Oracle® Database

SODA for Java Developer's Guide

Release 1.0

E58124-08

November 2016

Oracle Database SODA for Java Developer's Guide, Release 1.0

E58124-08

Copyright © 2014, 2016, Oracle and/or its affiliates. All rights reserved.

Primary Author: Drew Adams

Contributors: Sheila Moore, Maxim Orgiyan, Josh Spiegel

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

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

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

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

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

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

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

Contents

List of Examples

List of Tables

Preface

This document explains how to use Simple Oracle Document Access (SODA) for Java.

Audience

This document is intended for users of SODA for Java.

Documentation Accessibility

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

Access to Oracle Support

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

Related Documents

For more information, see the SODA Javadoc.

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.
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 SODA for Java

The Oracle SODA for Java API is described, including how to install and use it. The content here assumes that you are familiar with Java, JSON, and Oracle Database. The code samples here are Java code.

For information about JSON in Oracle Database, see *Oracle Database JSON Developer's Guide*.

Note:

SODA for Java supports the version of JSON described in RFC 4627. For further details, see Creating and Using Documents with SODA for Java (page 1-10).

Topics

1.1 SODA for Java Prerequisites

Before you can use SODA for Java you must configure your Java environment.

To use SODA for Java with Oracle Database:

- You must have Java Runtime Environment 1.6 (JRE 1.6).
- The following Java archive (JAR) files must be either in your `CLASSPATH` environment variable or passed using command-line option `classpath`:
  - `orajsoda.jar` (SODA for Java RDBMS implementation). Obtain the latest version at `https://github.com/oracle/soda-for-java/releases`.
  - `ojdbc6.jar` (the Oracle JDBC JAR file that is shipped with Oracle Database 12*c* Release 1 (12.1.0.2))
  - `javax.json-1.0.4.jar` (JSR353: the Java API for JSON processing)

- You must have Oracle Database 12*c* Release 1 (12.1.0.2) with Merge Label Request (MLR) bundle patch 20885778.

  Obtain the patch from My Oracle Support (`https://support.oracle.com`). Select tab **Patches & Updates**. Search for patch number 20885778, or access it directly at this URL: `https://support.oracle.com/rs?type=patch&id=20885778`.

Note:

Oracle recommends that you use AL32UTF8 (Unicode) for your database character set. Otherwise:

- Data can be altered by SODA for Java during input, because of lossy conversion to the database character set.
- Query-by-example (QBE) can return unpredictable results.

1.2 SODA for Java Overview

Simple Oracle Document Access (**SODA**) is a set of NoSQL-style APIs that let you create and store collections of documents in Oracle Database, retrieve them, and query them, without needing to know Structured Query Language (SQL) or how the data in the documents is stored in the database.

SODA for Java is a Java API that provides SODA. You can use it to perform create, read (retrieve), update, and delete (CRUD) operations on documents of any kind, and you can use it to query JSON documents.

Oracle relational database management system (RDBMS) supports storing and querying JSON data. To access this functionality, you need structured query language (SQL) with special JSON SQL operators and Java Database Connectivity (JDBC).

SODA for Java hides the complex SQL/JDBC programming with these SODA abstractions:

- Database
- Collection
- Document

A database contains collections, and each collection contains documents. SODA for Java is designed primarily for working with JSON documents, but a document can be of any Multipurpose Internet Mail Extensions (MIME) type.

SODA for Java provides CRUD operations on collections. These operations are transparently translated to SQL with JSON SQL operators and are executed by JDBC.

A (SODA) *database* is analogous to an Oracle Database schema, a *collection* is analogous to a table, and a *document* is analogous to a table row with one column for the document key (unique document identifier) and another column for the document content.

The remaining topics of this document describe various features of SODA for Java. For detailed information about specific Java methods, see the SODA for Java Javadoc.

1.3 Using SODA for Java

How to access SODA for Java is described, as well as how to use it to perform create, read (retrieve), update, and delete (CRUD) operations on collections.

(CRUD operations are also called “read and write operations” in this document.)

Topics

1.3.1 Getting Started with SODA for Java

How to access SODA for Java is described, as well as how to use it to create a database collection, insert a document into a collection, and retrieve a document from a collection.

Follow these steps to get started with SODA for Java:

1. Ensure that all of the prerequisites have been met for using SODA for Java. See SODA for Java Prerequisites (page 1-1).

2. Identify the database schema (user account) used to store collections, and grant database role `SODA_APP` to that schema:

   ```
   GRANT SODA_APP TO schemaName;
   ```

3. Place all required jar files and file `testSoda.java` (which contains the text in Example 1-1 (page 1-4)) into a directory.

4. In `testSoda.java`:

- Replace `hostName`, `port`, and `serviceName` with the hostname, port, and service name for your Oracle RDBMS instance.
- Replace `schemaName` and `password` with the name and password of the database schema identified in step 2. It will store the collection created in Example 1-1 (page 1-4).

5. Use the `cd` command to go to the directory that contains the jar files and file `testSoda.java`.
6. Execute these commands:

   ```
   javac -classpath "*" testSoda.java
   java -classpath "*:." testSoda
   ```

   Instead of the second of these commands, you can optionally use the following command. It has the additional effect of dropping the collection, cleaning up the database table that is used to store the collection and its metadata.

   ```
   java -classpath "*:." testSoda drop
   ```

   Using argument `drop` here has the effect of invoking method `drop()`, which is the proper way to drop a collection.

Caution:

Do *not* use SQL to drop the database table that underlies a collection. In addition to the documents that are stored in its table, a collection has metadata, which is also persisted in Oracle Database. Dropping the collection table does *not* also drop the associated metadata.

To work with SODA for Java you must first open a JDBC connection. This is illustrated in Example 1-1 (page 1-4). For details of how to open a JDBC connection, see *Oracle Database JDBC Developer's Guide*.

Example 1-1 testSoda.java

In this example, replace placeholders `hostName`, `port`, `schemaName`, and `password` with appropriate information for your database instance.

```
import java.sql.Connection;
import java.sql.DriverManager;

import oracle.soda.rdbms.OracleRDBMSClient;

import oracle.soda.OracleDatabase;
import oracle.soda.OracleCursor;
import oracle.soda.OracleCollection;
import oracle.soda.OracleDocument;
import oracle.soda.OracleException;

import java.util.Properties;

import oracle.jdbc.OracleConnection;

public class testSoda
{
  public static void main(String[] arg)
  {
      // Set up the JDBC connection string, schemaName, and password.
      // Replace with info appropriate for your Oracle Database instance.
      String url = "jdbc:oracle:thin:@//hostName:port/serviceName";
      Properties props = new Properties();
```

```
    props.setProperty("user", schemaName);
    props.setProperty("password", password);

    OracleConnection conn = null;

    try
    {
      // Get a JDBC connection to an Oracle instance.
      conn = (OracleConnection) DriverManager.getConnection(url, props);

      // Enable JDBC implicit statement caching
      conn.setImplicitCachingEnabled(true);
      conn.setStatementCacheSize(50);

      // Get an OracleRDBMSClient - starting point of SODA for Java application.
      OracleRDBMSClient cl = new OracleRDBMSClient();

      // Get a database.
      OracleDatabase db = cl.getDatabase(conn);

      // Create a collection with the name "MyJSONCollection".
      // This creates a database table, also named "MyJSONCollection", to store the collection.
      OracleCollection col = db.admin().createCollection("MyJSONCollection");

      // Create a JSON document.
      OracleDocument doc =
        db.createDocumentFromString("{ \"name\" : \"Alexander\" }");

      // Insert the document into a collection.
      col.insert(doc);

      // Find all documents in the collection.
      OracleCursor c = null;

      try
      {
        c = col.find().getCursor();
        OracleDocument resultDoc;

        while (c.hasNext())
        {
          // Get the next document.
          resultDoc = c.next();

          // Print document components
          System.out.println ("Key:         " + resultDoc.getKey());
          System.out.println ("Content:     " + resultDoc.getContentAsString());
          System.out.println ("Version:     " + resultDoc.getVersion());
          System.out.println ("Last modified: " + resultDoc.getLastModified());
          System.out.println ("Created on:    " + resultDoc.getCreatedOn());
          System.out.println ("Media:         " + resultDoc.getMediaType());
          System.out.println ("\n");
        }
      }
      finally
      {
        // IMPORTANT: YOU MUST CLOSE THE CURSOR TO RELEASE RESOURCES.
        if (c != null) c.close();
      }

      // Drop the collection, deleting the table underlying it and the collection metadata.
      if (arg.length > 0 && arg[0].equals("drop")) {
        col.admin().drop();
        System.out.println ("\nCollection dropped");
      }
  }
  // SODA for Java throws a checked OracleException
  catch (OracleException e) { e.printStackTrace(); }
  catch (Exception e) { e.printStackTrace(); }
```

```
    finally
    {
      try { if (conn != null)  conn.close(); }
      catch (Exception e) { }
    }
  }
}
```

1.3.2 Creating a New Document Collection with SODA for Java

How to use SODA for Java to create a new document collection is explained.

In your Java application, first create an `OracleRDBMSClient` object, which is the starting point for any Java application working with SODA for Java:

```
OracleRDBMSClient client = new OracleRDBMSClient();
```

Caution:

The `OracleRDBMSClient` object, *`client`*, is thread-safe. Other SODA for Java interfaces are *not* thread-safe, however — do not share them among multiple threads.

Next, pass the JDBC connection to method `OracleClient.getDatabase()`, to obtain an `OracleDatabase` object:

```
OracleDatabase db = client.getDatabase(jdbcConnection);
```

Note:

Oracle recommends that you *enable implicit statement caching* for the JDBC connection that you pass to SODA. This can improve the performance of read and write operations. The underlying implementation of read and write operations generates JDBC prepared statements.

If you do not enable implicit caching then each time a read or write operation is created a new JDBC prepared statement is constructed. With implicit caching enabled, a new JDBC prepared statement is created only if it is not already in the cache.

See also: *Oracle Database JDBC Developer's Guide* and *Oracle Universal Connection Pool for JDBC Developer's Guide*

Collection creation methods are available on interface `OracleDatabaseAdmin`. To access this interface, invoke method `admin()` on the `OracleDatabase` object:

```
OracleDatabaseAdmin dbAdmin = db.admin();
```

Now you can create a collection — an `OracleCollection` object — using the following code, where *`collectionName`* is the name of the collection:

```
OracleCollection col =  dbAdmin.createCollection("collectionName");
```

Method `createCollection(String` *`collectionName`*`)` creates the following in Oracle Database:

- Persistent default collection metadata.

- A table for storing the collection, in the schema with which the input JDBC connection is configured.

  By default, the table name is derived from the collection name. If you want a different table name from that provided by default then use custom collection metadata to explicitly provide the table name (see below).

  The *default table name* is derived from the collection name as follows:

  1. Each ASCII control character and double quotation mark character (`"`) in the collection name is replaced by an underscore character (_).
  2. If *all* of the following conditions apply, then all letters in the name are converted to uppercase, to provide the table name. In this case, you need not quote the table name in SQL code; otherwise, you must quote it.
     - The letters in the name are either all lowercase or all uppercase.
     - The name begins with an ASCII letter.
     - Each character in the name is alphanumeric ASCII, an underscore (_), a dollar sign ($), or a number sign (#).

     Note:

     Oracle recommends that you do *not* use dollar signs ($) or number signs (#) in Oracle identifier names.

  For example:

  - Collection names "col" and "COL" both result in a table named "COL". When used in SQL, the table name is interpreted case-insensitively, so it need not be enclosed in double quotation marks (`"`).
  - Collection name "myCol" results in a table named "myCol". When used in SQL, the table name is interpreted case-sensitively, so it must be enclosed in double quotation marks (`"`).

    Note:

    If the table name used by method `createCollection` names an *existing* table, in the schema with which the JDBC connection is configured, then the method tries to map that table to the collection. This behavior includes the default case, where the table name is derived from the collection name.

The *default collection metadata* has the following characteristics.

- Each document in the collection has these document components:
  - Key
  - Content
  - Creation timestamp
  - Last-modified timestamp
  - Version

- The collection can store only JSON documents.
- Document keys are automatically generated for documents that you add to the collection.

The default collection configuration is recommended in most cases, but collections are highly configurable. When you create a collection you can specify things such as the following:

- Storage details, such as the name of the table that stores the collection and the names and data types of its columns.
- The presence or absence of columns for creation timestamp, last-modified timestamp, and version.
- Whether the collection can store only JSON documents.
- Methods of document key generation, and whether document keys are client-assigned or generated automatically.
- Methods of version generation.

