Oracle® NoSQL Database

Java Direct Driver Developer's Guide

Release 26.1
E85378-38
July 2026

Oracle NoSQL Database Java Direct Driver Developer's Guide, Release 26.1

E85378-38

Copyright © 2011, 2026, Oracle and/or its affiliates.

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

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

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

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

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

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

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

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

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.

Oracle customer access to and use of Oracle support services will be pursuant to the terms and conditions specified in their Oracle order for the applicable services.

Contents

Preface

Conventions Used in This Book i
Diversity and Inclusion ii

1 Developing for Oracle NoSQL Database

Configuring Logging 2
Obtaining a KVStore Handle 2
- Using the KVStoreConfig Class 3

Using the Authentication APIs 3
- Prerequisites for using the Authentication APIs 4
  - Configuring SSL 4
  - Configuring Authentication 5
- Authentication using a LoginCredentials Instance 5
- Renewing Expired Login Credentials 7
- Authentication using Kerberos 9
- Authentication using Kerberos and JAAS 10
- Unauthorized Access 12

2 Introduction to Oracle KVLite

3 Introducing Oracle NoSQL Database Tables and Indexes

Defining Tables 1
- Supported Table Data Types 3
- Record Fields 4
- Defining Child Tables 5
- Defining Multi-Region Tables 5
  - Using CRDT datatype in a multi-region table 6
  - Add MR_COUNTER datatype in a multi-region table 7
  - Inserting JSON MR_COUNTER Values Programmatically 8
- Table Evolution 8

Using the UUID data type 9

Copyright © 2011, 2026, Oracle and/or its affiliates.

Inserting rows into a table with a UUID column 11
Updating rows of a table with a UUID column 12
Add or Remove a UUID column 12
Creating Indexes 13

4 Introducing Oracle NoSQL Database Namespaces

Creating Namespaces 1
Granting Authorization Access to Namespaces 2
Using and Setting Namespaces 5
Showing and Describing Namespaces 6
Dropping Namespaces 7

5 Primary and Shard Key Design

Primary Keys 1
Composite Keys 2
Data Type Limitations 2
Partial Primary Keys 3
Shard Keys 3
Row Data 4

6 Writing and Deleting Table Rows

Write Exceptions 1
Writing Rows to a Table in the Store 1
Writing Rows to a Child Table 3
Writing rows to an IDENTITY column 4
Other put Operations 8
Bulk Put Operations 9
Using Time to Live 12
Specifying a TTL Value 12
Updating a TTL Value 13
Deleting TTL Expiration 14
Setting Default Table TTL Values 14
Deleting Rows from the Store 15
Using multiDelete() 16

7 Reading Table Rows

Read Exceptions 1
Retrieving a Single Row 2

Copyright © 2011, 2026, Oracle and/or its affiliates.

Retrieve a Child Table 3
Using multiGet() 4
Iterating over Table Rows 6
Specifying Field Ranges 9
Iterating with Nested Tables 11
Reading Indexes 15
Parallel Scans 18
Bulk Get Operations 20

8 Using Data Types

Using Arrays 1
Using Binary 2
Using Enums 3
Using Fixed Binary 4
Using JSON 5
Using Maps 7
Using Embedded Records 8

9 Indexing Non-Scalar Data Types

Indexing Arrays 1
Indexing JSON Fields 2
Indexing Maps 4
Indexing by Map Keys 4
Indexing by Map Values 6
Indexing by a Specific Map Key Name 7
Indexing by Map Key and Value 9
Indexing Embedded Records 11

10 Using Row Versions

11 Consistency Guarantees

Specifying Consistency Policies 1
Using Simple Consistency 2
Using Time-Based Consistency 3
Using Version-Based Consistency 4

Copyright © 2011, 2026, Oracle and/or its affiliates.

12 Durability Guarantees

Setting Acknowledgment-Based Durability Policies 1
Setting Synchronization-Based Durability Policies 2
Setting Durability Guarantees 3

13 Executing a Sequence of Operations

Sequence Errors 1
Creating a Sequence 1
Executing a Sequence 4

14 Introduction to SQL for Oracle NoSQL Database

Running a simple query 1
Using binding variables 2
Accessing metadata 3
Using a query to update data 3

A JSON By Example

Sample Data A-2
UpdateJSON A-5
UpdateJSON.run() A-6
UpdateJSON.defineTable() A-7
UpdateJSON.createIndex() A-7
UpdateJSON.runDDL() A-8
UpdateJSON.updateTableWithoutQuery() A-8
UpdateJSON.updateTableWithIndex() A-10
UpdateJSON.updateTableUsingSQLQuery() A-11
UpdateJSON.updateZipCode() A-11
UpdateJSON.loadTable() A-12
UpdateJSON.displayTable() A-14
UpdateJSON.displayResult() A-15
UpdateJSON.parseArgs() A-15

B Table Data Definition Language Overview

Name Constraints B-1
DDL Comments B-2
CREATE TABLE B-2
    Field Definitions B-3
    Supported Data Types B-4

Copyright © 2011, 2026, Oracle and/or its affiliates.

Field Constraints B-5
Integer Serialized Constraints B-5
COMMENT B-6
DEFAULT B-6
IDENTITY B-6
UUID B-7
MR_COUNTER B-7
NOT NULL B-8
USING TTL B-8
Table Creation Examples B-9
Modify Table Definitions B-10
ALTER TABLE ADD field B-10
ALTER TABLE DROP Option B-11
ALTER TABLE USING TTL B-11
ALTER TABLE ADD REGIONS B-11
ALTER TABLE DROP REGIONS B-12
DROP TABLE B-12
CREATE INDEX B-13
Indexable Field Types B-13
Simple Indexes B-14
Multi-Key Indexes B-15
Multi-Key Index Restrictions B-15
JSON Indexes B-17
CREATE FULL TEXT INDEX B-18
DROP INDEX B-18
DESCRIBE AS JSON TABLE B-19
DESCRIBE AS JSON INDEX B-19
SHOW TABLES B-20
SHOW INDEXES B-20

C Exceptions

Copyright © 2011, 2026, Oracle and/or its affiliates.

