing applications that use this API to perform MySQL-specific `completion type` operations must change their application logic.

Return Value

`0` if successful, non-zero otherwise.

7.3.8 mysql_connect()

Deprecated; use mysql_real_connect().

Return Value

Initialized `MYSQL` structure. `NULL` if an error occurred.

7.3.9 mysql_create_db()

Not supported in Oracle Database 12*c*. Applications that rely on results of this call must change their application logic.

Return Value

`0` if successful; non-zero if an invalid MYSQL structure is passed in.

7.3.10 mysql_data_seek()

Seeks to a row in a result set based on the value specified in the offset parameter.

Offset value, being a row number, can range from `0` to `mysql_num_rows(result)` -1.

Return Value

None

7.3.11 mysql_debug()

Not supported in Oracle Database 12*c*. Applications that rely on results of this call must change their application logic.

7.3.12 mysql_debug_info()

Not supported in Oracle Database 12*c*. Applications that rely on results of this call must change their application logic.

Return Value

0 if successful; non-zero if invalid MYSQL structure.

7.3.13 mysql_drop_db()

Not supported in Oracle Database 12*c*. Applications that rely on results of this call must change their application logic.

Return Value

0 if successful; non-zero if invalid MYSQL structure.

7.3.14 mysql_dump_debug_info()

Not supported in Oracle Database 12*c*. Applications that rely on results of this call must change their application logic.

Return Value

0 if successful; non-zero if an invalid MYSQL structure is passed in.

7.3.15 mysql_eof()

DEPRECATED. Use mysql_errno() or mysql_error() instead.

Determines if the last row of a result set has been read.

Return Value

1 if fetched the last row; otherwise 0.

7.3.16 mysql_errno()

Returns Oracle error number of the last error on the connection or the global context.

If the previous call did not have an established connection, pass in NULL; this returns the last error number on global context.

Return Value

Last error number on the MYSQL connection, or the last error number on the global context.

7.3.17 mysql_error()

Returns Oracle error messages for the last error on the connection or the global context.

If the previous call did not have an established connection, pass in NULL; this returns the last error message on global context.

Return Value

Last error message on the MYSQL connection, or the last error message on the global context.

7.3.18 mysql_escape_string()

Encodes the string in the source (from parameter), places it in the destination (to parameter), and appends a terminating NULL.

Supports encoding of only one character, '\' using the current character set in the connection.

See mysql_real_escape_string().

Return Value

The length of the value placed into to, excluding the terminating NULL.

7.3.19 mysql_fetch_field()

Returns the definition of one column of a result set as a MYSQL_FIELD structure.

Only the following attributes of the MYSQL_FIELD structure are supported: flag, name, name_length, org_name, org_name_length, type, and max_length.

- The flag attribute supports only the following values: NOT_NULL_FLAG, NUM_FLAG, and BINARY_FLAG.
- The attribute org_name is set to have the same value as name attribute.
- The attribute org_name_length is set to have the same value as name_length attribute.

Return value

The MYSQL_FIELD structure for the current column. NULL if no columns are left.

7.3.20 mysql_fetch_field_direct()

Retrieves the column's field definition for a specified field number as a MYSQL_FIELD structure.

Return Value

Field definition for the specific field. NULL if an error occurred, or if field number fieldnr is not in range.

7.3.21 mysql_fetch_fields()

Returns an array of all MYSQL_FIELD structures for a result set. Each MYSQL_FIELD structure gives the field definition for one column of the result set.

Return Value

`NULL` if an error occurred.

7.3.22 mysql_fetch_lengths()

Returns an array of lengths of the column on the current row.

Return Value

An array of unsigned long integers that represent the size of each column. `NULL` if an error occurred.

7.3.23 mysql_fetch_row()

Retrieves the next row of a result set.

Return Value

A MYSQL_ROW structure for the next row. `NULL` if there are no more rows to retrieve or if an error occurred.

7.3.24 mysql_field_count()

Returns the number of columns in the result set for the recent query on the connection.

Return Value

Number of fields in the result set within the MYSQL structure.; `0` if an error occurred.

7.3.25 mysql_field_seek()

Sets the field cursor to the specified offset.

Return Value

The offset to the field set

7.3.26 mysql_field_tell()

Returns the position of the field; used for the current field.

Return Value

Offset of the current field

7.3.27 mysql_free_result()

Frees the memory allocated for the result set.

Return Value

None

7.3.28 mysql_get_character_set_info()

Not supported in Oracle Database 12*c*. Applications that rely on results of this call must change their application logic.

Return Value

None

7.3.29 mysql_get_client_info()

Returns MySQL version number defined by `MYSQL_SERVER_VERSION` macro in `mysql_version.h` header file, in string format. The macro definition is used in the `mysql_version.h file` that builds `oramysql` library; it is not the `mysql_version.h` file used by the application.

Return Value

A character string that represents MySQL client library version.

7.3.30 mysql_get_client_version()

Returns current MySQL version, as defined by `MYSQL_VEERSION_ID` macro in the `mysql_version.h` header file. The macro definition is used in the `mysql_version.h file` that builds `oramysql` library; it is not the `mysql_version.h` file used by the application.

Return Value

An unsigned long integer for MySQL version stored in the `MYSQL_VERSION_ID` macro. The macro definition is used in the `mysql_version.h file` that builds `oramysql` library; it is not the `mysql_version.h` file used by the application.

7.3.31 mysql_get_host_info()

Returns the host name used to connect to the database.

Return Value

A character string of host name. `NULL` in case of an error.

7.3.32 mysql_get_proto_info()

This is a no-op under Oracle environment. Applications that rely on results of this call must change their application logic.

Return Value

0

7.3.33 mysql_get_server_info()

Returns the Oracle server version in text string format, such as "12.1.0.1.0".

ORACLE®

Applications that rely on results of this call must change their application logic.

Return Value

A character string that represents Oracle Server Number. `NULL` if an error occurred.

7.3.34 mysql_get_server_version()

Returns Oracle Database version number, such as `120100`. This is in integer *XXYYZZ* format, where *XX* represents the major version, *YY* represents the minor version, and *ZZ* represents the version within the release level.

Return Value

Oracle Database version number. `0` if an error occurred.