This configurability also lets you map a new collection to an existing table.

To configure a collection in a nondefault way, you must create a JSON `OracleDocument` instance of custom collection metadata and pass it to method `createCollection(String` *`collectionName`*`, OracleDocument` *`collectionMetadata`*`)`. To build and generate this `OracleDocument` instance easily, you can use `OracleRDBMSMetadataBuilder`.

If you do not care about the details of collection storage and configuration, then use method `createCollection(`*`collectionName`*`)`, as in Example 1-2 (page 1-9).

You can search or change a collection only if it is open. A newly created collection is open for the life of your session.

Note:

Unless otherwise stated, the remainder of this documentation assumes that a collection has the default configuration.

See Also:

- Opening an Existing Document Collection with SODA for Java (page 1-8)
- SODA Collection Configuration Using Custom Metadata (page 1-41) for information about creating custom metadata and using `OracleRDBMSMetadataBuilder`

1.3.3 Opening an Existing Document Collection with SODA for Java

You can use `OracleDatabase` method `openCollection()` to open an existing document collection or to test whether a given name names an existing collection.

Example 1-2 (page 1-9) opens the collection named `myCollectionName` and returns the `OracleCollection` object that represents this collection. If the value returned is `null` then there is no existing collection named `myCollectionName`.

Example 1-2 Opening an Existing Document Collection

```
OracleCollection ocol = db.openCollection("myCollectionName");
```

1.3.4 Checking Whether a Given Collection Exists with SODA for Java

You can use `OracleDatabase` method `openCollection()` to check for the existence of a given collection. It returns `null` if the collection argument does not name an existing collection; otherwise, it opens the collection having that name.

In Example 1-2 (page 1-9), if `myCollectionName` does not name an existing collection then `ocol` is assigned the value `null`.

1.3.5 Discovering Existing Collections with SODA for Java

You can use `OracleDatabaseAdmin` method `getCollectionNames` to discover existing collections.

Example 1-3 (page 1-9) illustrates this, using method `getCollectionNames` with the simplest signature, which accepts no arguments. The example prints the names of all existing collections.

Example 1-3 Printing the Names of All Existing Collections

```
List<String> names =  db.admin().getCollectionNames();

for (String name : names)
 System.out.println ("Collection name: " + name);
```

1.3.6 Dropping a Document Collection with SODA for Java

You use `OracleCollectionAdmin` method `drop()` to drop a document collection.

Example 1-4 (page 1-9) drops collection `col`.

Caution:

This is the proper way to drop a collection — use method `drop()`. Do *not* use SQL to drop the database table that underlies the collection. Collections have persisted metadata, in addition to the documents that are stored in the collection table.

Note:

Commit all writes to a collection before using method `drop()`. For `drop()` to succeed, all uncommitted writes to the collection must first be committed. Otherwise, an exception is raised.

Example 1-4 Dropping a Document Collection

```
col.admin().drop();
```

1.3.6.1 If You Need To Drop and Re-Create a Collection...

Day-to-day use of a typical application that makes use of SODA does not require you to drop and re-create collections. But if you do need to do that for any reason then be aware of the important guideline presented here.

Do *not* drop a collection and then re-create it with *different metadata* if there is any application running that uses SODA objects. Shut down any such applications before re-creating the collection, so that all live SODA objects are released.

There is no problem just dropping a collection. Any read or write operation on a stale `OracleCollection` object that corresponds to a dropped collection raises an error. And there is no problem dropping a collection and then re-creating it with the same metadata.

But if you re-create a collection with different metadata, and if there are any live applications using SODA objects, then there is a risk that a stale collection object (an `OracleCollection` instance) is accessed, and no error is raised in this case.

This risk is increased if collection metadata is cached. If caching is enabled, a (shared or local) cache can return an entry for a stale collection object even if the collection has been dropped.

See Also:

SODA Collection Metadata Caching (page 1-40)

1.3.7 Creating and Using Documents with SODA for Java

Creation and use of documents by SODA for Java are described.

SODA for Java represents a document using Java interface `OracleDocument`. This interface is designed primarily to represent JSON documents, but it also supports other content types. An `OracleDocument` is simply a carrier of content.

To create JSON content for an `OracleDocument` instance, you can use your favorite package — for example, JSR353, the Java API for JSON processing (`https://jsonp.java.net/`). Here is an example of a simple JSON document:

```
{ "name" :    "Alexander",
  "address" : "1234 Main Street",
  "city" :    "Anytown",
  "state" :   "CA",
  "zip" :     "12345"
}
```

Note:

In SODA for Java, JSON content must conform to RFC 4627. In particular, JSON content must be either an object (as in the preceding example) or an array; it cannot be a scalar value. For example, according to RFC 4627, the string value `"hello"` is not, by itself, valid JSON content.

Also in SODA for Java, JSON content encoding must be either UTF-8 or UTF-16 (big endian (BE) or little endian (LE)). Although RFC 4627 also allows UTF-32 (BE and LE) encodings, SODA for Java does not support them.

To create an `OracleDocument` instance from content that is represented as a byte array or a `String` instance, use the following methods (which `OracleDatabase` inherits from `OracleDocumentFactory`), respectively:

- `createDocumentFromByteArray()`
- `createDocumentFromString()`

Note:

Documents used with SODA for Java are limited to approximately 2 gigabytes.

A document has these components:

- Key
- Content
- Creation time stamp
- Last-modified time stamp
- Version
- Media type (`"application/json"` for JSON documents)

Interface `OracleDocument` provides getter methods for accessing document components. If a document is missing a given component, then the corresponding getter method returns `null`.

When you create a document by invoking method `createDocumentFromString()` or `createDocumentFromByteArray()`:

- You might need to provide the document key as a method argument.

  In a collection, each document must have a key. You must provide the key when you create the document *only* if you expect to insert the document into a collection that does *not* automatically generate keys for inserted documents. By default, collections are configured to automatically generate document keys.

- You can provide the document content as a method argument (the `content` parameter is required, but its value can be `null`).
- The method sets the values of the creation time stamp, last-modified time stamp, and version to `null`.

Methods `createDocumentFromString()` and `createDocumentFromByteArray()` each have multiple variants:

- The simplest variant accepts only document content. The media type defaults to `"application/json"`, and the other components default to `null`. This variant is useful for creating documents for insertion into collections that automatically generate document keys.
- Another variant accepts both document key and document content. The media type defaults to `"application/json"`, and the other components default to `null`. This variant is useful for creating documents for insertion into collections that have client-assigned document keys.

- The most flexible (and most verbose) variant accepts key, content, and content type. Because it lets you specify content type, this variant is useful for creating non-JSON documents.

Example 1-5 (page 1-12) creates an `OracleDocument` instance with content only. The media type defaults to `"application/json"`, and the other document components default to `null`.

Example 1-6 (page 1-12) creates an `OracleDocument` instance with document key and content. The media type defaults to `"application/json"`, and the other document components default to `null`.

You write documents to collections using SODA for Java write operations, and you read documents from collections using SODA for Java read operations. The SODA for Java read and write operations are described in the following topics:

- Inserting Documents into Collections with SODA for Java (page 1-13) (write)
- Saving Documents into Collections with SODA for Java (page 1-14) (write)
- Finding Documents in Collections with SODA for Java (page 1-15) (read)
- Replacing Documents in a Collection with SODA for Java (page 1-19) (write)
- Removing Documents from a Collection with SODA for Java (page 1-20) (write)

See Also:

- `OracleDocumentFactory` Javadoc for more information about methods`createDocumentFromString()` and `createDocumentFromByteArray()`
- `OracleDocument` Javadoc for more information about getter methods

Example 1-5 Creating a Document with JSON Content

```
OracleDocument doc =
  odb.createDocumentFromString("{ \"name\" : \"Alexander\"}");

// Get the content
String content = doc.getContentAsString();

// Get the content type (it is "application/json")
String contentType = doc.getContentType();
```

Example 1-6 Creating a Document with Document Key and JSON Content

```
OracleDocument doc
  = odb.createDocumentFromString("myKey", "{ \"name\" : \"Alexander\"}");
```

1.3.8 Handling Transactions with SODA for Java

You can cause SODA for Java to treat individual read and write operations, or groups of them, as a single transaction.

The JDBC connection that you pass to method `OracleClient.getDatabase()` has auto-commit mode either on or off.

If auto-commit mode is *on*, then each SODA for Java read operation and write operation is treated as a single transaction. If the operation succeeds, then the transaction automatically commits. If the operation fails, then an `OracleException` or `RuntimeException` is thrown, and the transaction automatically rolls back. SODA for Java itself throws only checked exceptions (`OracleException` and exceptions derived from `OracleException`). However, SODA for Java is built upon JDBC, which can throw a `RuntimeException` that SODA for Java passes through.

If auto-commit is *off*, then you can combine multiple SODA for Java read or write operations into one transaction. If the transaction succeeds, then your application must explicitly commit it, by calling method `commit()` on the JDBC connection. If the transaction fails, then an `OracleException` or `RuntimeException`, is thrown. Your application must handle the exception and explicitly roll back the transaction, by invoking method `rollback()` on the JDBC connection. (`RuntimeException` can be thrown only by JDBC, as mentioned in the preceding paragraph.)

To facilitate transactional programming, SODA for Java supports optimistic locking.

See Also:

- Replacing Documents in a Collection with SODA for Java (page 1-19)
- Removing Documents from a Collection with SODA for Java (page 1-20)

1.3.9 Inserting Documents into Collections with SODA for Java

To insert a document into a collection, you invoke `OracleCollection` method `insert(OracleDocument)` or `insertAndGet(OracleDocument)`. These methods create document keys automatically, unless the collection is configured with client-assigned keys and the input document provides the key.

Method `insert(OracleDocument)` only inserts the document into the collection. Method `insertAndGet(OracleDocument)` also returns a result document, which contains the document key and any other generated document components (except the content).

Both methods automatically set the values of the creation time stamp, last-modified time stamp, and version (if the collection is configured to include these components and to generate the version automatically, as is the case by default).

Note:

If the collection is configured with client-assigned document keys (which is not the default case), and the input document provides a key that identifies an existing document in the collection, then these methods throw an exception. If you want the input document to *replace* the existing document instead of causing an exception, see Saving Documents into Collections with SODA for Java (page 1-14).

Example 1-7 (page 1-14) creates a document and inserts it into a collection using method `insert()`.

Example 1-8 (page 1-14) creates a document, inserts it into a collection using method `insertAndGet()`, and then gets each of the generated components from the result document (which contains them).

To efficiently insert a large number of documents into a collection, invoke `OracleCollection` method `insert(Iterator<OracleDocument>)` or `insertAndGet(Iterator<OracleDocument>)`. These methods are analogous to `insert(OracleDocument)` and `insertAndGet(OracleDocument)`, but instead of handling a single document, they handle multiple documents. Parameter `Iterator<oracleDocument>` is an iterator over multiple input documents.

Method `insertAndGet(Iterator<OracleDocument>)` returns a list of result documents — one `OracleDocument` instance for each input document. Each such result document contains the document key and any other generated document components (except the content). The order of the result documents corresponds to the order of input documents, allowing correlation of result and input documents.

See Also:

`OracleCollection` Javadoc for more information about methods `insert(OracleDocument)`, `insertAndGet(OracleDocument)`, `insert(Iterator<OracleDocument>)`, and `insertAndGet(Iterator<OracleDocument>)`

Example 1-7 Inserting a Document into a Collection

```
OracleDocument doc =
  db.createDocumentFromString("{ \"name\" : \"Alexander\"}");

col.insert(doc);
```

Example 1-8 Inserting a Document into a Collection and Getting the Result Document

```
OracleDocument doc =
  db.createDocumentFromString("{ \"name\" : \"Alexander\"}");

OracleDocument insertedDoc = col.insertAndGet(doc);

// Get the generated document key
String key = insertedDoc.getKey();

// Get the generated creation timestamp
String createdOn = insertedDoc.getCreatedOn();

// Get the generated last-modified timestamp
String lastModified = insertedDoc.getLastModified();

// Get the generated version
String version = insertedDoc.getVersion();
```

1.3.10 Saving Documents into Collections with SODA for Java

You use `OracleCollection` methods `save(OracleDocument)` and `saveAndGet(OracleDocument)` to save documents into collections.

These methods are similar to methods `insert(OracleDocument)` and `insertAndGet(OracleDocument)`except that, if the collection is configured with client-assigned document keys and the input document provides a key that already identifies a document in the collection, then the input document *replaces* the existing document. (Methods `insert(OracleDocument)` and `insertAndGet(OracleDocument)` throw an exception in that case.)