Preface

There are two different APIs that can be used to write Oracle NoSQL Database applications: the original Key/Value API, and the Table API. In addition, the Key/Value API is available in Java and C. The Table API is available in Java, C, node.js (Javascript), Python, and C#. This document describes how to write Oracle NoSQL Database applications using the Table API in Java.

> **Note**
>
> Most application developers should use one of the Table drivers because the Table API offers important features not found in the Key/Value API. The Key/Value API will no longer be enhanced in future releases of Oracle NoSQL Database.

This document provides the concepts surrounding Oracle NoSQL Database, data schema considerations, as well as introductory programming examples.

This document is aimed at the software engineer responsible for writing an Oracle NoSQL Database application.

Conventions Used in This Book

The following typographical conventions are used within in this manual:

Class names are represented in `monospaced font`, as are `method names`. For example: "The `KVStoreConfig()` constructor returns a `KVStoreConfig` class object."

Variable or non-literal text is presented in *italics*. For example: "Go to your *KVHOME* directory."

Program examples are displayed in a `monospaced font` on a shaded background. For example:

```
import oracle.kv.KVStore;
import oracle.kv.KVStoreConfig;

...

KVStoreConfig kconfig = new KVStoreConfig("exampleStore",
    "node1.example.org:5088, node2.example.org:4129");
KVStore kvstore = null;
```

In some situations, programming examples are updated from one chapter to the next. When this occurs, the new code is presented in **`monospaced bold`** font. For example:

```
import oracle.kv.KVStore;
import oracle.kv.KVStoreConfig;
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
import oracle.kv.KVStoreFactory;

...

KVStoreConfig kconfig = new KVStoreConfig("exampleStore",
    "node1.example.org:5088, node2.example.org:4129");
KVStore kvstore = null;

try {
    kvstore = KVStoreFactory.getStore(kconfig);
} catch (FaultException fe) {
    // Some internal error occurred. Either abort your application
    // or retry the operation.
}
```

> **Note**
>
> Finally, notes of special interest are represented using a note block such as this.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Copyright © 2011, 2026, Oracle and/or its affiliates.

1

Developing for Oracle NoSQL Database

You access the data in the Oracle NoSQL Database KVStore using Java drivers that are provided for the product. In addition to the Java drivers, several other drivers are also available. They are:

1. Java Key/Value Driver
2. C Table Driver
3. C Key/Value Driver
4. Python Table Driver
5. node.js Table Driver
6. C# Table Driver

> **Note**
>
> New users should use one of the Table drivers unless they require a feature only available in the Key/Value API (such as Large Object support). The Key/Value API will no longer be enhanced in future releases of Oracle NoSQL Database.

The Java and C Key/Value driver provides access to store data using key/value pairs. All other drivers provide access using tables. Also, the Java Key/Value driver provides Large Object (LOB) support that as of this release does not appear in the other drivers. However, users of the Java Tables driver can access the LOB API, even though the LOB API is accessed using the Key/Value interface.

Finally, the Java driver provides access to SQL for Oracle NoSQL Database, so you can run queries. For more information see Introduction to SQL for Oracle NoSQL Database.

Users of the Table drivers are able to create and use secondary indexing. The Java and C Key/Value drivers do not provide this support.

To work, the C Table, Python Table, node.js Table, and C# Table drivers require use of a proxy server which translates network activity between the driver and the Oracle NoSQL Database store. The proxy is written in Java, and can run on any machine that is network accessible by both your client code and the Oracle NoSQL Database store. However, for performance and security reasons, Oracle recommends that you run the proxy on the same local host as your driver, and that the proxy be used in a 1:1 configuration with your drivers (that is, each instance of the proxy should be used with just a single driver instance).

Regardless of the driver you decide to use, the provided classes and methods allow you to write data to the store, retrieve it, and delete it. You use these APIs to define consistency and durability guarantees. It is also possible to execute a sequence of store operations atomically so that all the operations succeed, or none of them do.

The rest of this book introduces the Java APIs that you use to access the store, and the concepts that go along with them.

Copyright © 2011, 2026, Oracle and/or its affiliates.

Configuring Logging

The Oracle NoSQL Database Java drivers use standard Java logging to capture debugging output using loggers in the "oracle.kv" hierarchy. These loggers are configured to use a `oracle.kv.util.ConsoleHandler` class, and to ignore any handlers for loggers above `oracle.kv` in the logger hierarchy. As a result, logging will be performed to the console at whatever logging levels are configured for the various loggers and for the `oracle.kv.util.ConsoleHandler` class. You can adjust what console output appears for these loggers by modifying the logging levels for the loggers and the logging handler in their application's logging configuration file.

You can also configure additional logging handlers for all loggers used by the Java driver by specifying handlers for the `oracle.kv` logger.

For example, if you want to enable file output for Java driver logging at the INFO level or above, add the following to your application's configuration file (that is, the file you identify using the `java.util.logging.config.file` system property):

```
 # Set the logging level for the FileHandler logging handler to INFO
java.util.logging.FileHandler.level=INFO

# Set the logging level for all Java driver loggers to INFO
oracle.kv.level=INFO

# Specify that Java driver loggers should supply log output to the
# standard file handler
oracle.kv.handlers=java.util.logging.FileHandler
```

For information on managing logging in a Java application, see the `java.util.logging` Javadoc.

Obtaining a KVStore Handle

To acccess the store for any reason, you must first obtain a `KVStore` handle, using the `KVStoreFactory.getStore()` method.

When you get a `KVStore` handle, provide a `KVStoreConfig` object to the handle. The configuration object identifies important properties about the store that you are accessing. This section describes the `KVStoreConfig` class. Minimally, use this class to identify the following information:

- The store name. The name you provide must be identical to the name used when you installed the store.
- The network contact information for one or more helper hosts. Such contact information consists of the network name and port information for hosts currently belonging to the store. Identify multiple hosts using an array of strings, from one element to several. We recommend using multiple hosts, since any host can be down temporarily, and other hosts are then useful.

In addition to the `KVStoreConfig` class object, you can also provide a `PasswordCredentials` class object to `KVStoreFactory.getStore()`. Do this if you are using a store configured to require authentication, which is recommended. See Using the Authentication APIs for more information.

Copyright © 2011, 2026, Oracle and/or its affiliates.