7.3.35 mysql_get_ssl_cipher()

This is a no-op API. Applications that rely on results of this call must change their application logic.

Return Value

`NULL`

7.3.36 mysql_hex_string()

Encodes string specified by `from` parameter to hexadecimal format. Each character is encoded as two hexadecimal digits. The result is placed in the `to` parameter, with a terminal `NULL` byte.

The `to` buffer should have a minimum size equal to `length*2+1` bytes.

Return Value

Length of the value placed into `to` parameter, excluding the terminating `NULL` character.

7.3.37 mysql_info()

This is a no-op API. Applications that rely on results of this call must change their application logic.

Return Value

`NULL`

7.3.38 mysql_init()

Allocates a `MYSQL` structure if `NULL` is passed. Otherwise, this call initializes the passed in `MYSQL` structure.

Return Value

Initialized MYSQL structure. `NULL` if `MYSQL` structure cannot be allocated or initialized.

7.3.39 mysql_insert_id()

This is a no-op API. Applications that rely on results of this call must change their application logic.

Return Value

`0`

7.3.40 mysql_kill()

This is a no-op API. Applications that rely on results of this call must change their application logic.

Return Value

`0`, and non-zero if an invalid MYSQL structure is passed in.

7.3.41 mysql_library_end()

Terminates `oramysql` library.

Return Value

7.3.42 mysql_library_init()

Initializes `oramysql` library.

Return Value

`0` if successful, non-zero in case of a failure to initialize MySQL library.

7.3.43 mysql_list_dbs()

Returns a list of database names that match the wild parameter on the server.

To use this API, the DBA creates the `oramysql_dbs_view` view, and grants privileges to `PUBLIC`.

For Oracle Database 12*c*

For Oracle Database 12*c*, view `oramysql_dbs_view` is based on the `V$DATABASE` and `V$PDBS` system objects.

When connecting to Oracle Database 12*c* and subsequent versions, use the following SQL script to create the view `oramysql_dbs_view` in Oracle Database 12*c*:

```
create view oramysql_dbs_view(name) as select left.name from v$pdbs left
  union select right.name from v$database right;
create public synonym oramysql_dbs_view for oramysql_dbs_view;
grant select on oramysql_dbs_view to public;
```

If `oramysql_dbs_view` view does not exist when an application calls the `mysql_list_dbs()` function, the information is retrieved from the `V$ PDBS` and `V$ DATABASE` tables. However, this generates errors if the user does not have privileges to access these tables.

For Oracle Databases prior to Oracle Database 12*c*

Use the following SQL script to create the view `oramysql_dbs_view` in the Oracle Database:

```
create view oramysql_dbs_view(name) as select name form v$database;
create public synonym oramysql_dbs_view for oramysql_dbs_view;
grant select on oramysql_dbs_view to public;
```

If the view does not exist, the wild parameter is ignored, and the call executes the following SQL statement:

```
select SYS_CONTEXT( 'USERENV', 'DB_NAME') from DUAL;
```

Return Value

`NULL` if an error occurs, a `MYSQL_RES` result set if successful.

7.3.44 mysql_list_fields()

Returns the column names that match the `wild` parameter for a specified table.

Return Value

`NULL` if an error occurred, a MySql result set if successful.

7.3.45 mysql_list_processes()

This is a no-op API. Applications that rely on results of this call must change their application logic.

Return Value

`NULL`

7.3.46 mysql_list_tables()

This is a no-op function. Applications that rely on results of this call must change their application logic.

Return Value

`NULL`

7.3.47 mysql_more_results()

Verifies if more results are available from the currently executing statement.

Return Value

`TRUE` if more results exist; `FALSE` if no more result sets exist.

7.3.48 mysql_next_result()

Gets the next result set.

Returns Value

`0` if successful and there are more results; `-1` if successful and there are no more results; `>0` if an error occurred.

7.3.49 mysql_num_fields()

Returns the number of columns in a result set.

Return Value

An unsigned integer that represents the number of columns in the result set; returns `0` if not successful.

7.3.50 mysql_num_rows()

Returns the number of rows in the result set.

Return Value

The number of rows in the result set; otherwise `0`.

7.3.51 mysql_options()

This is a no-op function. Applications that rely on results of this call must change their application logic.

Return Value

`0` if successful, non-zero if an invalid `MYSQL` structure is passed in.

7.3.52 mysql_ping()

If the server cannot be accessed, returns an error with connection failure details.

Return Value

`0` if success, non-zero if error occurred.

7.3.53 mysql_query()

Executes the SQL statement pointed to by the null-terminated string.

Return Value

`0` if successful, non-zero if an error occurred.

7.3.54 mysql_read_query_result()

This is a no-op function; query results from `mysql_send_query()` are available when that call completes.

Return Value

`0`

7.3.55 mysql_real_connect()

The `db` parameter is not used in Oracle Database 12*c*. Existing applications using this parameter to connect to a `db` must supply the connection identifier or service name in the `host` parameter. The connection string has the following format:

```
[//]host[:port][/service_name][:server][/instance_name]
```

For instance, the host parameter would appear as: `ca-tools3.us.oracle.com/orcl3`, when connecting to host `ca-tools3.us.oracle.com` with SID `orcl3`.

The parameters `db`, `port`, `unix_socket`, and `client_flag` are not in use. When the user must specify the port, it has to be in the syntax method used for `host` parameter.

Return Value

`MYSQL` structure initialized if successful. `NULL` in case initialization does not work.

7.3.56 mysql_real_escape_string()

Encodes the string in the source (`from` parameter) and the result is placed in the destination (`to` parameter) and a terminating null byte is appended.

Note that only single-quote characters are escaped. Each single-quote is escaped using Oracle semantics. The `to` buffer should have a minimum size of `length*2+1 bytes`. Each single quote in the original string is replaced by two consecutive single quotes.

See mysql_escape_string().

Return Value

The length of the value placed into `to` buffer, excluding the terminating `NULL`. `0` otherwise.

7.3.57 mysql_real_query()

This function executes the query string.

Return Value

`0` if successful, non-zero in case of an error.

7.3.58 mysql_refresh()

This is a no-op API. Applications that rely on results of this call must change their application logic.

Return Value

`0` if successful. Non-zero if an invalid MySQL structure was passed in.