Note:

By default, collections are configured with automatically generated document keys. Therefore, for a default collection, methods `save(OracleDocument)` and `saveAndGet(OracleDocument)` are equivalent to methods `insert(OracleDocument)` and `insertAndGet(OracleDocument)`, respectively.

Example 1-9 (page 1-15) saves a document into a collection that is configured with client-assigned document keys, using method `saveAndGet()`. It then gets the key and the generated document components (except the content) from the result document (which contains them).

See Also:

`OracleCollection` Javadoc for more information about methods `save(OracleDocument)` and `saveAndGet(OracleDocument)`

Example 1-9 Saving a Document into a Collection

```
OracleRDBMSClient cl = new OracleRDBMSClient();
OracleDatabase db = ...

// Configures the collection with client-assigned document keys
OracleDocument collMeta =
  cl.createMetadataBuilder().keyColumnAssignmentMethod("client").build();
OracleCollection cKeyColl = db.createCollection("collectionName", collMeta);

// For a collection configured with client-assigned document keys,
// you must provide the key for the input document.
OracleDocument cKeyDoc =
  db.createDocumentFromString("myKey", "{ \"name\" : \"Alexander\"}");

// If key "myKey" already identifies a document in the collection
// then cKeyDoc replaces the existing doc.
OracleDocument savedDoc = clientKeysColl.saveAndGet(cKeyDoc);

// Get document key ("myKey")
String key = savedDoc.getKey();

// Get the generated creation timestamp
String createdOn = savedDoc.getCreatedOn();

// Get the generated last-modified timestamp
String lastModified = savedDoc.getLastModified();

// Get the generated version
String version = savedDoc.getVersion();
```

1.3.11 Finding Documents in Collections with SODA for Java

To find documents in a collection, you invoke `OracleCollection` method `find()`, which returns an `OracleOperationBuilder` object that represents a query that finds all documents in the collection.

To execute the query, obtain a cursor for its results by invoking `OracleOperationBuilder` method `getCursor()`. Then use the cursor to visit each document in the result list. To determine whether the result list has a next document, and to obtain the next document, invoke `OracleCursor` methods `hasNext()` and `next()`, respectively. This is illustrated by Example 1-10 (page 1-16) and other examples here.

However, you typically do not work directly with the `OracleOperationBuilder` object. Instead, you *chain together* some of its methods, to specify various find operations. This is illustrated in the other examples here, which find documents by their keys or using query-by-example (QBE) filter specifications.

Note:

Examples here that use method `getContentAsString()` assume that all documents in the collection are JSON documents. If they are not, this method throws an exception.

See Also:

- OracleOperationBuilder Methods, Terminal and Nonterminal (page 1-21) for information about `OracleOperationBuilder` methods and chaining them together
- Replacing Documents in a Collection with SODA for Java (page 1-19) and Removing Documents from a Collection with SODA for Java (page 1-20) for information about using terminal `OracleOperationBuilder` methods for write operations
- Using Filter Specifications (QBEs) with SODA for Java (page 1-22) for information about queries that can be expressed as filter specifications

Example 1-10   Finding All Documents in a Collection

This example first obtains a cursor for a query result list that contains each document in a collection. It then uses the cursor in a `while` statement to get and print the content of each document in the result list, as a string. Finally, it closes the cursor.

Note:

To avoid resource leaks, *close* any cursor that you no longer need.

```
OracleCursor c = col.find().getCursor();

while (c.hasNext()) {
  OralceDocument resultDoc = c.next();
  System.out.println("Document content: " + resultDoc.getContentAsString());
}

// IMPORTANT: You must close the cursor to release resources!
c.close;
```

Example 1-11   Finding the Unique Document That Has a Given Document Key

This example chains together `OracleOperationBuilder` methods to specify an operation that finds the unique document whose key is `"key1"`. It uses nonterminal method `key()` to specify the document. It then uses terminal method `getOne()` to execute the read operation and return the document (or `null` if no such document is found).

```
OracleDocument doc = col.find().key("key1").getOne();
```

Example 1-12 Finding Multiple Documents with Specified Document Keys

This example defines `HashSet` **`myKeys`**, with (string) keys `"key1"`, `"key2"`, and `"key3"`. It then finds the documents that have those keys, and it prints the key and content of each of those documents.

Nonterminal method `keys()` specifies the documents with the given keys. Terminal method `getCursor()` executes the read operation and returns a cursor over the result documents.

Note:

The maximum number of keys in the set supplied to method `keys()` must not exceed 1000.

```
Set<String> myKeys = new HashSet<String>();
myKeys.put("key1");
myKeys.put("key2");
myKeys.put("key3");

OracleCursor c = col.find().keys(myKeys).getCursor();

while (c.hasNext(()) {
  OracleDocument resultDoc = c.next();

  // Print the document key and document content
  System.out.println ("Document key: " + resultDoc.getKey() + "\n" +
                        " document content: " + resultDoc.getContentAsString());
}

c.close();
```

Example 1-13 Finding Documents with a Filter Specification

Nonterminal method `filter()` provides a powerful way to filter JSON documents in a collection. Its `OracleDocument` parameter is a JSON query-by-example (QBE, also called a filter specification).

This example does the following:

1. Creates a filter specification that looks for all JSON documents whose `name` field has value `"Alexander"`.
2. Uses the filter specification to find the matching documents.
3. Prints the key and content of each document.

```
// Create the filter specification
OracleDocument filterSpec =
  db.createDocumentFromString("{ /"name/" : /"Alexander/"}");

OracleCursor c = col.find().filter(filterSpec).getCursor();

while (c.hasNext(()) {
  OracleDocument resultDoc = c.next();

  // Print the document key and document content
  System.out.println ("Document key: " + resultDoc.getKey() + "\n" +
                      " document content: " + resultDoc.getContent());
}

c.close();
```

Example 1-14 Specifying Pagination Queries with Methods skip() and limit()

This example uses nonterminal methods `skip()` and `limit()` in a pagination query. (Filter specification `filterSpec` is from Example 1-13 (page 1-17).)

```
// Find all documents matching the filterSpec, skip the first 1000,
// and limit the number of returned documents to 100.
OracleCursor c =
  col.find().filter(filterSpec).skip(1000).limit(100).getCursor();

while (c.hasNext(()) {
  OracleDocument resultDoc = c.next();

  // Print the document key and document content
  System.out.println ("Document key: " + resultDoc.getKey() + "\n" +
                      " document content: " + resultDoc.getContent());
}

c.close();
```

Example 1-15 Specifying Document Version

Nonterminal method `version()` specifies the document version. It is useful for implementing optimistic locking, when used with the terminal methods for write operations.

Nonterminal method `headerOnly()` specifies the return of document headers only. A document header has all the document components except the content.

```
// Find a document with key "key1" and version "version1".
OracleDocument doc = col.find().key("key1").version("version1").getOne();
```

Example 1-16 Finding Documents and Returning Only Their Headers

This example finds all documents with the specified document keys and returns only their headers. (The keys are those in `HashSet myKeys`, which is defined in Example 1-12 (page 1-17).)

```
// Find all documents matching the keys in HashSet myKeys.
// For each document, return all document components except the content.
OracleCursor c = col.find().keys(myKeys).headerOnly().getCursor();
```

Example 1-17 Counting the Number of Documents Found

This example uses terminal method `count()` to get a count of all of the documents in the collection. It then gets a count of all of the documents that are returned by the filter specification `filterSpec` from Example 1-13 (page 1-17).

```
// Get a count of all documents in the collection
int numDocs = col.find().count();

// Get a count of all documents in the collection that match a filter spec
numDocs = col.find().filter(filterSpec).count();
```

1.3.12 Replacing Documents in a Collection with SODA for Java

To replace the content of one document in a collection with the content of another, you chain together `OracleOperationBuilder` method `key(String)` with either method **`replaceOne`**`(OracleDocument)` or method **`replaceOneAndGet`**`(OracleDocument)`. Method `replaceOne(OracleDocument)` only replaces the document. Method `replaceOneAndGet(OracleDocument)` also returns a result document, which contains all document components except the content.

Both `replaceOne(OracleDocument)` and `replaceOneAndGet(OracleDocument)` update the values of the last-modified timestamp and the version. Replacement does *not* change the document key or the creation timestamp.

Note:

Some version-generation methods, including the default method, generate hash values of the document content. In such a case, if the document content does not change then neither does the version. For more information about version-generation methods, see SODA Collection Configuration Using Custom Metadata (page 1-41).

See Also:

- `OracleOperationBuilder` Javadoc for more information about `replaceOne()` and `replaceOneAndGet()`
- OracleOperationBuilder Methods, Terminal and Nonterminal (page 1-21) for information about `OracleOperationBuilder` methods and chaining them together

Example 1-18   Replacing a Document in a Collection and Getting the Result Document

This example replaces a document in a collection, gets the result document, and gets the generated components from the result document.

```
OracleDocument newDoc = ...
OracleDocument resultDoc = col.find().key("k1").replaceOneAndGet(newDoc);

if (resultDoc != null)
{
  // Get the generated document key (unchanged by replacement operation)
  String key = resultDoc.getKey();

  // Get the generated version
  String version = resultDoc.getVersion();

  // Get the generated last-modified timestamp
  String lastModified = resultDoc.getLastModified();

  // Get the creation timestamp (unchanged by replacement operation)
  String createdOn = resultDoc.getCreatedOn();
}
```

Example 1-19 Replacing a Document Only If the Version Has Not Changed

To implement optimistic locking when replacing a document, you can chain together methods `key()` and `version()`, as in this example.

```
OracleDocument resultDoc =
  col.find().key("k1").version("v1").replaceOneAndGet(newDoc);
```

1.3.13 Removing Documents from a Collection with SODA for Java

To remove a document from a collection, you chain together (1) `OracleCollection` method `find()` with these `OracleOperationBuilder` methods: (2) `key()`, `keys()`, or `filter()`; (3) `version()` (optional); and (4) `remove()`. Examples are provided.

See Also:

- `OracleOperationBuilder` Javadoc for more information about `key()`, `keys()`, `filter()`, `version()`, and `remove()`
- OracleOperationBuilder Methods, Terminal and Nonterminal (page 1-21) for information about `OracleOperationBuilder` methods and chaining them together
- Using Filter Specifications (QBEs) with SODA for Java (page 1-22)

Example 1-20 Removing a Document from a Collection Using a Document Key

This example removes the document whose document key is `"k1"`. The number of documents removed is returned.

```
// Count is 1, if the document with key "k1" is found in the collection.
// Count is 0, otherwise.
int count = col.find().key("k1").remove();
```

Example 1-21 Removing a Document Only If the Version Has Not Changed

To implement optimistic locking when removing a document, you can chain together methods `key()` and `version()`, as in this example.

```
col.find().key("k1").version("v1").remove();
```

Example 1-22 Removing Documents from a Collection Using Document Keys

This example removes the documents whose keys are `"k1"` and `"k2"`.

```
Set<String> myKeys = new HashSet<String>();
myKeys.add("k1");
myKeys.add("k2");

// Count is 2 if two documents with keys "k1" and "k2"
// were found in the collection.
int count = col.find().keys(myKeys).remove();
```

Example 1-23 Removing JSON Documents from a Collection Using a Filter

This example uses a filter to remove the JSON documents whose `greeting` field has value `"hello"`. It then prints the number of documents removed.

```
OracleDocument filterSpec =
   db.createDocumentFromString("{ \"greeting\" : \"hello\" }");

int count = col.find().filter(filterSpec).remove();

// Print the number of documents removed
System.out.println ("Removed " + count + " documents"):
```

1.3.14 OracleOperationBuilder Methods, Terminal and Nonterminal

You can chain together `OracleOperationBuilder` methods, to specify read and write operations against a collection.

`OracleOperationBuilder` provides the following nonterminal methods, which you can chain together to specify a read or write operation: `key()`, `keys()`, `filter()`, `version()`, `skip()`, `limit()`, and `headerOnly()`.

These are called **nonterminal** methods because they return the same `OracleOperationBuilder` object on which they are invoked, which allows them to be chained together. Nonterminal methods let you specify parts of an operation; they do not create or execute an operation.

`OracleOperationBuilder` also provides terminal methods. A **terminal** method always appears at the end of a method chain, and it creates and executes the operation.

The terminal methods for *read* operations are `getCursor()`, `getOne()`, and `count()`. The terminal methods for *write* operations are `replaceOne()`, `replaceOneAndGet()`, and `remove()`.