For a store that does not require authentication, get a store handle like this:

```
package kvstore.basicExample;

import oracle.kv.KVStore;
import oracle.kv.KVStoreConfig;
import oracle.kv.KVStoreFactory;

...

String[] hhosts = {"n1.example.org:5088", "n2.example.org:4129"};
KVStoreConfig kconfig = new KVStoreConfig("exampleStore", hhosts);
KVStore kvstore = KVStoreFactory.getStore(kconfig);
```

Using the KVStoreConfig Class

Use the `KVStoreConfig` class to describe properties about a `KVStore` handle. Most of the properties are optional, and those that are required are provided when you construct a class instance.

The properties that you can provide using `KVStoreConfig` are as follows:

- Consistency

  Consistency is a property describing how likely it is that a record read from a replica node is identical to the same record stored on a master node. For more information, see Consistency Guarantees.

- Durability

  Durability is a property describing how likely it is that a write operation performed on the master node will not be lost if the master node is lost or is shut down abnormally. For more information, see Durability Guarantees.

- Helper Hosts

  Helper hosts are hostname and port pairs that identify how to contact helper nodes within the store. Use an array of strings to identify multiple helper hosts . Typically, you will obtain these hostname and port pairs from the store's deployer or administrator. For example:

  ```
  String[] hhosts = {"n1.example.org:3333", "n2.example.org:3333"};
  ```

- Request Timeout

  Configures the amount of time the `KVStore` handle will wait for an operation to complete before it times out.

- Store name

  Identifies the name of the store.

- Password credentials and optionally a reauthentication handler

  See the next section on authentication.

Using the Authentication APIs

You can install Oracle NoSQL Database so that your client code does not have to authenticate to the store. (For the sake of clarity, most of the examples in this book do not perform authentication.) However, if you want your store to operate securely, you can require

Copyright © 2011, 2026, Oracle and/or its affiliates.

authentication. Requiring authentication incurs a performance cost, due to the overhead of using SSL and authentication. While we recommend that your production store requires authentication over SSL, some sites that are particularly performance sensitive can forgo that level of security. To understand more about a secure store and the various methods to access it, see the Security Guide.

Authentication involves sending username/password credentials to the store at the time a store handle is acquired.

If you configure your store to support authentication, it is automatically configured to communicate with clients using SSL. The use of SSL ensures privacy of the authentication and other sensitive information. To use SSL, you must install SSL certificates on the machines where your client code runs, to validate that the store being accessed is trustworthy.

You can authenticate to the store in several different ways. You can use Kerberos, or you can specify a `LoginCredentials` implementation instance to `KVStoreFactory.getStore()`. (Oracle NoSQL Database provides the `PasswordCredentials` class as a `LoginCredentials` implementation.) If you use Kerberos, you can either use security properties that Oracle NoSQL Database understands to provide necessary Kerberos information, or you can use the Java Authentication and Authorization Service (JAAS) programming framework.

For information on using `LoginCredentials`, see Authentication using a LoginCredentials Instance. For information on using Kerberos, see Authentication using Kerberos. For information on using JAAS with Kerberos, see Authentication using Kerberos and JAAS.

Prerequisites for using the Authentication APIs

You must configure SSL and authentication before using the Authentication APIs.

Configuring SSL

If you are using a secure store, then all communications between your client code and the store is transported over SSL, including authentication credentials. You must therefore configure your client code to use SSL. To do this, you identify where the SSL certificate data is, and you also separately indicate that the SSL transport is to be used.

Identifying the Trust Store

When an Oracle NoSQL Database store is configured to use the SSL transport, a series of security files are generated using a security configuration tool. One of these files is the `client.trust` file, which must be copied to any machine running Oracle NoSQL Database client code.

For information on using the security configuration tool, see Security Configuration in the *Security Guide*.

Your code must be told where the `client.trust` file can be found because it contains the certificates necessary to establish an SSL connection with the store. You indicate where this file is physically located on your machine using the `oracle.kv.ssl.trustStore` property. There are two ways to set this property:

1. Identify the location of the trust store by using a `Properties` object to set the `oracle.kv.ssl.trustStore` property. You then use `KVStoreConfig.setSecurityProperties()` to pass the `Properties` object to your `KVStore` handle.

   When you use this method, you use `KVSecurityConstants.SSL_TRUSTSTORE_FILE_PROPERTY` as the property name.

Copyright © 2011, 2026, Oracle and/or its affiliates.

2. Use the `oracle.kv.security` property to refer to a properties file, such as the `client.trust` file. In that file, set the `oracle.kv.ssl.trustStore` property.

Setting the SSL Transport Property

In addition to identifying the location of the `client.trust` file, you must also tell your client code to use the SSL transport. You do this by setting the `oracle.kv.transport` property. There are two ways to set this property:

1. Identify the location of the trust store by using a `Properties` object to set the `oracle.kv.transport` property. You then use `KVStoreConfig.setSecurityProperties()` to pass the `Properties` object to your `KVStore` handle.

   When you use this method, you use `KVSecurityConstants.TRANSPORT_PROPERTY` as the property name, and `KVSecurityConstants.SSL_TRANSPORT_NAME` as the property value.

2. Use the `oracle.kv.security` property to refer to a properties file, such as the `client.trust` file. In that file, set the `oracle.kv.transport` property.

Configuring Authentication

You need a user to access the secure store. For details on creating and managing a user in the database, see Configuring Authentication.

> **Note**
>
> If you are using secure KVLite, you can make use of the `admin` user and the `user.security` file that is created by default when you start KVLite.

Authentication using a LoginCredentials Instance

You can authenticate to the store by specifying a `LoginCredentials` implementation instance to `KVStoreFactory.getStore()`. Oracle NoSQL Database provides the `PasswordCredentials` class as a `LoginCredentials` implementation. If your store requires SSL to be used as the transport, configure that prior to performing the authentication. (See the previous section for details.)

Your code should be prepared to handle a failed authentication attempt. `KVStoreFactory.getStore()` will throw `AuthenticationFailure` in the event of a failed authentication attempt. You can catch that exception and handle the problem there.