7.3.59 mysql_reload()

Reloads the grant tables. This function is deprecated, and has not been implemented. Use mysql_query() instead. Applications that rely on results of this call must change their application logic.

7.3.60 mysql_rollback()

Rolls back the current transaction defined as the set of statements executed after the last mysql_commit() or mysql_real_connect() call. If the application is running under object mode, the modified or updated objects in the object cache for this transaction are also rolled back.

A `mysql_rollback()` call supports the default mode in Oracle Database 12*c*. It therefore ignores the `completion type` system variable.

Existing applications that use this API to perform MySQL-specific `completion type` operations must change their application logic.

Return Value

Error if an attempt is made to roll back a global transaction that is not currently active.

7.3.61 mysql_row_seek()

Sets to a particular row and returns offset of previous row.

Return Value

Offset of previous row in `MYSQL_ROW_OFFSET` structure.

7.3.62 mysql_row_tell()

Gives the current row position in the result set.

Return Value

Offset of current row in `MYSQL_ROW_OFFSET` structure. `NULL` if an error occurred.

7.3.63 mysql_select_db()

This is a no-op API. Applications that rely on results of this call must change their application logic.

Return Value

0

7.3.64 mysql_send_query()

Sends a query. This function is not asynchronous in `oramysql` library. Instead, the call blocks until the query is executed.

Return Value

0 if successful, non-zero if an error occurred.

7.3.65 mysql_server_end()

Terminates and cleans up `oramysql` library.

Return Value

7.3.66 mysql_server_init()

Initializes the `oramysql` client library before any connections are created. The function `mysql_library_init()` macro is defined to be `mysql_server_init()` in `mysql.h` header file. This call is not thread-safe. Only one thread is expected to call it.

Return Value

0 if successful, non-zero if an error occurred.

7.3.67 mysql_set_character_set()

This is a no-op API. Applications that rely on results of this call must change their application logic.

Return Value

0

7.3.68 mysql_set_local_infile_default()

This is a no-op API. Applications that rely on results of this call must change their application logic.

Return Value

0

7.3.69 mysql_set_local_infile_handler()

This is a no-op API. Applications that rely on results of this call must change their application logic.

Return Value

`0`

7.3.70 mysql_set_server_option()

This is a no-op API. Applications that rely on results of this call must change their application logic.

Return Value

`0`

7.3.71 mysql_shutdown()

Helps shutdown an Oracle Database instance. Before using the `mysql_shutdown` API, the C program must connect to server with `SYSDBA` or `SYSOPER` session.

The parameters `mysql_shutdown_level` and `mysql_enum_shutdown_level` are ignored. Internally, the `OCIDBShutdown()` call is executed in the `OCI_DEFAULT` mode.

Return Value

`0` if successful. Non-zero if an error occurred.

7.3.72 mysql_sqlstate()

Returns `SQLSTATE` string which is not null-terminated. There are many `SQLSTATE` codes in MySQL which are not in use.

Return Value

`SQLSTATE` code: `00000` - Success, or `HY000` - All other errors.

7.3.73 mysql_ssl_set()

This is a no-op API. Applications that rely on results of this call must change their application logic.

Return Value

`0` if successful. Non-zero if an invalid MYSQL structure was passed.

7.3.74 mysql_stat()

This is a no-op API. Applications that rely on results of this call must change their application logic.

Return Value

A string of 4 blanks (" ") if successful. `NULL` if an invalid `MYSQL` structure was passed.

7.3.75 mysql_stmt_affected_rows()

This function returns the number of rows affected by the execution on the prepared statement.

Return Value

Number of rows affected by the DML operation if successful. `(my_ulonglong)-1` if an error occurred, or a `SELECT` statement was executed.

7.3.76 mysql_stmt_attr_get()

This is a no-op API. Applications that rely on results of this call must change their application logic.

Return Value

`0`

7.3.77 mysql_stmt_attr_set()

This is a no-op API. Applications that rely on results of this call must change their application logic.

Return Value

`0`

7.3.78 mysql_stmt_bind_param()

This function binds all the parameters in the prepared statement.

Return Value

`0` if parameters are bound successfully. Non-zero if an error occurred.

7.3.79 mysql_stmt_bind_result()

Binds program variables for all `SELECT` list columns of a prepared statement.

Return Value

`0` if successful. Non-zero if an error occurred.

7.3.80 mysql_stmt_close()

Closes a `MYSQL_STMT` object.

Return Value

`0`

7.3.81 mysql_stmt_data_seek()

This function seeks to get data for a particular row.

Return Value

None

7.3.82 mysql_stmt_errno()

Returns error number for the last error that occurred on the MYSQL_STMT object.

Return Value

7.3.83 mysql_stmt_error()

This function returns error message for the last error that occurred on the MYSQL_STMT object.

Return Value

A const *char error message.

7.3.84 mysql_stmt_execute()

This function executes the prepared statement.

Return Value

0 if the statement executed successfully; non-zero if an error occurred.

7.3.85 mysql_stmt_fetch()

This function fetches one row in program variables bound by the mysql_stmt_bind_result call.

Return Value

0 if one row is successfully fetched. MYSQL_NO_DATA if no more rows/data exists. MYSQL_DATA_TRUNCATED if data truncation occurred. 1 if an error occurred.

7.3.86 mysql_stmt_fetch_column()

This function fetches one column from the current result set row.

Return Value

0 if the value was fetched successfully. Non-zero if an error occurred.

7.3.87 mysql_stmt_field_count()

Fetches the number of fields in the MYSQL_STMT object.

Return Value

0 if an error occurred; otherwise, the number of fields in the result set associated with the MYSQL_STMT object.

7.3.88 mysql_stmt_free_result()

Frees the result set associated with the MYSQL_STMT object.

Return Value

0

7.3.89 mysql_stmt_init()

Creates a new MYSQL_STMT object from the MYSQL connection object.

Return Value

MYSQL_STMT object if successful. NULL if an error occurred.

7.3.90 mysql_stmt_insert_id()

This is a no-op API. Applications that rely on results of this call must change their application logic.

Return Value

0

7.3.91 mysql_stmt_next_result()

This function is not implemented. Applications that rely on results of this call must change their application logic.