Note:

If you use `OracleCursor` method `next()` or `OracleOperationBuilder` method `getOne()`, and if the underlying document is larger than 2 gigabytes, then an exception is thrown.

Unless the Javadoc documentation for a method states otherwise, you can chain together any nonterminal methods, and you can end the chain with any terminal method. However, not all combinations make sense. For example, it does not make sense to chain method `version()` together with any method except `key()`, or to chain method `key()` or `keys()` together with method `filter()`.

Table 1-1 (page 1-21) briefly describes `OracleOperationBuilder` nonterminal methods for building operations against a collection.

Table 1-1 OracleOperationBuilder Nonterminal Methods

Method	Description
`key()`	Find a document that has the specified document key.
`keys()`	Find documents that have the specified document keys. The maximum number of keys passed as argument must not exceed 1000.
`filter()`	Find documents that match a filter specification (a query-by-example expressed in JSON).

Table 1-1 (Cont.) OracleOperationBuilder Nonterminal Methods

Method	Description
`version()`	Find documents that have the specified version. This is typically used with `key()`. For example: `find().key("key1").version("version1")`.
`headerOnly()`	Exclude document content from the result.
`skip()`	Skip the specified number of documents in the result.
`limit()`	Limit the number of documents in the result to the specified number.

Table 1-2 (page 1-22) briefly describes `OracleOperationBuilder` terminal methods for creating and executing read operations against a collection.

Table 1-2 OracleOperationBuilder Terminal Methods for Read Operations

Method	Description
`getOne()`	Create and execute an operation that returns at most one document — for example, an operation that includes an invocation of nonterminal method `key()`.
`getCursor()`	Get a cursor over read operation results.
`count()`	Count the number of documents found by the operation.

See Also:

- Finding Documents in Collections with SODA for Java (page 1-15) for descriptions and examples of using `OracleOperationBuilder` methods to find documents
- Replacing Documents in a Collection with SODA for Java (page 1-19) and Removing Documents from a Collection with SODA for Java (page 1-20) for descriptions and examples of using the `OracleOperationBuilder` terminal write methods
- Using Filter Specifications (QBEs) with SODA for Java (page 1-22) for information about queries that can be expressed as filter specifications
- The SODA for Java Javadoc for complete information about `OracleOperationBuilder` methods

1.3.15 Using Filter Specifications (QBEs) with SODA for Java

A filter specification, also called a query-by-example or QBE, is a SODA query that uses a pattern that is expressed in JSON. The query selects the JSON documents in a collection that satisfy it, meaning that the filter specification evaluates to true for only those documents.

QBE patterns use operators for this document selection or matching, including basic field operators, such as testing for field existence or value comparison, and logical operators, such as union (`$or`), intersection (`$and`), and negation (`$not`).

Note:

QBE is not supported on a heterogeneous collection, that is, a collection that has the media type column. Such a collection is designed for storing both JSON and non-JSON content.

See Also:

- Querying With a Filter Specification (page 1-29)
- SODA Paths (page 1-29)
- SODA Filter Specifications (QBEs) (page 1-31)
- Media Type Column Name (page 1-55)

1.3.15.1 Sample JSON Documents

A few sample JSON documents are presented here. They are referenced in some query-by-example (QBE) examples, as well as in some reference descriptions.

See Also:

- Example 1-13 (page 1-17) and Example 1-23 (page 1-20)
- Basic Field Clause (page 1-33)

Example 1-24 Sample JSON Document 1

```
{ "name" : "Jason",
  "age" : 45,
  "address" : [ { "street" : "25 A street",
                  "city" : "Mono Vista",
                  "zip" : 94088,
                  "state" : "CA" } ],
  "drinks" : "tea" }
```

Example 1-25 Sample JSON Document 2

```
{ "name" : "Mary",
  "age" : 50,
  "address" : [ { "street" : "15 C street",
                  "city" : "Mono Vista",
                  "zip" : 97090,
                  "state" : "OR" },
                { "street" : "30 ABC avenue",
                  "city" : "Markstown",
                  "zip" : 90001,
                  "state" : "CA" } ] }
```

Example 1-26 Sample JSON Document 3

```
{ "name" : "Mark",
  "age" : 65,
  "drinks" : ["soda", "tea"] }
```

1.3.15.2 Using Paths in QBEs

A query-by-example (QBE) contains zero or more paths to document fields. (In the context of a QBE, "path to a field" is often shortened informally to "field".) A path to a field can have multiple steps, and it can cross the boundaries of both objects and arrays.

For example, this QBE matches all documents where a `zip` field exists under field `address` and has value `94088`:

```
{ "address.zip" : 94088 }
```

The preceding filter specification matches sample document 1.

Paths can target particular elements of an array in a JSON document, by enclosing the array position in square brackets (`[` and `]`).

For example, path `address[1].zip` targets all `zip` fields in the second object of array `addresses.` (Array position numbers start at 0, not 1.) The following QBE matches sample document 2 because the second object of its `address` array has a `zip` field with value `90001`.

```
{ "address[1].zip" : 90001}
```

Instead of specifying a particular array position, you can specify a list of positions (for example, `[1,2]`) or a range of positions (for example, `[1 to 3]`). The following QBE matches sample document 3 because it has `"soda"` as the first element (position 0) of array `drinks`.

```
{ "drinks[0,1]" : "soda" }
```

And this QBE does not match any of the sample documents because they do not have `"soda"` as the second or third array element (position 1 or 2).

```
{ "drinks[1 to 2]" : "soda" }
```

If you do not specify an array step then `[*]` is assumed, which matches *any* array element — `*` acts as a wildcard. For example, if the value of field `drinks` is an array then the following QBE matches if the value of any array element is the string `"tea"`:

```
{"drinks" : "tea"}
```

This QBE thus matches sample documents 1 and 2. An equivalent QBE that uses the wildcard explicitly is the following:

```
{"drinks[*]" : "tea"}
```

See Also:

- SODA Paths (page 1-29)
- Sample JSON Documents (page 1-23)

1.3.15.3 Using QBE Basic Field Operators

A query-by-example (QBE) basic field operator tests whether a given field satisfies a given set of criteria. A basic field operator is either `$exists` or a comparison operator.

A comparison operator compares the value of a field with one or more other values. The comparison operators are `$eq`, `$ne`, `$gt`, `$gte`, `$lte`, `$startsWith`, `$regex`, `$in`, `$nin`, and `$all`.

One of the simplest and most useful filter specifications tests a field for equality to a specific value. For example, this filter specification matches any document that has a field `name` whose value is `"Jason"`:

```
{ "name" : { "$eq" : "Jason" } }
```

For convenience, you can omit QBE operator `$eq`. This scalar-equality filter specification is thus equivalent to the preceding one, which uses `$eq`:

```
{ "name" : "Jason" }
```

Both of the preceding filter specifications match sample document 1.

`$eq` is an example of a QBE comparison operator. You can combine multiple comparison operators in the object that is the value of a single QBE field.

For example, the following QBE uses comparison operators `$gt` and `$lt`. It matches sample document 2, because that document contains an `age` field with a value (50) that is both greater than (`$gt`) 45 and less than (`$lt)` 55.

```
{ "age" : { "$gt" : 45, "$lt" : 55 } }
```

See Also:

- Table 1-3 (page 1-34)
- Basic Field Clause (page 1-33) for more information about basic field clauses
- Sample JSON Documents (page 1-23)

1.3.15.4 Using QBE Logical Combining Operators

You use the query-by-example (QBE) logical combining operators, `$and`, `$or`, and `$nor`, to combine conditions to form more complex QBEs. Each accepts an array of conditions as its argument.

QBE logical combining operator `$and` matches a document if each condition in its array argument matches it. For example, this QBE matches sample document 1, because that document contains a field `name` whose value starts with `"Ja"`, and it contains a field `drinks` whose value is `"tea"`.

```
{"$and" : [ {"name" : {"$startsWith" : "Ja"}}, {"drinks" : "tea"} ]}
```

Often you can omit operator `$and`. For example, the following query is equivalent to the previous one:

```
{"name" : {"$startsWith" : "Ja"}, "drinks" : "tea"}
```

QBE logical combining operator `$or` matches a document if at least one of the conditions in its array argument matches it.

For example, the following QBE matches sample documents 2 and 3, because those documents contain a field `zip` under a field `address`, where the value of `zip` is less than 94000, or a field `drinks` whose value is `"soda"`, or both:

```
{"$or" : [ {"address.zip" : {"$le" : 94000}}, {"drinks" : "soda"} ]}
```

QBE logical combining operator `$nor` matches a document if no condition in its array argument matches it. (Operators `$nor` and `$or` are logical complements.)

The following query matches sample document 1, because in that document there is neither a field `zip` under a field `address`, where the value of `zip` is less than 94000 nor a field `drinks` whose value is `"soda"`:

```
{"$nor" : [{"address.zip" : {"$le" : 94000}}, {"drinks" : "soda"}]}
```

Each element in the array argument of a logical combining operator is a condition.

For example, the following condition has a single logical combining clause, with operator `$and`. The array value of `$and` has two conditions: the first condition restricts the value of field `age`. The second condition has a single logical combining clause with `$or`, and it restricts either the value of field `name` or the value of field `drinks`.

```
{ "$and" : [ { "age" : {"$gte" : 60} },
             { "$or" : [ {"name" :  "Jason"},
                         {"drinks" : {"$in" : ["tea", "soda"]}} ] } ] }
```

- The condition with the comparison for field `age` matches sample document 3.
- The condition with logical combining operator `$or` matches sample documents 1 and 3.
- The overall condition matches only sample document 3, because that is the only document that satisfies both the condition on `age` and the condition that uses `$or`.

This condition has two conditions in the array argument of operator `$or`. The first of these has a single logical combining clause with `$and`, and it restricts the values of fields `name` and `drinks`. The second has a single logical combining clause with `$nor`, and it restricts the values of fields `age` and `name`.

```
{ "$or" : [ { "$and" : [ {"name" : "Jason"},
                         {"drinks" : {"$in" : ["tea", "soda"]}} ] },
            { "$nor" : [ {"age" : {"$lt" : 65}},
                         {"name" : "Jason"} ] } ] }
```

- The condition with operator `$and` matches sample document 1.
- The condition with operator `$nor` matches sample document 3.
- The overall condition matches both sample documents 1 and 3, because each of these documents satisfies at least one condition in the `$or` argument.

See Also:

- Logical Clause (page 1-37)
- Omitting $and (page 1-37)
- Sample JSON Documents (page 1-23)

1.3.15.5 Using Logical Operator $not

You use query-by-example (QBE) logical operator `$not` to negate the value of its operand, which is either a single existence or comparison criterion. When the operand criterion is true, the `$not` clause evaluates to false; when the criterion is false, `$not` evaluates to true.

For example, this QBE matches sample documents 1 and 3: document 1 has a field matching path `address.zip` and whose value is not `"90001"`, and document 3 has no field matching path `address.zip`.

```
{"address.zip" : {"$not" : { "$eq" : "90001" }}}
```

See Also:

- Logical Clause (page 1-37)
- Sample JSON Documents (page 1-23)

1.3.15.6 Using Nested Conditions

You can use a query-by-example (QBE) with a nested condition to match a document that has a field with an array value with object elements, where a given object of the array satisfies multiple criteria.

The following condition matches documents that have both a `city` value of `"Mono Vista"` and a `state` value of`"CA"` in the *same object* under array `address`.

```
{"address" : { "city" : "Mono Vista", "state" : "CA"}}
```

It specifies that there must be a parent field `address`, and if the value of that field is an array then at least one object in the array must have a `city` field with value `"Mono Vista"` and a `state` field with value `"CA"`. Of the three sample documents, this QBE matches only sample document 1.

The following QBE also matches sample document 1, but it matches sample document 2 as well:

```
{"address.city" : "Mono Vista", "address.state" : "CA"}
```

Unlike the preceding QBE, nothing here constrains the city and state to belong to the same address. Instead, this QBE specifies only that matching documents must have a `city` field with value `"Mono Vista"` in some object of an `address` array and a `state` field with value `"CA"` in some object of an `address` array. It does not specify that fields `address.city` and `address.state` must reside within the same object.

See Also:

- Nested-Condition Clause (page 1-38)
- Sample JSON Documents (page 1-23)

1.3.15.7 Using QBE Operator $id

Other query-by-example (QBE) operators generally look for particular JSON fields within documents and try to match their values. Operator `$id` instead matches document keys. You use operator `$id` in the outermost condition of a QBE.