The following is a simple example of obtaining a store handle for a secured store. The SSL transport is used in this example.

```
import java.util.Properties;

import oracle.kv.AuthenticationFailure;
import oracle.kv.PasswordCredentials;
import oracle.kv.KVSecurityConstants;
import oracle.kv.KVStoreConfig;
import oracle.kv.KVStoreFactory;



KVStore store = null;
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
try {
    /*
     * storeName, hostName, port, username, and password are all
     * strings that would come from somewhere else in your
     * application.
     */
    KVStoreConfig kconfig =
        new KVStoreConfig(storeName, hostName + ":" + port);

    /* Set the required security properties */
    Properties secProps = new Properties();
    secProps.setProperty(KVSecurityConstants.TRANSPORT_PROPERTY,
                         KVSecurityConstants.SSL_TRANSPORT_NAME);
    secProps.setProperty
        (KVSecurityConstants.SSL_TRUSTSTORE_FILE_PROPERTY,
        "/home/kv/client.trust");
    kconfig.setSecurityProperties(secProps);

    store =
        KVStoreFactory.getStore(kconfig,
          new PasswordCredentials(username,
                                  password.toCharArray()),
                                  null /* ReauthenticateHandler */);
} catch (AuthenticationFailureException afe) {
    /*
     * Could potentially retry the login, possibly with different
     * credentials, but in this simple example, we just fail the
     * attempt.
     */
    System.out.println("authentication failed!");
    return;
    }
```

Another way to handle the login is to place your authentication credentials in a flat text file that contains all the necessary properties for authentication. In order for this to work, a password store must have been configured for your Oracle NoSQL Database store. See the *Security Guide* for information on setting up password stores.

For example, suppose your store has been configured to use a password file password store and it is contained in a file called `login.pwd`. In that case, you might create a login properties file called `login.txt` that looks like this:

```
oracle.kv.auth.username=clientUID1
oracle.kv.auth.pwdfile.file=/home/nosql/login.pwd
oracle.kv.transport=ssl
oracle.kv.ssl.trustStore=/home/nosql/client.trust
```

In this case, you can perform authentication in the following way:

```
import oracle.kv.AuthenticationFailure;
import oracle.kv.PasswordCredentials;
import oracle.kv.KVStoreConfig;
import oracle.kv.KVStoreFactory;

/* the client gets login credentials from the login.txt file */
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
/* can be set on command line as well */
System.setProperty("oracle.kv.security", "/home/nosql/login.txt");

KVStore store = null;
try {
    /*
     * storeName, hostName, port are all strings that would come
     * from somewhere else in your application.
     *
     * Notice that we do not pass in any login credentials.
     * All of that information comes from login.txt
     */
    myStoreHandle =
        KVStoreFactory.getStore(
            new KVStoreConfig(storeName, hostName + ":" + port))
} catch (AuthenticationFailureException afe) {
    /*
     * Could potentially retry the login, possibly with different
     * credentials, but in this simple example, we just fail the
     * attempt.
     */
    System.out.println("authentication failed!")
    return;
}
```

Renewing Expired Login Credentials

It is possible for an authentication session to expire. This can happen for several reasons. One is that the store's administrator has configured the store to not allow session extension and the session has timed out. These properties are configured using `sessionExtendAllow` and `sessionTimeout`.

Reauthentication might also be required if some kind of a major disruption has occurred to the store which caused the authentication session to become invalidated. This is a pathological condition which you should not see with any kind of frequency in a production store. Stores which are installed in labs might exhibit this condition more, especially if the stores are frequently restarted.

An application can encounter an expired authentication session at any point in its lifetime, so robust code that must remain running should always be written to respond to authentication session expirations.

When an authentication session expires, by default the method which is attempting store access will throw `AuthenticationRequiredException`. Upon seeing this, your code needs to reauthenticate to the store, and then retry the failed operation.

You can manually reauthenticate to the store by using the `KVStore.login()` method. This method requires you to provide the login credentials via a `LoginCredentials` class instance (such as `PasswordCredentials`):

```
try {
    ...
    /* Store access code happens here */
    ...
} catch (AuthenticationRequiredException are) {
    /*
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
     * myStoreHandle is a KVStore class instance.
     *
     * pwCreds is a PasswordCredentials class instance, obtained
     * from somewhere else in your code.
     */
    myStoreHandle.login(pwCreds);
}
```

Note that this is not required if you use the `oracle.kv.auth.username` and `oracle.kv.auth.pwdfile.file` properties, as shown in the previous section. In that case, your Oracle NoSQL Database client code will automatically and silently reauthenticate your client using the values specified by those properties.

A third option is to create a `ReauthenticationHandler` class implementation that performs your reauthentication for you. This option is only necessary if you provided a `LoginCredentials` implementation instance (that is, `PasswordCredentials`) in a call to `KVStoreFactory.getStore()`, and you want to avoid a subsequent need to retry operations by catching `AuthenticationRequiredException`.

A truly robust example of a `ReauthenticationHandler` implementation is beyond the scope of this manual (it would be driven by highly unique requirements that are unlikely to be appropriate for your site). Still, in the interest of completeness, the following shows a very simple and not very elegant implementation of `ReauthenticationHandler`:

```
package kvstore.basicExample

import oracle.kv.ReauthenticationHandler;
import oracle.kv.PasswordCredentials;

public class MyReauthHandler implements ReauthenticationHandler {
    public void reauthenticate(KVStore reauthStore) {
        /*
         * The code to obtain the username and password strings would
         * go here. This should be consistent with the code to perform
         * simple authentication for your client.
         */
        PasswordCredentials cred = new PasswordCredentials(username,
            password.toCharArray());

        reauthStore.login(cred);
    }
}
```