Return Value

0

7.3.92 mysql_stmt_num_rows()

Returns the number of rows in a stored result set. In case of a non-stored (unbuffered result set), it returns the total number of rows fetched so far.

Return Value

0 if an error occurred in fetching the number of rows.

7.3.93 mysql_stmt_param_count()

Returns the number of bind parameters in the prepared statement.

Return Value

0 if an error occurred in returning the number of bind parameters.

7.3.94 mysql_stmt_param_metadata()

This function is cast to MySql result set `(MYSQL_RES *) NULL`

Return Value

NULL

7.3.95 mysql_stmt_prepare()

Prepares a statement in the `MYSQL_STMT` for execution.

Return Value

0 if successful, non-zero if an error occurred.

7.3.96 mysql_stmt_reset()

Resets the prepared statement in the `MYSQL_STMT`.

Return Value

0

7.3.97 mysql_stmt_result_metadata()

Returns the metadata for the result of a `SELECT` statement that is executed through a `MYSQL_STMT` object.

Return Value

A result set that describes the metadata of the prepared `SELECT` statement. `NULL` if an error occurred.

7.3.98 mysql_stmt_row_seek()

Seeks to a row position and returns the offset of the previous row.

Return Value

An offset of the previous row in `MYSQL_ROW_OFFSET` structure.

7.3.99 mysql_stmt_row_tell()

Gives the current row position in the result set.

Return Value

Current row position. `NULL` if an error occurred.

7.3.100 mysql_stmt_send_long_data()

Sends parameter data to the server in parts.

The function `mysql_stmt_bind_param()` must be called first, then `mysql_stmt_send_long_data()`, followed by `mysql_stmt_execute()`.

The function can be called multiple times to send parts of a character or binary data value for a column.

Return Value

`0` if the data is sent to the server successfully, non-zero if an error occurred.

7.3.101 mysql_stmt_sqlstate()

Returns `SQLSTATE` string for the recent prepared statement. There are many `SQLSTATE` codes in MySQL that are not used.

Return Value

`SQLSTATE` codes: "`00000`" - Success, or "`HY0000`" - All other errors.

7.3.102 mysql_stmt_store_result()

Stores the result set from the last query.

If the last query was a `SELECT`, a result set is returned. If the last statement was a non-`SELECT` or error, a `NULL` result set is returned.

Return Value

A valid result set if successful, `NULL` if an error occurred, or a non-`SELECT` statement.

7.3.103 mysql_store_result()

Stores the result set from the last query.

If the last query was `SELECT`, returns a result set.

If the last statement was a non-`SELECT` or an error, a `NULL` result set is returned.

Return Value

A valid result set if successful; otherwise, `NULL` for errors or non-`SELECT` statements.

7.3.104 mysql_thread_end()

This is a no-op API. Applications that rely on results of this call must change their application logic.

Return Value

7.3.105 mysql_thread_id()

Returns Oracle session identifier (SID) for the connection. This is obtained internally by executing the following SQL statement:

```
select SYS_CONTEXT('USERENV', 'SID') from DUAL;
```

Applications that rely on results of this call must change their application logic.

Return Value

Oracle session identifier (SID). `0` if an error occurs.

7.3.106 mysql_thread_init()

This is a no-op API. Applications that rely on results of this call must change their application logic.

Return Value

`0`

7.3.107 mysql_thread_safe()

The `oramysql` library is thread-safe, so this function always returns `TRUE`.

Return Value

`TRUE`

7.3.108 mysql_use_result()

Initiates a result set retrieval.

Return Value

`NULL` if an error occurred, a valid result set if successful.

7.3.109 mysql_warning_count()

This is a no-op API. Applications that rely on results of this call must change their application logic.

Return Value

`0` if successful, non-zero if an error occurred.

8

API Reference for SQL Translation of JDBC Applications

Consider the APIs that are part of the `oracle.jdbc` package, specifically the elements of `oracle.jdbc` that assist in SQL translation. To successfully migrate JDBC applications, it is important to understand the translation properties, interfaces, and the error translation mechanisms.

> ✎ **See Also:**
>
> - Complete documentation of the `oracle.jdbc` package in *Oracle Database JDBC Java API Reference*

8.1.1 Translation Properties

The translation properties are listed in Table 8-1

Table 8-1   Translation Properties

Property	Description
sqlTranslationProfile	Specifies the name of the transaction profile
sqlErrorTranslationFile	Specifies the path of the SQL error translation file

8.1.1.1 sqlTranslationProfile

The property `oracle.jdbc.sqlTranslationProfile` specifies the name of the transaction profile.

Declaration

```
oracle.jdbc.sqlTranslationProfile
```

Constant

```
OracleConnection.CONNECTION_PROPERTY_SQL_TRANSLATON_PROFILE
```

The value of the constant is `oracle.jdbc.sqlTranslationProfile`. This is also the property name.

Property Value

The value is a string. There is no default value.

Remarks

The property `sqlTranslationProfile` can be set as either a system property or a connection property. The property is required to use SQL translation. If this property is set then all statements created by the connection have SQL translation enabled unless otherwise specified.

8.1.1.2 sqlErrorTranslationFile

The property `oracle.jdbc.sqlErrorTranslationFile` specifies the path of the SQL error translation file.

Declaration

`oracle.jdbc.sqlErrorTranslationFile`

Constant

`Oracle.connection.CONNECTION_PROPERTY_SQL_ERROR_TRANSLATION_FILE.`

Property Value

The value is a path name. It has no default value.

Exceptions

An error in establishing a connection results in a SQLException but without a valid connection. However the SQL error translation file path is available either as a system property or connection property and will be used to translate the error.

Remarks

This file is used only for translating errors which occur when connection establishment fails. Once the connection is established this file is bypassed and is not considered even if it contains the translation details for any error which occurs after the connection is established. The property `sqlErrorTranslationFile` can be either a system property or a connection property. The content of this file is used to translate Oracle SQLExceptions into foreign SQLExceptions when there is no valid connection.

8.1.2 OracleTranslatingConnection Interface

This interface is only implemented by a Connection object that supports SQL Translation. The main purpose of this interface is to get non-translating statements (including `preparedStatement` and `CallableStatement`) from a translating connection.