Example 1-27 (page 1-28)shows three QBEs that use `$id`.

Note:

As an alternative to using a `$id` condition in a SODA for Java QBE, you can use `OracleOperatorBuild` method `key()` or `keys()` to specify document keys in conjunction with method `filter()`.

See Also:

- Finding Documents in Collections with SODA for Java (page 1-15)
- ID Clause (page 1-39)

Example 1-27 Using $id To Find Documents That Have Given Keys

```
// Find the unique document that has key "key1".
{"$id" : "key1"}

// Find the documents that have any of the keys "key1", "key2", and "key3".
{"$id" : ["key1","key2","key3"]}

// Find the documents that have at least one of the keys "key1" and "key2",
// and that have an object with a field address.zip whose value is at least 94000.
{"$and" : [{$id : ["key1", "key2"]},
           {"address.zip" : { "$gte" : 94000 }}]}
```

1.3.15.8 Using QBE Operator $orderby

Query-by-example (QBE) operator `$orderby` is described.

It sorts query results in ascending or descending order.

The following QBE specifies the order of fields `age` and `salary`. A value of 1 specifies ascending order for `age`. A value of -2 specifies descending order for `salary`. Sorting is done first by `age` and then by `salary`, because the absolute value of 1 is less than the absolute value of -2.

```
{ "$query" : { "age" : { "$gt" :  40 } },
  "$orderby" :  { "age" : 1, "salary" : -2 } }
```

When you use operator `$orderby` in a filter specification together with one or more filter conditions, you must wrap those conditions with operator `$query`. In the preceding query, the returned documents are restricted to those that satisfy a filter condition that specifies that field `age` must have a value greater than 40.

See Also:

Orderby Clause Sorts Selected Objects (page 1-32)

1.3.15.9 Querying With a Filter Specification

You can query a collection for documents that match a particular filter specification (query-by-example, or QBE). You do this by passing a JSON `OracleDocument` that represents the QBE to method `OracleOperationBuilder filter()`.

Example 1-28 (page 1-29) illustrates this.

Example 1-28 Executing a Filter Specification

```
OracleDatabase db = ...

// OracleCollection - assume it is empty
OracleCollection col = ...

// Insert into the collection a document with field "name" set to "Jason"
// and field "location" set to "California".
OracleDocument doc =
   db.createDocumentFromString("{\"name\" : \"Jason\",
                               \"location\" : \"California\"}");
col.insert(doc);

// Insert another document into the collection with field "name" set to "Mary",
// and field "location" set to "California".
doc = db.createDocumentFromString("{\"name\" : \"Mary\",
                                  \"location\" : \"California\"}");
col.insert(doc);

// Create a filter specification for matching all documents with
// the field "name" set to "Jason"
OracleDocument filterSpec =
   db.createDocumentFromString("{\"name\" : \"Jason\"}");

// Run the filter specification
OracleCursor c = col.find().filter(filterSpec).getCursor();

// The cursor returns a single document with this content:
// { "name" : "Json", "location" : "California" } --
// the first document inserted above.

while (c.hasNext())
{
     OracleDocument c = c.next();
}

c.close();
```

1.4 SODA Paths

SODA specifications contain paths, each of which targets a value in a JSON document. A path is composed of a series of steps.

Note:

In paths, you must use strict JSON syntax. That is, you must enclose every nonnumeric value in double quotation marks ("). For information about strict and lax JSON syntax, see *Oracle Database JSON Developer's Guide*.

The characters used in path steps are of two kinds: syntactic and allowed. **Syntactic characters** have special syntactic meaning for JSON. They are the following:

- Period (`.`), which separates a parent-object field name from a child-object field name.
- Brackets (`[` and `]`), which are array delimiters.
- Comma (`,`), which separates array elements or index components.
- Wildcard (`*`), which is a placeholder. It matches any index in an array step and any field name in a field step.

Allowed characters are those that are not syntactic.

There are two kinds of steps in a path: field steps and array steps.

A **field step** is one of the following:

- The wildcard character `*` (by itself)
- A sequence of allowed characters — for example, `cat`
- A sequence of characters (allowed or syntactic) enclosed in backquote characters (`) — for example, `` `dog` `` and `` `cat*dog` ``

Within a field step that is enclosed in backquote characters, a syntactic character does not act syntactically; it is treated literally as an ordinary character. You must enclose any field step that contains a syntactic character in a pair of backquote characters, if you intend for the syntactic character to be treated literally.

Because all of the characters in `dog` are allowed, backquote characters are optional in `` `dog` ``. Because each of the following field steps contains a syntactic character, they must be enclosed in backquote characters:

```
`cat.dog`
`cat[dog]`
`cat,dog`
`cat*dog`
```

In `` `cat*dog` `` the asterisk does not act as a wildcard. Because it is escaped by backquotes, it acts as an ordinary character. But in the path `{ "*.b" : 42 }`, the unescaped asterisk acts as a wildcard; it is a placeholder for a field name. Similarly, the unescaped period also acts syntactically.

If a step that you enclose in backquote characters *contains* a backquote character, then you must represent that character using two consecutive backquote characters. For example: ``` `Customer``s Comment` ```.

A period (`.`) must be followed by a field step. After the first step in a path, each field step must be preceded by a period.

An **array step** is delimited by brackets (`[` and `]`). Inside the brackets can be either:

- The wildcard character `*` (by itself)
- One or more of these index components:
  - A single index, which is an integer greater than or equal to zero
  - An index range, which has this syntax:

    x **to** *y*

x and *y* are integers greater than or equal to zero, and *x* is less than or equal to *y*. There must be at least one whitespace character between *x* and `to` and between `to` and *y*.

Multiple components must be separated by commas (`,`). In a list of multiple components, indexes must be in ascending order, and ranges cannot overlap.

For example, these are valid array steps:

```
[*]
[1]
[1,2,3]
[1 to 3]
[1, 3 to 5]
```

The following are *not* valid array steps:

```
[*, 6]
[3, 2, 1]
[3 to 1]
[1 to 3, 2 to 4]
```

1.5 SODA Filter Specifications (QBEs)

You can select JSON documents in a collection by pattern-matching.

A **filter specification**, also known as a **query-by-example (QBE)** or simply a **filter**, is a SODA query that uses a pattern expressed in JSON. Some SODA operations use a filter specification to select all JSON documents in a collection that satisfy it, meaning that the filter specification evaluates to true for only those objects of the collection. A filter specification thus specifies characteristics that the documents that satisfy it must possess.

A filter specification pattern can use QBE **operators**, which are predefined JSON fields whose names start with **$**. The JSON value of an operator is called its **operand** or its argument.[1]

Although a SODA operator is itself a JSON field, for ease of exposition in the context of filter specification descriptions, the term *field* refers here to a JSON field that is *not* a SODA operator. (In the context of a QBE, "field" is often used informally to mean "path to a field".)

Note:

You must use *strict* JSON syntax in a filter specification. That is, you must enclose every nonnumeric value in double quotation marks. This includes QBE operators. For information about strict and lax JSON syntax, see *Oracle Database JSON Developer's Guide*.

A filter specification is a JSON object. There are three kinds of filter specification:

- Empty filter: **`{ }`**. An empty filter matches *all* objects in a collection.
- Composite filter.
- Filter-condition filter.

[1] A syntax error is raised if the argument to a QBE operator is not of the required type (for example, if `$gt` is passed an argument that is not a string or a number).

A filter specification (QBE) can appear only at the top (root) level of a query. However, a filter condition can be used either on its own, as a filter-condition filter (a QBE), or at a lower level, in the query clause of a composite filter.

Note:

QBE is not supported on a heterogeneous collection, that is, a collection that has the media type column. Such a collection is designed for storing JSON and non-JSON content.

See Also:

- Composite Filters (page 1-32)
- Filter Conditions (page 1-33)
- Media Type Column Name (page 1-55)

1.5.1 Composite Filters

A composite filter specification (query-by-example, or QBE) can appear only at the top level. That is, you cannot nest a composite filter inside another composite filter or inside a filter condition.

A composite filter consists of one or both of these clauses:

- Query clause

  It has the form `$query` *`filter_condition`*. See Filter Conditions (page 1-33).

- Orderby clause

  It has the form `$orderby` *`orderby_spec`*. See Orderby Clause Sorts Selected Objects (page 1-32).

Neither clause can appear more than once.

The following composite filter contains both clauses:

```
{ "$query" : { "salary" : { "gt" : 10000 } },
  "$orderby" : { "age" : -1, "zipcode" : 2 } } }
```

In this example, the query clause selects documents that have a salary field whose value is greater than 10,000, and the orderby clause sorts the selected documents first by descending age and then by ascending zip code.

1.5.1.1 Orderby Clause Sorts Selected Objects

A filter specification (query-by-example, or QBE) with an orderby clause returns the selected JSON documents in sorted order.

This is the syntax of an orderby clause:

```
"$orderby" : { field1 : direction1, field2 : direction2, ... }
```

The value of operator `$orderby` is a JSON object with one or more members.

Each *`field`* is a string that is interpreted as a path from the root of the candidate object.

Each *`direction`* is a non-zero integer. It sorts the returned documents by the *`field`* value in ascending or descending order, depending on whether the value is positive or negative, respectively.

The fields in the `$orderby` operand are sorted in the order of their magnitudes (absolute values), smaller magnitudes before larger ones. For example, a field with value -1 sorts before a field with value 2, which sorts before a field with value 3.

The following filter specification selects objects in which field `salary` has a value greater than 10,000 and less than or equal to 20,000. It sorts the objects first in descending order by `age` and then in ascending order by `zipcode`.

```
{ "$query"   : { "salary" : { "$gt" : 10000, "$lte" : 20000 } },
  "$orderby" : { "age" : -1, "zipcode" : 2 } }
```

The following SQL `SELECT` statement fragment is analogous:

```
WHERE (salary > 10000) AND (salary <= 20000)
ORDER BY age DESC, zipcode ASC
```

If the absolute values of two or more sort directions are *equal* then the order in which the fields are sorted is determined by the order in which they appear in the serialized JSON content that you use to create the JSON document.

Oracle recommends that you use sort directions that have *unequal* absolute values, to precisely govern the order in which the fields are used, especially if you use an external tool or library to create the JSON content and you are unsure of the order in which the resulting content is serialized.

See Also:

SODA Paths (page 1-29), for information about path strings

1.5.2 Filter Conditions

A filter condition can be used either on its own, as a filter specification, or at a lower level, in the query clause of a composite filter specification.

A **filter condition**, sometimes called just a **condition**, consists of one or more of these clauses:

- Basic Field Clause (page 1-33)
- Logical Clause (page 1-37)
- Nested-Condition Clause (page 1-38)
- Special-Criterion Clause (page 1-39)

A filter condition is true if and only if all of its clauses are true. A filter condition cannot be empty.

1.5.2.1 Basic Field Clause

A **basic field clause** specifies that a given field must satisfy a given set of criteria.

It can take the following forms:

- **Existence clause**: a field[2] followed by an **existence criterion**, which is a JSON object with operator `$exists` followed its operand (argument), a scalar. A JSON **scalar** is a value other than an object or an array; that is, it is a JSON number, string, `true`, `false`, or `null`.

An existence clause tests whether the field exists. It matches a document only if one of these is true:

– The field exists and the operand is any scalar value except `false`, `null`, or 0.

– The field does not exist and the operand is `false`, `null`, or 0.

For example, this existence clause tests whether there is a document with field `address.zip`:

```
"address.zip" : { "$exists" : true }
```

- **Scalar equality clause**: a field followed by a scalar value.

  A scalar equality clause tests whether the value of the field is equal to the scalar value. It is equivalent to a comparison clause for the same field that tests the same value using `$eq`.

  For example, this scalar equality clause tests whether the field `salary` has the value 10000:

  ```
  "salary" : 10000
  ```

  It is equivalent to the following comparison clause:

  ```
  "salary" : { "$eq" : 10000 }
  ```

- **Comparison clause**: a field followed by a JSON object containing one or more comparison criteria. A **comparison criterion** is a comparison operator followed by its operand. (The operators appear as JSON field names and the field values are the operands.)

  The **comparison operators** are `$eq`, `$ne`, `$gt`, `$lt`, `$gte`, `$lte`, `$startsWith`, `$regex`, `$in`, `$nin`, and `$all`.

  A comparison clause tests whether the value of the field satisfies *all* of the comparison criteria.

  For example, this comparison clause has two criteria. The first tests whether field `age` is greater than 18; the second tests whether it is less than or equal to 45:

  ```
  "age" : { "$gt" : 18, "$lte" : 45 }
  ```

Table 1-3 (page 1-34) describes the comparison operators. See Sample JSON Documents (page 1-23) for the documents used in column Example.