You would then supply a `MyReauthHandler` instance when you obtain your store handle:

```
import java.util.Properties;

import oracle.kv.AuthenticationFailure;
import oracle.kv.PasswordCredentials;
import oracle.kv.KVSecurityConstants;
import oracle.kv.KVStoreConfig;
import oracle.kv.KVStoreFactory;

import kvstore.basicExample.MyReauthHandler;
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
KVStore store = null;
try {
    /*
     * storeName, hostName, port, username, and password are all
     * strings that would come from somewhere else in your
     * application. The code you use to obtain your username
     * and password should be consistent with the code used to
     * obtain that information in MyReauthHandler.
     */
    KVStoreConfig kconfig =
        new KVStoreConfig(storeName, hostName + ":" + port);

    /* Set the required security properties */
    Properties secProps = new Properties();
    secProps.setProperty(KVSecurityConstants.TRANSPORT_PROPERTY,
                         KVSecurityConstants.SSL_TRANSPORT_NAME);
    secProps.setProperty
        (KVSecurityConstants.SSL_TRUSTSTORE_FILE_PROPERTY,
        "/home/kv/client.trust");
    kconfig.setSecurityProperties(secProps);

    store =
        KVStoreFactory.getStore(kconfig,
            new PasswordCredentials(username,
                                    password.toCharArray()));
            new MyReauthHandler());
} catch (AuthenticationFailureException afe) {
    /*
     * Could potentially retry the login, possibly with different
     * credentials, but in this simple example, we just fail the
     * attempt.
     */
    System.out.println("authentication failed!")
    return;
}
```

Authentication using Kerberos

You can authenticate to the store by using Kerberos. To do this, you must already have installed Kerberos and obtained the necessary login and service information.

The following is a simple example of obtaining a store handle for a secured store, and using Kerberos to authenticate. Information specific to Kerberos, such as the Kerberos user name, is specified using `KVSecurityConstants` that are set as properties to the `KVStoreConfig` instance which is used to create the store handle.

```
import java.util.Properties;

import oracle.kv.KVSecurityConstants;
import oracle.kv.KVStore;
import oracle.kv.KVStoreConfig;
import oracle.kv.KVStoreFactory;

KVStore store = null;
/*
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
 * storeName, hostName, port, username, and password are all
 * strings that would come from somewhere else in your
 * application.
 */
KVStoreConfig kconfig =
    new KVStoreConfig(storeName, hostName + ":" + port);

/* Set the required security properties */
Properties secProps = new Properties();

/* Set the user name */
secProps.setProperty(KVSecurityConstants.AUTH_USERNAME_PROPERTY,
                     "krbuser");

/* Use Kerberos */
secProps.setProperty(KVSecurityConstants.AUTH_EXT_MECH_PROPERTY,
                      "kerberos");

/* Set SSL for the wire level encryption */
secProps.setProperty(KVSecurityConstants.TRANSPORT_PROPERTY,
                      KVSecurityConstants.SSL_TRANSPORT_NAME);

/* Set the location of the public trust file for SSL */
secProps.setProperty
    (KVSecurityConstants.SSL_TRUSTSTORE_FILE_PROPERTY,
    "/home/kv/client.trust");

/* Set the service principal associated with the helper host */
final String servicesDesc =
        "localhost:oraclenosql/localhost@EXAMPLE.COM";
secProps.setProperty(
        KVSecurityConstants.AUTH_KRB_SERVICES_PROPERTY,
        servicesDesc);

/*
 * Set the default realm name to permit using a short name for the
 * user principal
 */
secProps.setProperty(KVSecurityConstants.AUTH_KRB_REALM_PROPERTY,
                      "EXAMPLE.COM");

/* Specify the client keytab file location */
secProps.setProperty(KVSecurityConstants.AUTH_KRB_KEYTAB_PROPERTY,
                      "/tmp/krbuser.keytab");

kconfig.setSecurityProperties(secProps);

store = KVStoreFactory.getStore(kconfig);
```

Authentication using Kerberos and JAAS

You can authenticate to the store by using Kerberos and the Java Authentication and Authorization Service (JAAS) login API. To do this, you must already have installed Kerberos and obtained the necessary login and service information.

Copyright © 2011, 2026, Oracle and/or its affiliates.

The following is a simple example of obtaining a store handle for a secured store, and using Kerberos with JAAS to authenticate.

To use JAAS, you create a configuration file that contains required Kerberos configuration information. For example, the following could be placed in the file named `jaas.config`:

```
oraclenosql {
  com.sun.security.auth.module.Krb5LoginModule required
  principal="krbuser"
  useKeyTab="true"
  keyTab="/tmp/krbuser.keytab";
};
```

To identify this file to your application, set the Java property `java.security.auth.login.config` using the `-D` option when you run your application.

Beyond that, you use `KVSecurityConstants` to specify necessary properties, such as the SSL transport. You can also specify necessary Kerberos properties, such as the Kerberos user name, using `KVSecurityConstants`, or you can use the `KerberosCredentials` class to do this.

```
import java.security.PrivilegedActionException;
import java.security.PrivilegedExceptionAction;
import java.util.Properties;

import javax.security.auth.Subject;
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;

import oracle.kv.KerberosCredentials;
import oracle.kv.KVSecurityConstants;
import oracle.kv.KVStore;
import oracle.kv.KVStoreConfig;
import oracle.kv.KVStoreFactory;

/*
 * storeName, hostName, port, username, and password are all
 * strings that would come from somewhere else in your
 * application.
 */
final KVStoreConfig kconfig =
    new KVStoreConfig(storeName, hostName + ":" + port);

/* Set the required security properties */
Properties secProps = new Properties();

/* Set SSL for the wire level encryption */
secProps.setProperty(KVSecurityConstants.TRANSPORT_PROPERTY,
                     KVSecurityConstants.SSL_TRANSPORT_NAME);

/* Set the location of the public trust file for SSL */
secProps.setProperty
    (KVSecurityConstants.SSL_TRUSTSTORE_FILE_PROPERTY,
    "/home/kv/client.trust");

/* Use Kerberos */
secProps.setProperty(KVSecurityConstants.AUTH_EXT_MECH_PROPERTY,
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
                    "kerberos");

/* Set Kerberos properties */
final Properties krbProperties = new Properties();

/* Set the service principal associated with the helper host */
final String servicesPpal =
    "localhost:oraclenosql/localhost@EXAMPLE.COM";
krbProperties.setProperty(KVSecurityConstants.AUTH_KRB_SERVICES_PROPERTY,
                          hostName + ":" + servicesPpal);

/* Set default realm name, because the short name
 * for the user principal is used.
 */
krbProperties.setProperty(KVSecurityConstants.AUTH_KRB_REALM_PROPERTY,
                          "EXAMPLE.COM");

/* Specify Kerberos principal */
final KerberosCredentials krbCreds =
    new KerberosCredentials("krbuser", krbProperties);

try {
    /* Get a login context */
    final Subject subj = new Subject();
    final LoginContext lc = new LoginContext("oraclenosql", subj);

    /* Attempt to log in */
    lc.login();

    /* Get the store using the credentials specified in the subject */
    kconfig.setSecurityProperties(secProps);

    store = Subject.doAs(
        subj, new PrivilegedExceptionAction<KVStore>() {
            @Override
            public KVStore run() throws Exception {
                return KVStoreFactory.getStore(kconfig, krbCreds, null);
            }
        });
} catch (LoginException le) {
    // LoginException handling goes here
} catch (PrivilegedActionException pae) {
    // PrivilegedActionException handling goes here
} catch (Exception e) {
    // General Exception handling goes here
}
```