The public interface `oracle.jdbc.OracleTranslatingConnection` defines the factory methods for creating translating and non-translating `Statement` objects.

The `OracleTranslatingConnection` enumerations are listed in Table 8-2:

Table 8-2   OracleTranslatingConnection Enumeration

Name	Description
SqlTranslationVersion	Provides the Keys to the map

The `OracleTranslatingConnection` methods are listed in Table 8-3:

Table 8-3   OracleTranslatingConnection Methods

Name	Description
createStatement()	Creates a `Statement` object with option to translate or not translate SQL.
prepareCall()	Creates a `CallableStatement` object with option to translate or not translate SQL.
prepareStatement()	Creates a `PreparedStatement` object with option to translate or not translate SQL.
getSQLTranslationVersions()	Returns a map of all the translation versions of the query during SQL Translation.

8.1.2.1 SqlTranslationVersion

The `SqlTranslationVersion` enumerated values specify the keys to the getSQLTranslationVersions() method.

Syntax

```
public enum SqlTranslationVersion {
   ORIGINAL_SQL,
   JDBC_MARKER_CONVERTED,
   TRANSLATED
}
```

The following table lists all the `SqlTranslationVersion` enumeration values with a description of each enumerated value.

Member Name	Description
ORIGINAL_SQL	Specifies the original vendor specific sql
JDBC_MARKER_CONVERTED	Specifies that JDBC parameter markers ('?') is replaced with Oracle style parameter markers (':b$<n>$'). Hence consecutive '?'s will be converted to `:b1`, `:b2`, `:b3` and so on. This change is required to take care of any changes in the order of parameters during translation. This version is sent to the server for translation. Hence any custom translations on the server must be registered from this version and not the `ORIGINAL_SQL` version.
TRANSLATED	Specifies the translated query returned from the server

8.1.2.2 createStatement()

This group of methods create a `Statement` object, and specify whether the statement supports SQL translation. If the value of parameter `translating` is `TRUE`, then the returning statement supports translation and is identical to the corresponding version in the java.sql.Connection interface without the translating argument. If the value is `FALSE`, then the returning statement does not support translation.

Syntax	Description
`public Statement createStatement( boolean translating) throws SQLException;`	Creates a `Statement` object with option to translate or not translate SQL.
`public Statement createStatement( int resultSetType, int resultSetConcurrency, boolean translating) throws SQLException;`	Creates a `Statement` object with the given type and concurrency with option to translate or not translate SQL.
`public Statement createStatement( int resultSetType, int resultSetConcurrency, int resultSetHoldability, boolean translating) throws SQLException;`	Creates a `Statement` object with the given type, concurrency, and holdability with option to translate or not translate SQL.

Parameters

Parameter	Description
`resultSetType`	Specifies the `int` value representing the result set type.
`resultSetConcurrency`	Specifies the `int` value representing the result set concurrency type.
`resultSetHoldability`	Specifies the `int` value representing the result set holdability type.
`translating`	Specifies whether or not the statement supports translation.

Return Value

The `createStatement()` method returns a `Statement` object.

Exceptions

The `createStatement()` method throws `SQLException`.

Example

Import the following packages before running the example:

```
import java.sql.*;
import java.util.Properties;

import oracle.jdbc.OracleConnection;
import oracle.jdbc.OracleTranslatingConnection;
import oracle.jdbc.pool.OracleDataSource;
```

Run the following SQL statements:

```
conn system/manager;
grant create sql translation profile to HR;
```

```
conn HR/hr;
drop table sample_tab;
create table sample_tab (c1 number, c2 varchar2(100));
insert into sample_tab values (1, 'A');
insert into sample_tab values (2, 'B');
commit;
exec dbms_sql_translator.drop_profile('FOO');
exec  dbms_sql_translator.create_profile('FOO');
exec  dbms_sql_translator.register_sql_translation('FOO','select row of (c1, c2)
from sample_tab','select c1, c2 from sample_tab');
```

Example 8-1   Using the createStatement() method

```
public class SQLTransStmt
{
  static String url="jdbc:oracle:thin:@localhost:5521:orcl";
  static String user="HR", pwd="hr";
  static String PROFILE = "FOO";
  static String primitiveSql = "select row of (c1, c2) from sample_tab";

  public static void main(String[] args) throws Exception
  {
    OracleDataSource ods = new OracleDataSource();
    ods.setURL(url);

    Properties props = new Properties();
    props.put("user", user);
    props.put("password", pwd);
    props.put(OracleConnection.CONNECTION_PROPERTY_SQL_TRANSLATION_PROFILE, PROFILE);
    ods.setConnectionProperties(props);
    Connection conn = ods.getConnection();
    System.out.println("connection for SQL translation: "+conn);

    try{
      OracleTranslatingConnection trConn = (OracleTranslatingConnection) conn;
      System.out.println("Call:
oracle.jdbc.OracleTranslatingConnection.createStatement(true)");
      Statement trStmt = trConn.createStatement(true);
      System.out.println("executeQuery for: "+primitiveSql);
      ResultSet trRs = trStmt.executeQuery(primitiveSql);
      while (trRs.next())
        System.out.println("C1:"+trRs.getInt(1)+", C2:"+trRs.getString(2));
      trRs.close();
      trStmt.close();
    }catch (Exception e) {
      e.printStackTrace();
    }

    try{
      OracleTranslatingConnection trConn = (OracleTranslatingConnection) conn;
      System.out.println("Call:
oracle.jdbc.OracleTranslatingConnection.createStatement(false)");
      Statement trStmt = trConn.createStatement(false);
      System.out.println("executeQuery for: "+primitiveSql);
      ResultSet trRs = trStmt.executeQuery(primitiveSql);
      while (trRs.next())
        System.out.println("C1:"+trRs.getInt(1)+", C2:"+trRs.getString(2));
      trRs.close();
      trStmt.close();
    }catch (Exception e) {
      System.out.println("expected Exception: "+e.getMessage());
```

```
    }

    try{
        OracleTranslatingConnection trConn = (OracleTranslatingConnection) conn;
        System.out.println("Call: oracle.jdbc.OracleTranslatingConnection.
createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE, true)");
        Statement trStmt = trConn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE, true);
        System.out.println("executeQuery for: "+primitiveSql);
        ResultSet trRs = trStmt.executeQuery(primitiveSql);
        while (trRs.next())
            System.out.println("C1:"+trRs.getInt(1)+", C2:"+trRs.getString(2));
        System.out.println("move resultset back to 2nd row...");
        trRs.absolute(2);
        while (trRs.next())
            System.out.println("C1:"+trRs.getInt(1)+", C2:"+trRs.getString(2));
        trRs.close();
        trStmt.close();
    }catch (Exception e) {
        e.printStackTrace();
    }

    try{
        conn.setAutoCommit(false);
        OracleTranslatingConnection trConn = (OracleTranslatingConnection) conn;
        System.out.println("Call:
oracle.jdbc.OracleTranslatingConnection.createStatement(ResultSet.TYPE_SCROLL_SENSITI
VE, ResultSet.CONCUR_UPDATABLE,
 ResultSet.HOLD_CURSORS_OVER_COMMIT, true)");
        Statement trStmt = trConn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE, ResultSet.HOLD_CURSORS_OVER_COMMIT, true);
        System.out.println("executeQuery for: "+primitiveSql);
        ResultSet trRs = trStmt.executeQuery(primitiveSql);
        trRs.last();
        System.out.println("C1:"+trRs.getInt(1)+", C2:"+trRs.getString(2));
        trRs.updateString(2, "Hello");
        trRs.updateRow();
        conn.commit();
        System.out.println("accept the update and list all of the rows again...");
        trRs.beforeFirst();
        while (trRs.next())
            System.out.println("C1:"+trRs.getInt(1)+", C2:"+trRs.getString(2));
        trRs.close();
        trStmt.close();
    }catch (Exception e) {
        e.printStackTrace();
    }

    conn.close();
  }
}
```