Table 1-3 Query-By-Example (QBE) Comparison Operators

Operator	Description	Operand	Example
`$eq`	Matches document only if field value equals argument value.	JSON scalar.	`{"name" : { "$eq" : "Jason" }}` matches sample document 1.

[2] A *field* here is any JSON field that is not an operator. And as always, operators and fields must be enclosed in double quotation marks (`"`) when used in SODA.

Table 1-3 (Cont.) Query-By-Example (QBE) Comparison Operators

Operator	Description	Operand	Example
`$ne`	Matches document only if field value does not equal argument value or there is no such field in the document.	JSON scalar.	`{"name" : { "$ne" : "Jason" }}` matches sample documents 2 and 3.
`$gt`	Matches document only if field value is greater than argument value.	JSON number or string.	`{"age" : { "$gt" : 45 }}` matches sample document 2.
`$lt`	Matches document only if field value is less than argument value.	JSON number or string.	`{"age" : { "$lt" : 50 }}` matches sample document 1.
`$gte`	Matches document only if field value is greater than or equal to argument value.	JSON number or string.	`{"age" : { "$gte" : 45 }}` matches sample documents 1, 2, and 3.
`$lte`	Matches document only if field value is less than or equal to argument value.	JSON number or string.	`{"age" : { "$lte" : 45 }}` matches sample document 1.
`$startsWith`	Matches document only if field value starts with argument value.	JSON string.	`{"name" : {"$startsWith" : "J"}}` matches sample document 1.
`$regex`	Matches document only if field value matches argument regular expression.	SQL regular expression, as a JSON string. See *Oracle Database SQL Language Reference*.	`{"name" : { "$regex" : ".*son"}}` matches sample document 1.
`$in`	Matches document only if field exists and its value equals at least one value in the argument array.	Non-empty JSON array of scalars.	`{"address.zip" : { "$in" : [ 94088, 90001 ] }}` matches sample documents 1 and 2.

Table 1-3 (Cont.) Query-By-Example (QBE) Comparison Operators

Operator	Description	Operand	Example
`$nin`	Matches document only if one of these is true: • Field exists, but its value is not equal to any value in the argument array. • Field does not exist.	Non-empty JSON array of scalars.[1]	`{"address.zip" : { "$nin" : [ 90001 ] }}` matches sample documents 1 and 2.

`$all`	Matches document only if one of these is true:
	• Field value is an array that contains all values in the argument array.
	• Field value is a scalar value and the argument array contains a single matching value.
		Non-empty JSON array of scalars.[1]	`{"drinks" : { "$all" : ["soda", "tea"]}}`
			matches sample document 2.
			`{"drinks": { "$all" : ["tea"]}}`
			matches sample documents 1 and 2.

[1] A syntax error is raised if the array does not contain at least one element.

Note:

When a path that does not end in an array step uses a comparison operator or `$not` applied to a comparison clause, and the path targets an array, the test applies to *each* element of the array.

For example, the QBE `{"animal" : {"$eq" : "cat"}}` matches the JSON data `{"animal" : ["dog", "cat"]}`, even though `"cat"` is an array element. The QBE `{"animal" : {$not : {"$eq" : "frog"}}}` matches the same data, because each of the array elements is tested for equality with `"frog"` and this test fails. (See "Logical Clause (page 1-37)" for information about operator `$not`.)

See Also:

- Basic Field Clause (page 1-33)
- Nested-Condition Clause (page 1-38)
- Composite Filters (page 1-32) for information about a query clause
- Sample JSON Documents (page 1-23)

1.5.2.2 Logical Clause

A **logical clause** is either a `$not` clause or logical combining clause.

- A **$not clause** is a field followed by a JSON object that has operator `$not` followed by its operand. The operand for `$not` is a single existence or comparison criterion.

  A `$not` clause logically negates the value of the `$not` operand. When the operand criterion is true, the `$not` clause evaluates to false; when the criterion is false, `$not` evaluates to true.

  For example, the following `$not` clause matches documents that have no field `address.zip`, as well as documents that have such a field but whose value is a scalar other than `"90001"` or an array that has no elements equal to `"90001"`:

  ```
  "address.zip" : {"$not" : { "$eq" : "90001" }}
  ```

  In contrast, the following comparison clause has the complementary effect: it matches documents that have a field `address.zip` whose value is either the scalar `"90001"` or an array that contains that scalar value.

  ```
  "address.zip" : { "$eq" : "90001"}}
  ```

- A **logical combining clause** is a **logical operator** —`$and`, `$or`, or `$nor`—followed by a non-empty array of one or more filter conditions.[3]

  This logical combining clause uses operator `$or`.

  ```
  "$or" [ { "name" : "Joe" }, { "salary" : 10000 } ]
  ```

  The following logical combining clause uses operator `$and`. Its array operand has two filter conditions as its members. The second of these is a condition with a logical combining clause that uses operator `$or`.

  ```
  "$and" : [ {"age" : {"$gte" : 60}},
             {"$or" : [{"name" : "Jason"}, {"drinks" : "tea"}]} ]
  ```

1.5.2.2.1 Omitting $and

Sometimes you can omit the use of `$and`.

A filter condition is true if and only if *all* of its clauses are true. And a comparison clause can contain multiple comparison criteria, *all* of which must be true for the comparison as whole to be true. In each of these, logical conjunction (AND) is implied. Because of this you can often omit the use of `$and`, for brevity.

This is illustrated by Example 1-29 (page 1-38) and Example 1-30 (page 1-38), which are equivalent in their effect. Operator `$and` is explicit in Example 1-29 (page 1-38) and implicit (omitted) in Example 1-30 (page 1-38).

The filter specifies objects for which the `name` starts with "Fred" *and* the salary is greater than 10,000 and less than or equal to 20,000 *and* either `address.city` is "Bedrock" or `address.zipcode` is 12345 *and* `married` is `true`.

A rule of thumb for `$and` omission is this: If you omit `$and`, make sure that no field or operator in the resulting filter appears multiple times at the same level in the same object.

This rule precludes using a QBE such as this, where field `salary` appears twice at the same level in the same object:

[3] A syntax error is raised if the array does not contain at least one element.

```
{ "salary" : { "$gt" : 10000 }, "age" : { "$gt" : 40 },
"salary" : { "$lt" : 20000 } }
```

And it precludes using a QBE such as this, where the same comparison operator, `$regex`, is applied more than once to field `name` in the same comparison:

```
{ "name" : { "$regex" : "son", "$regex" : "Jas" } }
```

The behavior here is *not* that the field condition is true if and only if both of the `$regex` criteria are true. To be sure to get that effect, you would use a QBE such as this one:

```
{ $and : [ { "name" : { "$regex" : "son" }, { "name" : { "$regex" : "Jas" } ] }
```

If you do not follow the rule of thumb for `$and` omission then only one of the conflicting conditions or criteria that use the same field or operator is evaluated; the others are ignored, and no error is raised. For the `salary` example, only one of the `salary` comparison clauses is evaluated; for the `name` example, only one of the `$regex` criteria is evaluated. Which one of the set of multiple conditions or criteria gets evaluated is undefined.

Example 1-29 Filter Specification with Explicit $and Operator

```
{ "$and" : [ { "name"    : { "$startsWith" : "Fred" } },
             { "salary"  : { "$gt"  : 10000, "$lte" : 20000 } },
             { "$or"     : [ { "address.city"    : "Bedrock" },
                             { "address.zipcode" : 12345 } ] },
             { "married" : true } ] }
```

Example 1-30 Filter Specification with Implicit $and Operator

```
{ "name"    : { "$startsWith" : "Fred" },
  "salary"  : { "$gt"  : 10000, "$lte" : 20000 },
  "$or"     : [ { "address.city"    : "Bedrock" },
                { "address.zipcode" : 12345 } ],
  "married" : true }
```

1.5.2.3 Nested-Condition Clause

A nested-condition clause consists of a parent field followed by a single condition. All fields contained in the condition are scoped to the parent field.

`parent_field` : *`condition`*

Note:

Since the condition of a nested-condition clause follows a field, it cannot contain a special-criterion clause. The latter can occur only at root level.

For example, suppose that field `address` has child fields `city` and `state`. The following nested-condition clause tests whether field `address.city` has the value `"Boston"` and field `address.state` has the value `"MA"`:

```
"address" : { "city" : "Boston", "state" : "MA" }
```

Similarly, this nested-condition clause tests whether the value of `address.city` starts with `Bos` and `address.state` has the value `"MA"`:

```
"address" : { "city" : { "$startsWith : "Bos" }, "state" : "MA" }
```

Suppose that you have this document:

```
{ "address" : [ { "city" : "Boston", "state" : "MA" },
                { "city" : "Los Angeles", "state" : "CA" } ] }
```

The following query matches each path in the document *independently*. Each object element of an `address` array is matched independently to see if it has a city value of `"Boston"` or a state value of `"CA"`.

```
{ "address.city" : "Boston", "address.state" : "CA" }
```

This query without a nested condition thus matches the preceding document, which has no *single object with both* city `"Boston"` and state `"CA"`.

The following query, with a nested-condition clause for parent field `address`, does *not* match the preceding document, because that document has no single object in an `address` array with both a field `city` of value `"Boston"` and a field `state` of value `"CA"`.

```
{ "address" : { "city" : "Boston", "state" : "CA" } }
```

See Also:

Special-Criterion Clause (page 1-39)

1.5.2.4 Special-Criterion Clause

A special-criterion clause is used only in a root-level condition, that is, a condition used in a composite filter or in a filter-condition filter.

Currently the only special-criterion clause is the ID clause.

1.5.2.4.1 ID Clause

Other query-by-example (QBE) operators generally look for particular JSON fields within documents and try to match their values. An ID clause, which uses operator `$id`, instead matches document keys.

A document key uniquely identifies a given document. It is metadata, like the creation timestamp, last-modified timestamp, and version. It pertains to the document as a whole and is not part of the document content.

The syntax of an ID clause is QBE operator `$id` followed by either a scalar key (document identifier) or a non-empty array of scalar keys.[4] The scalar key must be either an integer or a string. The array elements must be either all integers or all strings. For example:

```
"$id" : "USA"
"$id" : [1001,1002,1003]
```

You can use operator `$id` only in the outermost condition of a QBE. More precisely, if a QBE also uses other operators in addition to `$id`, then the outermost condition must have operator `$and`, and the sole occurrence of a `$id` condition must be an element of the array argument to that `$and` occurrence.

Example 1-31 (page 1-40) illustrates this. It finds documents that have at least one of the keys `key1` and `key2` and that have a `color` field with value `"red"`.

[4] A syntax error is raised if the array does not contain at least one element.

Example 1-31 Use of Operator $id in the Outermost QBE Condition

```
{ "$and" : [ { $id : [ "key1", "key2" ] }, { "color" : "red" } ] }
```

1.6 SODA Collection Metadata Caching

SODA collection metadata is stored persistently in the database, just like collection data. It is fetched transparently when needed, to perform collection operations. Fetching metadata from the database carries a performance cost. You can cache collection metadata in clients, to improve performance by avoiding database access to retrieve the metadata.

These are the main use cases for collection metadata caching:

- Listing a collection, then opening one or more of the collections listed.
- Creating a collection, then opening it.
- Reopening a collection.

In all of these cases, cached metadata can be used to open the collection.

A collection metadata cache can be *shared* by all of the `OracleDatabase` objects that are obtained from a given `OracleRDBMSClient` object, or it can be *local* to a single `OracleDatabase` object. Both kinds of caching are disabled by default.

If both local and shared caches are enabled for the same `OracleDatabase` object, entry lookup proceeds as follows:

1. The local cache is checked for an entry pertaining to a given collection used by the database object.
2. If not found in the local cache, the shared cache is checked for an entry for the collection.
3. If an entry for the collection is found in neither cache then the database is accessed to try to obtain the its metadata.

1.6.1 Enabling Collection Metadata Caching

Collection metadata caching is disabled by default. You can use constructor `OracleRDBMSClient(Properties *props*)` to enable shared or local collection metadata caching.

Parameter `*props*` here is a `Properties` instance that you initialize with one or both of the following properties:

- Property `oracle.soda.sharedMetadataCache` with value `"true"`: enable the shared cache
- Property `oracle.soda.localMetadataCache` with value `"true"`: enable the local cache

Example 1-32 (page 1-40) illustrates this; it enables both shared and local caching.