Unauthorized Access

Clients which must authenticate to a store are granted some amount of access to the store. This could range from a limited set of privileges to full, complete access. The amount of access is defined by the roles and privileges granted to the authenticating user. Therefore, a call to the Oracle NoSQL Database API could fail due to not having the authorization to perform the operation. When this happens, `UnauthorizedException` will be thrown.

Copyright © 2011, 2026, Oracle and/or its affiliates.

When `UnauthorizedException` is seen, the operation should not be retried. Instead, the operation should either be abandoned entirely, or your code could attempt to reauthenticate using different credentials that would have the required permissions necessary to perform the operation. Note that a client can log out of a store using `KVStore.logout()`. How your code logs back in is determined by how your store is configured for access, as described in the previous sections.

```
// Open a store handle, and perform authentication as you do
// as described earlier in this section.

...


try {
    // When you attempt some operation (such as a put or delete)
    // to a secure store, you should catch UnauthorizedException
    // in case the user credentials you are using do not have the
    // privileges necessary to perform the operation.
} catch (UnauthorizedException ue) {
    /*
     * When you see this, either abandon the operation entirely,
     * or log out and log back in with credentials that might
     * have the proper permissions for the operation.
     */
    System.out.println("authorization failed!")
    return;
}
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

2

Introduction to Oracle KVLite

KVLite is a single-node, single shard store. It usually runs in a single process and is used to develop and test client applications. It is a simplified version of the database that comes bundled with the Oracle NoSQL Database package. To know more on how to start KVLite, see Quick Start to KVLite.

Copyright © 2011, 2026, Oracle and/or its affiliates.

3

Introducing Oracle NoSQL Database Tables and Indexes

Using the Table API (in one of the supported languages) is the recommended method of developing an Oracle NoSQL Database client application. Table APIs let you manipulate data using a tables metaphor, in which data is organized in multiple columns of data. The table APIs support an unlimited number of subtables. You can also create indexes to improve query performance against your tables.

If you have a mix of clients accessing your store using both Table and Key/Value APIs, a remote chance exists that keys from different clients could collide. As a best practice, it is recommended that you do not use your data store to store both raw key/value data and table data. If this is required for your situation, then it is recommended that you store your key/value data in a table with 2 columns: a key column and a data column. Doing so will allow you to use the Table APIs on both your table and key/value data. To avoid any possible conflict between keys, however unlikely, make sure that every KV key has either only:

- A single component
- A single major component

> **Note**
>
> Throughout this manual, examples call `TableAPI.getTable()`. The cost of calling this API is relatively high, because doing so requires a round trip to the store to fulfill the request. For optimal performance, call this method sparingly in your code.

Defining Tables

Before an Oracle NoSQL Database client can read or write to a table in the store, you must first create the tables. There are several ways to do this, but this document focuses on using Table DDL statements. You can submit these statements to the store directly using the SQL CLI. However, the recommended approach is to submit DDL statements to the store programmatically. This section describes both direct and programmatic methods.

The DDL language that you use to define tables is described in Table Data Definition Language Overview. This section provides a brief overview of how to use that language.

As an introductory example, suppose you want to create a table called `myTable` with four columns: `item`, `description`, `count`, and `percentage`. To create your table, use the following statement from the SQL CLI:

```
sql-> CREATE TABLE myTable (
    item STRING,
    description STRING,
    count INTEGER,
    percentage DOUBLE,
    PRIMARY KEY (item) // Every table must have a primary key
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
);
Statement completed successfully
```

> **Note**
>
> Primary keys are a concept that have not yet been introduced. See Primary and Shard Key Design for a complete explanation of what they are, and how you should use them. For now, be sure a primary key exists for every table you create, just as the previous example illustrates.

Executing DDL Statements Programmatically

To add the table definition to the store programmatically use the `KVStore.execute()` or `KVStore.executeSync()` methods. (The latter method executes the statement synchronously.)

For example:

```
package kvstore.basicExample;

import oracle.kv.FaultException;
import oracle.kv.StatementResult;
import oracle.kv.KVStore;
import oracle.kv.table.TableAPI;
...
// store handle creation and open omitted
...

StatementResult result = null;
String statement = null;

public void createTable() {
    StatementResult result = null;
    String statement = null;

    try {
        /*
         * Add a table to the database.
         * Execute this statement asynchronously.
         */
        statement =
            "CREATE TABLE myTable (" +
            "item STRING," +
            "description STRING," +
            "count INTEGER," +
            "percentage DOUBLE," +
            "PRIMARY KEY (item))"; // Required"
        result = store.executeSync(statement);

        displayResult(result, statement);

    } catch (IllegalArgumentException e) {
        System.out.println("Invalid statement:\n" + e.getMessage());
    } catch (FaultException e) {
        System.out.println
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
            ("Statement couldn't be executed, please retry: " + e);
    }
}