8.1.2.3 prepareCall()

This group of methods create a `CallableStatement` object, and specify whether the statement supports SQL translation. If the value of parameter `translating` is `TRUE`, then the returning statement supports translation. If the value is `FALSE`, then the returning statement does not support translation.

Syntax	Description
| ```
public CallableStatement prepareCall(
  String sql,
  boolean translating)
throws SQLException;
``` | Creates a `CallableStatement` object with option to translate or not translate SQL |
| ```
public CallableStatement prepareCall(
  String sql,
  int resultSetType,
  int resultSetConcurrency,
  boolean translating)
throws SQLException;
``` | Creates a `CallableStatement` object with the given type and concurrency with option to translate or not translate SQL |
| ```
public CallableStatement prepareCall(
  String sql,
  int resultSetType,
  int resultSetConcurrency,
  int resultSetHoldability,
  boolean translating)
throws SQLException;
``` | Creates a `CallableStatement` object with the given type, concurrency, and holdability with option to translate or not translate SQL |

Parameters

Parameter	Description
`sql`	Specifies the `String` SQL statement value to be sent to the database; may contain one or more parameters
`resultSetType`	Specifies the `int` value representing the result set type
`resultSetConcurrency`	Specifies the `int` value representing the result set concurrency type
`resultSetHoldability`	Specifies the `int` value representing the result set holdability type
`translating`	Specifies whether or not the statement supports translation

Return Value

The `prepareCall()` method returns a `CallableStatement` object.

Exceptions

The `prepareCall()` method throws `SQLException`.

Example

Import the following packages before running the example:

```
import java.sql.*;
import java.util.Properties;

import oracle.jdbc.OracleConnection;
import oracle.jdbc.OracleTranslatingConnection;
import oracle.jdbc.pool.OracleDataSource;
```

Run the following SQL statements:

```
conn system/manager;
grant create sql translation profile to HR;

conn HR/hr;

create or replace procedure sample_proc (p_num number, p_vchar in out varchar2) AS
begin
    p_vchar := 'p_num'||p_num||',  p_vchar'||p_vchar;
end;
/

exec dbms_sql_translator.drop_profile('FOO');
exec  dbms_sql_translator.create_profile('FOO');
exec  dbms_sql_translator.register_sql_translation('FOO', 'exec
sample_proc(:b1, :b2)', '{call sample_proc(:b1, :b2)}');
```

Example 8-2   Using the prepareCall() method

```
public class SQLTransCstmt
{
  static String url="jdbc:oracle:thin:@localhost:5521:orcl";
  static String user="HR", pwd="hr";
  static String PROFILE = "FOO";
  static String primitiveSql = "exec sample_proc(:b1, :b2)";

  public static void main(String[] args) throws Exception
  {
    OracleDataSource ods = new OracleDataSource();
    ods.setURL(url);

    Properties props = new Properties();
    props.put("user", user);
    props.put("password", pwd);
    props.put(OracleConnection.CONNECTION_PROPERTY_SQL_TRANSLATION_PROFILE,
      PROFILE);
    ods.setConnectionProperties(props);
    Connection conn = ods.getConnection();
    System.out.println("connection for SQL translation: "+conn);

    try{
      OracleTranslatingConnection trConn = (OracleTranslatingConnection) conn;
      System.out.println(
        "Call: oracle.jdbc.OracleTranslatingConnection.prepareCall(sql, true)");
      CallableStatement trStmt = trConn.prepareCall(primitiveSql, true);
      trStmt.setInt("b1", 1);
      trStmt.setString("b2", "A");
      trStmt.registerOutParameter("b2", Types.VARCHAR);
      System.out.println("execute for: "+primitiveSql);
      trStmt.execute();
      System.out.println("out param: "+trStmt.getString("b2"));

      trStmt.close();
    }catch (Exception e) {
      e.printStackTrace();
    }

    try{
      OracleTranslatingConnection trConn = (OracleTranslatingConnection) conn;
      System.out.println(
```