Example 1-32 Enabling Collection Metadata Caching

```
Properties props = new Properties();
props.put("oracle.soda.sharedMetadataCache", "true");
props.put("oracle.soda.localMetadataCache", "true");
OracleRDBMSClient cl = new OracleRDBMSClient(props);
```

1.6.2 Shared Collection Metadata Cache

Each SODA client (`OracleRDBMSClient` object) is optionally associated with a collection metadata cache that records metadata for all collections (`OracleCollection` objects) that are created for all `OracleDatabase` objects created from that client. The cache is released when its associated client is released.

The number of entries in a shared cache is limited to 10,000 entries (100 database schemas times 100 collections per schema). A shared cache uses a least-recently-used (LRU) replacement policy: the least recently used entry is replaced by the addition of a new entry, when the cache is full (it has 10,000 entries).

A shared metadata cache requires *locking* to avoid access conflict, which can affect performance negatively because it limits concurrency.

1.6.3 Local Collection Metadata Cache

Each `OracleDatabase` object is optionally associated with a local collection metadata cache. It records metadata only for collections that are created for that `OracleDatabase` object. A local cache is released when its associated `OracleDatabase` object is released.

There is no limit on the number of entries for a local cache — entries are never evicted. The number of entries continues to grow as new collections are created for the given database object.

The lack of an eviction policy for local metadata caches means that cached collection metadata is always available; once cached, the database need never be accessed to obtain it.

With local caching, because there is no sharing, using different database objects to access the same collection can result in more round trips and more data replication than is the case for shared caching.

Unlike a shared metadata cache, a local cache requires no locking.

Caution:

Because the number of entries in the local cache is unbounded, Oracle does not recommend using the local cache if a particular Oracle Database object is used to create a large number of collections, as it could result in running out of memory.

1.7 SODA Collection Configuration Using Custom Metadata

SODA collections are highly configurable. You can use custom metadata, which differs from the metadata that is provided by default.

However, Oracle recommends against using custom metadata without a compelling reason. Doing so requires familiarity with Oracle Database concepts, such as SQL data types (described in *Oracle Database SQL Language Reference*). SODA collections are implemented on top of Oracle Database tables (or views). Therefore, many collection configuration components are related to the underlying table configuration.

Reasons to use custom metadata include:

- To configure SecureFiles LOB storage.

- To configure a collection to store documents other than JSON (a **heterogeneous collection**).
- To map an existing Oracle RDBMS table or view to a new collection.
- To specify that a collection mapping to an existing table is read-only.
- To use a `VARCHAR2` column for JSON content, and to increase the default maximum length of data allowed in the column.

  You might want to increase the maximum allowed data length if your database is configured with extended data types, which extends the maximum length of these data types to 32767 bytes. For more information about extended data types, see *Oracle Database SQL Language Reference*.

Two methods for creating collections are available on interface `OracleDatabaseAdmin` (accessed by invoking method `admin()` on an `OracleDatabase` object):

```
createCollection(String collectionName);
createCollection(String collectionName, OracleDocument collectionMetadata);
```

The first method, which accepts only one argument, creates a collection with the default metadata. The default metadata specifies database schema name, table name (for the table storing the collection), five table columns (key, content, version, last-modified timestamp, and creation timestamp), and the details of these table columns. Each table column is represented by a field with a JSON object as value. That object contains additional details about the column—name, SQL type, and so on. (See Example 1-33 (page 1-42).)

The second method, which accepts two arguments, lets you provide custom collection metadata in the form of a JSON `OracleDocument` object.

1.7.1 Getting the Metadata of an Existing Collection

`OracleCollectionAdmin` method `getMetadata()` returns the JSON metadata document for a collection.

```
collectionName.admin().getMetadata();
```

Example 1-33 (page 1-42) shows the result of calling method `getContentAsString()` on the metadata document for a collection with the default configuration that was created using `OracleDatabaseAdmin` method `createCollection(String collectionName)`.

Example 1-33 getMetadata Output for Collection with Default Configuration

```
{
   "schemaName" : "mySchemaName",
   "tableName" : "myTableName",
   "keyColumn" :
   {
      "name" : "ID",
      "sqlType" : "VARCHAR2",
      "maxLength" : 255,
      "assignmentMethod" : "UUID"
   },
   "contentColumn" :
   {
      "name" : "JSON_DOCUMENT",
      "sqlType" : "BLOB",
      "compress" : "NONE",
      "cache" : true,
      "encrypt" : "NONE",
      "validation" : "STRICT"
   },
   "versionColumn" :
   {
     "name" : "VERSION",
     "type":"String",
     "method":"SHA256"
   },
   "lastModifiedColumn" :
   {
     "name":"LAST_MODIFIED"
   },
   "creationTimeColumn":
   {
      "name":"CREATED_ON"
   },
   "readOnly":false
}
```

1.7.2 Creating Custom Metadata for a Collection

Collection metadata is represented as a JSON `OracleDocument` instance. You can create such an instance directly, but Oracle recommends that you instead use `OracleRDBMSMetadataBuilder`, which you obtain by invoking `OracleRDBMSClient` method `createMetadataBuilder()`.

Method `createMetadataBuilder()` returns an `OracleRDBMSMetadataBuilder` instance that is preloaded with the default collection metadata. You can modify this preloaded metadata by calling `OracleRDBMSMetadataBuilder` methods that create custom metadata.

These methods correspond to different collection metadata components. You can customize these components by invoking builder methods in a chained manner. At the end of the chain, you invoke method `build()` to create collection metadata as a JSON `OracleDocument` object.

Example 1-34 (page 1-44) illustrates this; it uses `OracleRDBMSMetadataBuilder` to create a collection that has custom metadata: no creation time column, a media type column, and a non-default version column method. It first uses method `createMetadataBuilder()` to create a metadata builder object. It then invokes builder methods on that object to define the specific metadata to use, and it invokes `build()` to create a `collectionMetadata` object with that metadata. Finally, it creates a new collection that has this metadata.

In this case, the metadata that is specified, and the methods that define it, are as follows:

Method	Metadata
`creationTimeColumnName()`	There is to be no creation time column. By default, the column is present. A `null` value here specifies that it is absent.
`mediaTypeColumnName()`	The media type column is to be named `MY_MEDIA_TYPE_COLUMN`. By default, there is no media type column.

Method	Metadata
`versionColumnMethod()`	The version column method is to be `UUID`, instead of the default method, `SHA256`.

Example 1-34 Creating a Collection That Has Custom Metadata

```
OracleRDBMSClient cl = new OracleRDBMSClient();
OracleRDBMSMetadataBuilder b = cl.createMetadataBuilder();
OracleDatabase db = cl.getDatabase(jdbcConnection);

// Create custom metadata
OracleDocument collectionMetadata = b.creationTimeColumnName(null).
                                      mediaTypeColumnName("MY_MEDIA_TYPE_COLUMN").
                                      versionColumnMethod("UUID").
                                      build();

// Create a new collection with the specified custom metadata
db.admin().createCollection("collectionName", collectionMetadata);
```

1.7.3 Collection Metadata Components

Collection metadata is composed of multiple components.

- Schema (page 1-45)
- Table or View (page 1-45)
- Key Column Name (page 1-46)
- Key Column Type (page 1-46)
- Key Column Max Length (page 1-47)
- Key Column Assignment Method (page 1-47)
- Key Column Sequence Name (page 1-48)
- Content Column Name (page 1-49)
- Content Column Type (page 1-49)
- Content Column Max Length (page 1-49)
- Content Column JSON Validation (page 1-50)
- Content Column SecureFiles LOB Compression (page 1-51)
- Content Column SecureFiles LOB Cache (page 1-51)
- Content Column SecureFiles LOB Encryption (page 1-52)
- Version Column Name (page 1-52)
- Version Column Generation Method (page 1-52)
- Last-Modified Time Stamp Column Name (page 1-54)
- Last-Modified Column Index Name (page 1-54)

- Creation Time Stamp Column Name (page 1-54)
- Media Type Column Name (page 1-55)
- Read Only (page 1-55)

See Also:

`OracleRDBMSMetadataBuilder` methods Javadoc for more information about collection metadata components

Note:

The identifiers used for collection metadata components (schema name, table name, view name, database sequence name, and column names) must be valid Oracle quoted identifiers. Some characters and words that are allowed in Oracle quoted identifiers are strongly discouraged. For details, see *Oracle Database SQL Language Reference*).

1.7.3.1 Schema

The collection metadata component that specifies the name of the Oracle Database schema that owns the table or view to which the collection is mapped.

Property	Value
Default value	None
Allowed values	Valid Oracle quoted identifier[6]. If this value contains double quotation marks (") or control characters, SODA for Java replaces them with underscore characters (_).
`OracleRDBMSMetadataBuilder` method for selecting component	`schemaName()`
JSON collection metadata document path	`schemaName`

See Also:

Oracle Database SQL Language Reference for information about valid Oracle quoted identifiers

1.7.3.2 Table or View

The collection metadata component that specifies the name of the table or view to which the collection is mapped.

Property	Value
Default value	None

5 Reminder: letter case is significant for a quoted SQL identifier; it is interpreted case-sensitively.

Property	Value
Allowed values	Valid Oracle quoted identifier[6]. If this value contains double quotation marks (") or control characters, SODA for Java replaces them with underscore characters (_).
`OracleRDBMSMetadataBuilder` method for selecting component	`tableName()` or `viewName()`
JSON collection metadata document path	`tableName` or `viewName`

See Also:

Oracle Database SQL Language Reference for information about valid Oracle quoted identifiers

1.7.3.3 Key Column Name

The collection metadata component that specifies the name of the column that stores the document key.

Property	Value
Default value	`ID`
Allowed values	Valid Oracle quoted identifier[6] (as defined in *Oracle Database SQL Language Reference*). If this value contains double quotation marks (") or control characters, SODA for Java replaces them with underscore characters (_).
`OracleRDBMSMetadataBuilder` method for selecting component	`keyColumnName()`
JSON collection metadata document path	`keyColumn.name`

1.7.3.4 Key Column Type

The collection metadata component that specifies the SQL data type of the column that stores the document key.

Property	Value
Default value	`VARCHAR2`

Allowed values	`VARCHAR2`
	`NUMBER`
	`RAW(16)`

`OracleRDBMSMetadataBuilder` method for selecting component	`keyColumnType()`
JSON collection metadata document path	`keyColumn.sqlType`

Caution:

If client-assigned keys are used and the key column type is `VARCHAR2` then Oracle recommends that the database character set be AL32UTF8. This ensures that conversion of the keys to the database character set is lossless.

Otherwise, if client-assigned keys contain characters that are not supported in your database character set then conversion of the key into the database character set during a read or write operation is lossy. This can lead to duplicate-key errors during insert operations. More generally, it can lead to unpredictable results. For example, a read operation could return a value that is associated with a different key from the one you expect.

1.7.3.5 Key Column Max Length

The collection metadata component that specifies the maximum length of the key column in bytes. This component applies only to keys of type `VARCHAR2`.

Property	Value
Default value	`255`
Allowed values	At least 32 bytes if key assignment method is `UUID` or `GUID`. See Key Column Assignment Method (page 1-47).
`OracleRDBMSMetadataBuilder` method for selecting component	`keyColumnMaxLength()`
JSON collection metadata document path	`keyColumn.maxLength`

See Also:

Key Column Type (page 1-46)

1.7.3.6 Key Column Assignment Method

The collection metadata component that specifies the method used to assign keys to objects that are inserted into the collection.

Property	Value
Default value	`UUID`

Allowed values	`UUID`
	`GUID`
	`SEQUENCE`
	`CLIENT`
	For descriptions of these methods, see Table 1-4 (page 1-48).

`OracleRDBMSMetadataBuilder` method for selecting component	`keyColumnAssignmentMethod()`

Property	Value
JSON collection metadata document path	`keyColumn.assignmentMethod`

Table 1-4 Key Assignment Methods

Method	Description
`GUID`	Keys are generated in Oracle RDBMS by the SQL function `SYS_GUID`, described in *Oracle Database SQL Language Reference*.
`SEQUENCE`	Keys are generated in Oracle Database by a database sequence. If you specify the key assignment method as SEQUENCE then you must also specify the name of that sequence — see Key Column Sequence Name (page 1-48).
`CLIENT`	Keys are assigned by the client application.
`UUID` (default)	Keys are generated by SODA for Java, based on the `UUID` capability of the Java Virtual Machine (JVM) underlying SODA for Java.

1.7.3.7 Key Column Sequence Name

The collection metadata component that specifies the name of the database sequence that generates keys for documents that are inserted into a collection if the key assignment method is `SEQUENCE`.