```
          "Call: oracle.jdbc.OracleTranslatingConnection.prepareCall(sql, false)");
      CallableStatement trStmt = trConn.prepareCall(primitiveSql, false);
      trStmt.setInt(1, 1);
      trStmt.setString(2, "A");
      System.out.println("execute for: "+primitiveSql);
      ResultSet trRs = trStmt.executeQuery();
      while (trRs.next())
        System.out.println("C1:"+trRs.getInt(1)+", C2:"+trRs.getString(2));
      trRs.close();

      trStmt.close();
    }catch (Exception e) {
      System.out.println("expected Exception: "+e.getMessage());
    }

    conn.close();
  }
}
```

8.1.2.4 prepareStatement()

This group of methods create a `PreparedStatement` object, and specify whether the statement supports SQL translation. If the value of parameter `translating` is `TRUE`, then the returning statement supports translation. If the value is `FALSE`, then the returning statement does not support translation.

Syntax	Description

`public PreparedStatement prepareStatement(`
`  String sql,`
`  boolean translating)`
`throws SQLException;`
	Creates a `PreparedStatement` object with option to translate or not translate SQL

`public PreparedStatement prepareStatement(`
`  String sql,`
`  int resultSetType,`
`  int resultSetConcur,`
`  boolean translating)`
`throws SQLException;`
	Creates a `PreparedStatement` object with the given type and concurrency with option to translate or not translate SQL

`public PreparedStatement prepareStatement(`
`      String sql,`
`      int resultSetType,`
`      int resultSetConcur,`
`      int resultSetHold,`
`      boolean translating)`
`throws SQLException;`
	Creates a `PreparedStatement` object with the given type, concurrency, and holdability with option to translate or not translate SQL

Parameter	Description
`sql`	Specifies the `String` SQL statement value to be sent to the database; may contain one or more parameters
`resultSetType`	Specifies the `int` value representing the result set type

Parameter	Description
resultSetConcur	Specifies the `int` value representing the result set concurrency type
resultSetHold	Specifies the `int` value representing the result set holdability type
translating	Specifies whether or not the statement supports translation

Return Value

The `prepareStatement()` method returns a `PreparedStatement` object.

Usage Notes

When the "?" placeholder is used with the `prepareStatement()` method, the driver internally changes the "?" to Oracle-style parameters because the server side translator can only work with Oracle-style markers. This is necessary to distinguish the bind variables. If not, any change in the order of the bind variables will be indistinguishable. The replaced oracle style markers follow the format :b<n> where <n> is an incremental number. For example, `exec sample_proc(?,?)` becomes `exec sample_proc(:b1,:b2)`.

To further exemplify, consider a scenario of a vendor format where the vendor query selecting top three rows is `SELECT * FROM employees WHERE first_name=? AND employee_id=? TOP 3`. The query has to be converted to oracle dialect. In this case the following translation is to be registered on the server:

From:

```
SELECT * FROM employees WHERE first_name=:b1 AND employee_id=:b2 TOP 3
```

To:

```
SELECT * FROM employees WHERE first_name=:b1 AND employee_id=:b2 AND ROWNUM <= 3
```

See SqlTranslationVersion and "SQL Translation of JDBC Applications" for more information.

Exceptions

The `prepareStatement()` method throws `SQLException`.

Example

Import the following packages before running the example:

```
import java.sql.*;
import java.util.Properties;

import oracle.jdbc.OracleConnection;
import oracle.jdbc.OracleTranslatingConnection;
import oracle.jdbc.pool.OracleDataSource;
```

Run the following SQL statements:

```
conn system/manager;
grant create sql translation profile to HR;
```

```
conn HR/hr;
drop table sample_tab;
create table sample_tab (c1 number, c2 varchar2(100));
insert into sample_tab values (1, 'A');
insert into sample_tab values (1, 'A');
insert into sample_tab values (1, 'A');
commit;
exec dbms_sql_translator.drop_profile('FOO');
exec  dbms_sql_translator.create_profile('FOO');
exec  dbms_sql_translator.register_sql_translation('FOO','select row of select c1,
c2 from sample_tab
 where c1=:b1 and c2=:b2','select c1, c2 from sample_tab where c1=:b1 and c2=:b2');
```

Example 8-3   Using the prepareStatement() method

```
public class SQLTransPstmt
{
  static String url="jdbc:oracle:thin:@localhost:5521:orcl";
  static String user="HR", pwd="hr";
  static String PROFILE = "FOO";
  static String primitiveSql = "select row of select c1, c2 from sample_tab
    where c1=:b1 and c2=:b2";

  public static void main(String[] args) throws Exception
  {
    OracleDataSource ods = new OracleDataSource();
    ods.setURL(url);

    Properties props = new Properties();
    props.put("user", user);
    props.put("password", pwd);
    props.put(OracleConnection.CONNECTION_PROPERTY_SQL_TRANSLATION_PROFILE,
      PROFILE);
    ods.setConnectionProperties(props);
    Connection conn = ods.getConnection();
    System.out.println("connection for SQL translation: "+conn);

    try{
      OracleTranslatingConnection trConn = (OracleTranslatingConnection) conn;
      System.out.println("Call:
        oracle.jdbc.OracleTranslatingConnection.prepareStatement(sql, true)");
      PreparedStatement trStmt = trConn.prepareStatement(primitiveSql, true);
      trStmt.setInt(1, 1);
      trStmt.setString(2, "A");
      System.out.println("executeQuery for: "+primitiveSql);
      ResultSet trRs = trStmt.executeQuery();
      while (trRs.next())
        System.out.println("C1:"+trRs.getInt(1)+", C2:"+trRs.getString(2));
        trRs.close();
        trStmt.close();
      }catch (Exception e) {
      e.printStackTrace();
    }

    try{
      OracleTranslatingConnection trConn = (OracleTranslatingConnection) conn;
      System.out.println("Call:
        oracle.jdbc.OracleTranslatingConnection.prepareStatement(sql, false)");
      PreparedStatement trStmt = trConn.prepareStatement(primitiveSql, false);
      trStmt.setInt(1, 1);
      trStmt.setString(2, "A");
```

```
        System.out.println("executeQuery for: "+primitiveSql);
        ResultSet trRs = trStmt.executeQuery();
        while (trRs.next())
          System.out.println("C1:"+trRs.getInt(1)+", C2:"+trRs.getString(2));
        trRs.close();

        trStmt.close();
      }catch (Exception e) {
      System.out.println("expected Exception: "+e.getMessage());
      }

      try{
        OracleTranslatingConnection trConn = (OracleTranslatingConnection) conn;
        System.out.println("Call:
          oracle.jdbc.OracleTranslatingConnection.prepareStatement(
          sql, ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE, true)");
        PreparedStatement trStmt = trConn.prepareStatement(
          primitiveSql, ResultSet.TYPE_SCROLL_SENSITIVE,
          ResultSet.CONCUR_READ_ONLY, true);
        trStmt.setInt(1, 1);
        trStmt.setString(2, "A");
        System.out.println("executeQuery for: "+primitiveSql);
        ResultSet trRs = trStmt.executeQuery();
        while (trRs.next())
          System.out.println("C1:"+trRs.getInt(1)+", C2:"+trRs.getString(2));

        System.out.println("trRs.beforeFirst and show resultSet again...");
        trRs.beforeFirst();
        while (trRs.next())
          System.out.println("C1:"+trRs.getInt(1)+", C2:"+trRs.getString(2));
        trRs.close();
        trStmt.close();
      }catch (Exception e) {
      e.printStackTrace();
      }

    conn.close();
    }
}
```

8.1.2.5 getSQLTranslationVersions()

Returns a map of all the translation versions of the query during SQL Translation. In
case of an exception, and if suppressExceptions is true, then the translated version in
the map is NULL.

Syntax

```
public Map<SqlTranslationVersion, String> getSqlTranslationVersions(
  String sql,
  boolean suppressExceptions)
throws SQL Exception;
```

Return Value

Map with all translation versions of a query. See SqlTranslationVersion enum for more
details about returning versions.

Exception

This method throws `SQLException` if there is a problem in query translation, provided `suppressExceptions` is false.

8.1.3 Error Translation Configuration File

An XML configuration file (path) is provided as a value of the `oracle.jdbc.sqlErrorTranslationFile` property. This file contains the translations information for errors. These errors occur when a connection to the server cannot be established and thus translation cannot happen on the server. Error messages are of the type that define the state of the database that prevents the connection from being established.

The structure of the configuration XML file is defined in the DTD as follows:

```
<!DOCTYPE LocalTranslationProfile[

<!ELEMENT LocalTranslationProfile (Exception+)>
<!ELEMENT Exception (ORAError, ErrorCode, SQLState )>
<!ELEMENT ORAError (#PCDATA)>
<!ELEMENT ErrorCode (#PCDATA)>
<!ELEMENT SQLState (#PCDATA)>
]>
```

where,

- `ORAError` is an `int` value and specifies the error code for the oracle error.
- `ErrorCode` is an `int` value and specifies the vendor error code, that is, the translated code.
- `SQLState` is a `String` value and specifies the vendor SQL state.

Glossary

adapter

A real-time, proprietary tool used to enable access to data stored in one database from another database. Adapters are commonly used to translate SQL, map data types, and facilitate the integration of SQL statements, triggers, and stored procedures.

custom SQL translation

A scenario in which users can register their customer-specific translations of SQL statements with the SQL Translation Profile. During the translation of non-Oracle statements, the profile looks up the custom translations first. Then, if no match is found, it invokes the SQL Translator.

data integration

The exchange of data between different databases, either asynchronously in real-time transactions or synchronously as batch processes.

data integration framework

A set of tools and processes used to enable data exchanges between different databases. Traditional frameworks include many nightly processes such as large batch extractions and feeds, and bulk loading of data. Newer frameworks can include small daily processes and feeds occurring in near real time.

database schema migration

The process of identifying and converting tables, columns, and other objects in a non-Oracle schema to conform to the naming, size, and other conventions required by Oracle Database.

error translation

A scenario in which users can register vendor-specific translations of error codes and messages with the SQL Translation Profile. During SQL execution, client applications rely on vendor-specific error codes and messages. When errors occur, the translated error codes and messages are returned instead of the Oracle error codes and messages.

migration

The process of modifying a non-Oracle application, including all of its components (such as architecture, data, SQL code, and client) to use the Oracle RDBMS rather than a proprietary database management system.

migration repository

A data store in Oracle Database that Oracle SQL Developer uses to manage the metadata for the source and target schema models during a migration. Multiple migration repositories can be used to migrate from several databases to Oracle Database at the same time.

Oracle Database Gateways

A set of Oracle products that support data integration with non-Oracle systems synchronously using consistent APIs.

Oracle GoldenGate

An Oracle product that supports modular, transaction-level data integration between diverse data sources that are stored in SQL Server, Sybase, DB2, Oracle, and other databases.

Oracle SQL*Loader

A fast, flexible, and free Oracle utility that supports loading data from flat files into Oracle Database. It supports several data formats and many different encodings. It also supports parallel data loading.

Oracle SQL Developer Migration Wizard

An Oracle tool that enables the migration of a third-party database to an Oracle database in batch mode. Migration includes data, schemas, objects, triggers, and stored procedures.

SQL dialect

A variation or extension of SQL implemented by a database vendor. When migrating client applications from third-party databases to Oracle, all non-Oracle SQL statements must be translated into Oracle SQL. Because these non-Oracle SQL statements are embedded within the source code of client applications, locating and translating them is a time-consuming, manual task. This release enhances the Oracle database to accept non-Oracle SQL statements from external vendors, and translate them automatically at run time before execution.

SQL Translation Profile

A database schema object that directs how non-Oracle SQL statements are translated into Oracle SQL dialects. This schema also contains translations of error codes, SQLSTATEs, and error messages to be returned when errors occur during the SQL execution.

When migrating a client application with non-Oracle SQL statements to Oracle, the user creates a SQL Translation Profile and configures it to translate the SQL statements and errors for the application. At run time, the application sets the translation profile in the Oracle database to translate its SQL statements and errors.

SQL Translator

The SQL Translator is a software component, provided by Oracle or third-party vendors, which can be installed in Oracle Database. It translates the SQL statements

of a client program before they are processed by the Oracle Database SQL compiler. If an error results from translated SQL statement execution, then Oracle Database SQL compiler generates an Oracle error message.

SQLSTATE

A status parameter defined by the ANSI SQL standard. It is a 5-character string that indicates the status of a SQL operation. Some of these values are:

- *00XXX*: Unqualified Successful Completion

- *01XXX*: Warning

- *02XXX*: No Data

- *07XXX*: Dynamic SQL Error

- *08XXX*: Connection Exception

- *09XXX*: Triggered Action Exception

Index

ORACLE®