If you specify the key assignment method as `SEQUENCE` then you must also specify the name of that sequence. If the specified sequence does not exist then SODA for Java creates it.

Property	Value
Default value	None
Allowed values	Valid Oracle quoted identifier[6] (as defined in *Oracle Database SQL Language Reference*). If this value contains double quotation marks (`"`) or control characters, SODA for Java replaces them with underscore characters (_).
OracleRDBMSMetadataBuilder method for selecting component	`keyColumnSequenceName()`
JSON collection metadata document path	`keyColumn.sequenceName`

Note:

If you drop a collection using SODA for Java, the sequence used for key generation is *not* dropped. This is because it might not have been created using SODA for Java. To drop the sequence, use SQL command `DROP SEQUENCE`, after first dropping the collection.

See Also:

- Key Column Assignment Method (page 1-47)
- *Oracle Database SQL Language Reference* for information about `DROP SEQUENCE`
- *Oracle Database Concepts* for information about database sequences

1.7.3.8 Content Column Name

The collection metadata component that specifies the name of the column that stores the database content.

Property	Value
Default value	`JSON_DOCUMENT`
Allowed values	Valid Oracle quoted identifier[6] (as defined in *Oracle Database SQL Language Reference*). If this value contains double quotation marks (") or control characters, SODA for Java replaces them with underscore characters (_).
`OracleRDBMSMetadataBuilder` method for selecting component	`contentColumnName()`
JSON collection metadata document path	`contentColumn.name`

1.7.3.9 Content Column Type

The collection metadata component that specifies the SQL data type of the column that stores the document content.

Property	Value
Default value	`BLOB`

Allowed values	`VARCHAR2`
	`BLOB`
	`CLOB`

`OracleRDBMSMetadataBuilder` method for selecting component	`contentColumnType()`
JSON collection metadata document path	`contentColumn.sqlType`

1.7.3.10 Content Column Max Length

The collection metadata component that specifies the maximum length of the content column in bytes. This component applies only to content of type `VARCHAR2`.

Property	Value
Default value	4000

Property	Value
Allowed values	32767 if extended data types are enabled. Otherwise, 4000 if content column type is `VARCHAR2`.
`OracleRDBMSMetadataBuilder` method for selecting component	`contentColumnMaxLength()`
JSON collection metadata document path	`contentColumn.maxLength`

See Also:

- Content Column Type (page 1-49)
- *Oracle Database SQL Language Reference* for information about extended data types

1.7.3.11 Content Column JSON Validation

The collection metadata component that specifies the syntax to which JSON content must conform—strict or lax.

Property	Value
Default value	`STANDARD`

Allowed values	`STANDARD`
	`STRICT`
	`LAX` (default for SQL condition `is json`)

`OracleRDBMSMetadataBuilder` method for selecting component	`contentColumnValidation()`
JSON collection metadata document path	`contentColumn.validation`

- `STANDARD` validates according to the JSON RFC 4627 standard. (It corresponds to the strict syntax defined for Oracle SQL condition `is json`.)
- `STRICT` is the same as `STANDARD`, except that it also verifies that the document does not contain duplicate JSON field names. (It corresponds to the strict syntax defined for Oracle SQL condition `is json` when the SQL keywords `WITH UNIQUE KEYS` are also used.)
- `LAX` validates more loosely. (It corresponds to the lax syntax defined for Oracle SQL condition `is json`.) Some of the relaxations that `LAX` allows include the following:
  - It does not require JSON field names to be enclosed in double quotation marks (`"`).
  - It allows uppercase, lowercase, and mixed case versions of `true`, `false`, and `null`.
  - Numerals can be represented in additional ways.

See Also:

- *Oracle Database JSON Developer's Guide* for information about strict and lax syntax JSON syntax
- `http://tools.ietf.org/html/rfc4627` for the JSON RFC 4627 standard

1.7.3.12 Content Column SecureFiles LOB Compression

The collection metadata component that specifies the SecureFiles LOB compression setting.

Property	Value
Default value	`NONE`

Allowed values	`NONE`
	`HIGH`
	`MEDIUM`
	`LOW`

`OracleRDBMSMetadataBuilder` method for selecting component	`contentColumnCompress()`
JSON collection metadata document path	`contentColumn.compress`

See Also:

Oracle Database SecureFiles and Large Objects Developer's Guide for information about SecureFiles LOB storage

1.7.3.13 Content Column SecureFiles LOB Cache

The collection metadata component that specifies the SecureFiles LOB cache setting.

Property	Value
Default value	`TRUE`
Allowed values	`TRUE` `FALSE`
`OracleRDBMSMetadataBuilder` method for selecting component	`contentColumnCache()`
JSON collection metadata document path	`contentColumn.cache`

See Also:

Oracle Database SecureFiles and Large Objects Developer's Guide for information about SecureFiles LOB storage

1.7.3.14 Content Column SecureFiles LOB Encryption

The collection metadata component that specifies the SecureFiles LOB encryption setting.

Property	Value
Default value	`NONE`

Allowed values	`NONE`
	`3DES168`
	`AES128`
	`AES192`
	`AES256`

`OracleRDBMSMetadataBuilder` method for selecting component	`contentColumnEncrypt()`
JSON collection metadata document path	`contentColumn.encrypt`

See Also:

Oracle Database SecureFiles and Large Objects Developer's Guide for information about SecureFiles LOB storage

1.7.3.15 Version Column Name

The collection metadata component that specifies the name of the column that stores the document version.

Property	Value
Default value	`VERSION`
Allowed values	Valid Oracle quoted identifier[6] (as defined in *Oracle Database SQL Language Reference*). If this value contains double quotation marks (") or control characters, SODA for Java replaces them with underscore characters (_).
`OracleRDBMSMetadataBuilder` method for selecting component	`versionColumnName()`
JSON collection metadata document path	`versionColumn.name`

1.7.3.16 Version Column Generation Method

The collection metadata component that specifies the method used to compute version values for objects when they are inserted into a collection or replaced.

Example 1-34 (page 1-44) uses this metadata component.

Property	Value
Default value	`SHA256`

Property	Value

Allowed values	`UUID`
	`TIMESTAMP`
	`MD5`
	`SHA256`
	`SEQUENTIAL`
	`NONE`

`OracleRDBMSMetadataBuilder` method for selecting component	`versionColumnMethod()`
JSON collection metadata document path	`versionColumn.method`

Table 1-5 (page 1-53) describes the version generation methods.

Table 1-5 Version Generation Methods

Method	Description
`UUID`	Ignoring object content, SODA for Java generates a universally unique identifier (UUID) when the document is inserted and for every replace operation. Efficient, but the version changes even if the original and replacement documents have identical content. Version column type value is `VARCHAR2(255)`.
`TIMESTAMP`	Ignoring object content, SODA for Java generates a value from the time stamp and coverts it to `LONG`. This method might require a round trip to the database instance to get the timestamp. As with `UUID`, the version changes even if the original and replacement documents have identical content. Version column type value is `NUMBER`.
`MD5`	SODA for Java uses the MD5 algorithm to compute a hash value of the document content. This method is less efficient than `UUID`, but the version changes only if the document content changes. Version column type value is `VARCHAR2(255)`.
`SHA256` (default)	SODA for Java uses the SHA256 algorithm to compute a hash value of the document content. This method is less efficient than `UUID`, but the version changes only if the document content changes. Version column type value is `VARCHAR2(255)`.
`SEQUENTIAL`	Ignoring object content, SODA for Java assigns version 1 when the object is inserted and increments the version value every time the object is replaced. Version values are easily understood by human users, but the version changes even if the original and replacement documents have identical content. Version column type value is `NUMBER`.
`NONE`	If the version column is present, `NONE` means that the version is generated outside SODA for Java (for example, by a database trigger).

See Also:

Table 1-5 (page 1-53) for descriptions of the allowed values

1.7.3.17 Last-Modified Time Stamp Column Name

The collection metadata component that specifies the name of the column that stores the last-modified time stamp of the document.

Property	Value
Default value	`LAST_MODIFIED`
Allowed values	Valid Oracle quoted identifier[6] (as defined in *Oracle Database SQL Language Reference*). If this value contains double quotation marks (") or control characters, SODA for Java replaces them with underscore characters (_).
`OracleRDBMSMetadataBuilder` method for selecting component	`lastModifiedColumnName()`
JSON collection metadata document path	`lastModifiedColumn.name`

1.7.3.18 Last-Modified Column Index Name

The collection metadata component that specifies the name of the index on the last-modified column.

Note:

This component is currently for internal use only. Do not change its value.

Property	Value
Default value	None
Allowed values	Valid Oracle quoted identifier[6] (as defined in *Oracle Database SQL Language Reference*). If this value contains double quotation marks (") or control characters, SODA for Java replaces them with underscore characters (_).
`OracleRDBMSMetadataBuilder` method for selecting component	`lastModifiedColumnIndex()`
JSON collection metadata document path	`lastModifiedColumn.index`

1.7.3.19 Creation Time Stamp Column Name

The collection metadata component that specifies the name of the column that stores the creation time stamp of the document. This timestamp is generated during the `insert`, `insertAndGet`, `save`, or `saveAndGet` operation.

Example 1-34 (page 1-44) uses this metadata component.

Property	Value
Default value	`CREATED_ON`
Allowed values	Valid Oracle quoted identifier[6] (as defined in *Oracle Database SQL Language Reference*). If this value contains double quotation marks (") or control characters, SODA for Java replaces them with underscore characters (_).
`OracleRDBMSMetadataBuilder` method for selecting component	`creationTimeColumnName()`
JSON collection metadata document path	`creationTimeColumn.name`

1.7.3.20 Media Type Column Name

The collection metadata component that specifies the name of the column that stores the media type of the document. A media type column is needed if the collection is to be heterogeneous, that is, it can store documents other than JSON.

Example 1-34 (page 1-44) uses this metadata component.

Note:

You cannot use query-by-example (QBE) with a heterogeneous collection (an error is raised if you try).

Property	Value
Default value	None
Allowed values	Valid Oracle quoted identifier[6] (as defined in *Oracle Database SQL Language Reference*). If this value contains double quotation marks (") or control characters then SODA for Java replaces them with underscore characters (_).
`OracleRDBMSMetadataBuilder` method for selecting component	`mediaTypeColumnName()`
JSON collection metadata document path	`mediaTypeColumn.name`

1.7.3.21 Read Only

The collection metadata component that specifies whether the collection is read-only.

Property	Value
Default value	`FALSE`
Allowed values	`TRUE` `FALSE`
`OracleRDBMSMetadataBuilder` method for selecting component	`readOnly()`

Property	Value
JSON collection metadata document path	`readOnly`

A

SODA for Java Core Interfaces

The SODA for Java core interfaces are described.

Table A-1 (page A-1) lists and briefly describes these interfaces. For complete information about them, see the SODA Javadoc.

Table A-1 SODA for Java Core Interfaces

Interface	Description
`OracleClient`	SODA for Java entry point (client)
`OracleDocument`	Document Content is typically JSON; possibly a MIME type (for example, image, audio, or video) Provides methods that get document content and metadata.
`OracleDatabase`	Database of collections of documents Provides methods that access `OracleDatabaseAdmin` and open existing collections. Inherits methods that create documents suitable for insertion into collections. Obtained by invoking `OracleClient.getDatabase()`.
`OracleDatabaseAdmin`	Provides methods that create collections and get their metadata. Obtained by invoking `OracleDatabase.admin()`.
`OracleCollection`	Collection of documents Provides methods that access `OracleOperationBuilder` and `OracleCollectionAdmin` and insert and save collection documents. Obtained by invoking `OracleDatabase.admin().createCollection()` or, if it already exists, `OracleDatabase.openCollection()`.
`OracleCollectionAdmin`	Provides methods that index and drop collections and get their metadata. Obtained by invoking `OracleDatabase.admin()`.

Table A-1 (Cont.) SODA for Java Core Interfaces

Interface	Description
`OracleOperationBuilder`	Builder and executor of read and write operations on a collection. Provides nonterminal methods for building operations (for example, `skip()` and `limit()`) and terminal methods for executing operations (for example, `getCursor()`, `count()`, and `remove()`). Obtained by invoking `OracleCollection.find()`, which returns an `OracleOperationBuilder` object that represents a query that finds all documents in the collection.
`OracleCursor`	Cursor for result list of query that `OracleCollection.find()` returns `next()` method returns the next document from the query result list. Obtained by invoking `OracleOperationBuilder.getCursor()`.

Index