private void displayResult(StatementResult result, String statement) {
    System.out.println("============================");
    if (result.isSuccessful()) {
        System.out.println("Statement was successful:\n\t" +
            statement);
        System.out.println("Results:\n\t" + result.getInfo());
    } else if (result.isCancelled()) {
        System.out.println("Statement was cancelled:\n\t" +
            statement);
    } else {
        /*
         * statement was not successful: may be in error, or may still
         * be in progress.
         */
        if (result.isDone()) {
            System.out.println("Statement failed:\n\t" + statement);
            System.out.println("Problem:\n\t" +
                result.getErrorMessage());
        } else {
            System.out.println("Statement in progress:\n\t" +
                statement);
            System.out.println("Status:\n\t" + result.getInfo());
        }
    }
}
```

Supported Table Data Types

You specify schema for each column in an Oracle NoSQL Database table. This schema can be a primitive data type, or complex data types that are handled as objects.

Oracle NoSQL Database tables support the following data types:

Data Type	Description
Array	An array of values, all of the same type.
Binary	Implemented as a byte array with no predetermined fixed size.
Boolean	
Double	
Enum	An enumeration, represented as an array of strings.
Fixed Binary	Implemented as a byte array with no predetermined fixed size.
Float	
Integer	
Json	Any valid JSON data.
Long	
Number	A numeric type capable of handling any type of number or any value or precision.
Map	An unordered map type, where all entries are constrained by a single type.

Copyright © 2011, 2026, Oracle and/or its affiliates.

Data Type	Description
Records	See the following section.
String	
Timestamp	An absolute timestamp encapsulating a date and, optionally, a time value.

Record Fields

As described in Defining Child Tables, you can create child tables to hold subordinate information, such as addresses in a contacts database, or vendor contact information for an inventory system. When you do this, you can create an unlimited number of rows in the child table, and you can index the fields in the child table's rows.

However, you do not need to create child tables to organize subordinate data. If you have simple requirements for subordinate data, you can use record fields, instead of child tables. In general, you can use record fields instead of child tables if you want only a fixed, small number of instances of the record for each parent table row. For anything beyond trivial cases, use child tables.

> **Note**
>
> There is no downside to using child tables for even trivial cases.

The assumption when using record fields is that you have a fixed, known number of records to manage (unless you organize them as arrays). For example, in a contacts database, child tables let you have an unlimited number of addresses associated for each user. By using records, rather than child tables, you can associate a fixed number of addresses by creating a record field for each supported address (home and work, for example).

For example:

```
CREATE TABLE myContactsTable (
    uid STRING,
    surname STRING,
    familiarName STRING,
    homePhone STRING,
    workPhone STRING,
    homeAddress RECORD (street STRING, city STRING, state STRING,
                zip INTEGER),
    workAddress RECORD (street STRING, city STRING, state STRING,
                zip INTEGER),
    PRIMARY KEY(uid))
```

Alternatively, you can create an array of record fields. This lets you create an unlimited number of address records per field. In general, however, you should use child tables in this case.

```
CREATE TABLE myContactsTable (
    uid STRING,
    surname STRING,
    familiarName STRING,
    homePhone STRING,
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
    workPhone STRING,
    addresses ARRAY(RECORD (street STRING, city STRING, state STRING,
                zip INTEGER))),
    PRIMARY KEY(uid))
```

Defining Child Tables

Oracle NoSQL Database tables can be organized in a parent/child hierarchy. There is no limit to how many child tables you can create, nor is there a limit to how deep the child table nesting can go.

By default, child tables are not retrieved when you retrieve a parent table, nor is the parent retrieved when you retrieve a child table.

To create a child table, you name the table using the format: *<parentTableName>.<childTableName>*. For example, suppose you had the trivial table called `myInventory`:

```
CREATE TABLE myInventory (
  itemCategory STRING,
  description STRING,
  PRIMARY KEY (itemCategory)
)
```

We can create a child table called `itemDetails` in the following way:

```
CREATE TABLE myInventory.itemDetails (
    itemSKU STRING,
    itemDescription STRING,
    price FLOAT,
    inventoryCount INTEGER,
    PRIMARY KEY (itemSKU)
)
```

Note that when you do this, the child table inherits the parent table's primary key. In this trivial case, the child table's primary key is actually two fields: `itemCategory` and `itemSKU`. This has several ramifications, one of which is that the parent's primary key fields are retrieved when you retrieve the child table. See Retrieve a Child Table for more information.

Defining Multi-Region Tables

A Multi-Region Table or MR Table is a global logical table that is stored and maintained in different regions or installations. It is a *read-anywhere* and *write-anywhere* table that lives in multiple regions.

Consider an Oracle NoSQL Database with three regions, Frankfurt, London, and Dublin. To create a table called `users` that stores user details for all the three regions, you must create an MR table on each KVStore in the connected graph, and specify the list of regions that the table should span.

Copyright © 2011, 2026, Oracle and/or its affiliates.

For example, to create the `users` table in all the three regions, you must execute the following command from each region separately:

```
CREATE TABLE users (
  id INTEGER,
  firstName STRING,
  lastName STRING,
  age INTEGER,
  primary key (id)
) IN REGIONS fra, lnd, dub;
```

For information about MR Tables, see Life Cycle of MR Tables in the *Concepts Guide*.

Using CRDT datatype in a multi-region table

Overview of the MR_COUNTER data type

MR_Counter data type is a counter CRDT. CRDT stands for Conflict-free Replicated Data Type. In a multi-region setup of an Oracle NoSQL Database, a CRDT is a data type that can be replicated across servers where regions can be updated independently and it converges on a correct common state. Changes in the regions are concurrent and not synchronized with one another. In short, CRDTs provide a way for concurrent modifications to be merged across regions without user intervention. Oracle NoSQL Database currently supports the counter CRDT which is called MR_Counter. The MR_COUNTER datatype is a subtype of the INTEGER or LONG or NUMBER data type. You can also use the MR_COUNTER data type in a schema-less JSON field, which means one or more fields in a JSON document can be of MR_COUNTER data type.

Why do you need MR_Counter in a multi-region table?

In a multi-region database configuration, copies of the same data need to be stored in multiple regions. This configuration needs to deal with the fact that the data may be concurrently modified in different regions.

Take an example of a multi-region table in three different regions (where data is stored in three different Oracle NoSQL Database stores). Concurrent updates of the same data in multiple regions, without coordination between the machines hosting the regions, can result in inconsistencies between the regions, which in the general case may not be resolvable. Restoring consistency and data integrity when there are conflicts between updates may require some or all of the updates to be entirely or partially dropped. For example, in the current configuration of a multi-region table in the Oracle NoSQL Database, if the same column (a counter) of a multi-region table is updated across two regions at the same time with different values, a conflict arises.

Currently, the conflict resolution is that the latest write overwrites the value across regions. For example, Region 1 updates column1 with a value R1, and region2 updates column1 with a value R2, and if the region2 update happens after region1, the value of the column (counter) in both the regions becomes R2. This is not what is actually desired. Rather every region should update the column (a counter) at their end and also the system internally needs to determine the sum of the column across regions.

One way to handle this conflict is making serializable/linearizable transactions (one transaction is completed and changes are synchronized in all regions and only then the next transaction happens). A significant problem of having serializable transactions is performance. This is where MR_COUNTER datatype comes in handy. With MR_COUNTER datatype, we don't need serializable transactions and the conflict resolution is taken care of. That is, MR_COUNTER datatype ensures that though data modifications can happen simultaneously

Copyright © 2011, 2026, Oracle and/or its affiliates.

on different regions, the data can always be merged into a consistent state. This merge is performed automatically by MR_COUNTER datatype, without requiring any special conflict resolution code or user intervention.

Use-case for MR_COUNTER datatype

Consider a Telecom provider providing different services and packages to its customers. One such service is a "Family Plan" option where a customer and their family share the Data Usage plan. The customer is allocated a free data usage limit for a month which your the customer's entire family collectively uses. When the total usage of customer's family reaches 90 percent of the data limit, the telecom provider sends the customer an alert. Say there are four members in customer's family plan who are spread across different physical regions. The customer needs to get an alert from the telecom provider once the total consumption of their family reaches 90 percent of the free usage. The data is replicated in different regions to cater to latency, throughput, and better performance. That means there are four regions and each has a kvstore containing the details of the customer's data usage. The usage of their family members needs to be updated in different regions and at any point in time, the total usage should be monitored and an alert should be sent if the data usage reaches the limit.

An MR_COUNTER data type is ideal in such a situation to do conflict-free tracking of the data usage across different regions. In the above example, an increment counter in every data region's data store will track the data usage in that region. The consolidated data usage for all regions can be determined by the system at any point without any user intervention. That is the total data usage at any point in time can be easily determined by the system using an MR_COUNTER datatype.

Types of MR_COUNTER Datatype

Currently, Oracle NoSQL Database supports only one type of MR_COUNTER data type. which is Positive-Negative (PN) counter.

Positive-Negative (PN) Counter

A PN counter can be incremented or decremented. Therefore, these can serve as a general-purpose counter. For example, you can use these counters to count the number of users active on a social media website at any point. When the users go offline you need to decrement the counter.

To create a multi-region table with an MR_COUNTER column, See Create multi-region table with an MR_COUNTER column section in the Administrator's Guide.

Add MR_COUNTER datatype in a multi-region table

For example, to create a multi-region table `myTable` with a counter data type, you must execute the following command from each region.

```
CREATE Table myTable( name STRING,
                      count INTEGER AS MR_COUNTER,
                      PRIMARY KEY(name)) IN REGIONS DEN,LON;
```

Example using JSON MR_COUNTER data type:

Create a JSON MR_COUNTER data type in a multi-region table as shown below:

```
CREATE TABLE demoJSONMR(name STRING,
jsonWithCounter JSON(counter as INTEGER MR_COUNTER,
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
                        person.count as LONG MR_COUNTER),
PRIMARY KEY(name)) IN REGIONS FRA,LND;
```

In the statement above, you create a multi-region table with a STRING column and a column(JSON documents). You are identifying two of the fields in the JSON document as MR_COUNTER data type.. The first field is `counter`, which is an INTEGER MR_COUNTER data type. The second field is `count` within an embedded JSON document (`person`). The `count` field is of LONG MR_COUNTER data type.

Inserting JSON MR_COUNTER Values Programmatically

You can insert row (and MR_COUNTER values) into your application table programmatically using PUT API.

Create a sample table:

```
CREATE TABLE exampleTable (id INTEGER,
jsonColumn JSON(counter AS INTEGER MR_COUNTER,
                person.score AS LONG MR_COUNTER),
PRIMARY KEY(id));
```

Insert row of data using PUT API:

```
KVStore store = KVStoreFactory.getStore(kvStoreConfig);
TableAPI api = store.getTableAPI();
Table table = api.getTable("exampleTable");
Row row = table.createRow();
String json = "{\"counter\": 0, \"person\": {\"score\" : 0}}";
row.put("id", 1);
row.putJson("jsonColumn", json);
api.put(row, null, null);
```

> **Note**
>
> You need to include all defined json mr_counters when inserting a row. For example, in the above code, giving an empty JSON MR_COUNTER like `INSERT INTO exampleTable VALUES (1, {})` will throw an error. The values of the JSON MR_COUNTER need not be 0 but it can be any atomic values.

Updating JSON MR_COUNTER values: You cannot update a JSON MR_COUNTER programatically using API. It can be done using UPDATE statement (DML) only.

Table Evolution

As your application is used over time, it's often necessary to update your tables to either add new fields or remove existing fields that are no longer required. Table evolution is the term used to update table definitions, adding or removing fields, or changing field properties, such as a default value. You may even add a particular kind of column, like an IDENTITY column, to increment some value automatically. Only tables that already exist in the store are candidates for table evolution.

Use the `ALTER TABLE` statement to perform table evolution. See Modify Table Definitions.

Copyright © 2011, 2026, Oracle and/or its affiliates.

> **Note**
>
> You cannot remove a field if it is a primary key field, or if it participates in an index. You also cannot add primary key fields during table evolution.

For example, the following statements evolve the table that was created in the previous section. In this example, you would submit each statement to the store consecutively, using either the API or the CLI.

```
ALTER TABLE myInventory.itemDetails (ADD salePrice FLOAT)


ALTER TABLE myInventory.itemDetails (DROP inventoryCount)
```

Using the UUID data type

Overview of the UUID data type

A universally unique identifier (UUID) is a 128-bit number used to identify information in computer systems. You can create a UUID and use it to uniquely identify something. In its canonical textual representation, the 16 octets of a UUID are represented as 32 hexadecimal (base-16) digits, displayed in five groups separated by hyphens, in the form 8-4-4-4-12 for a total of 36 characters (32 hexadecimal characters and 4 hyphens). For example, `a81bc81b-dead-4e5d-abff-90865d1e13b1`.

In Oracle NoSQL, UUID values are represented by the UUID data type. The UUID data type is considered a subtype of the STRING data type, because UUID values are displayed in their canonical textual format and, in general, behave the same as string values in the various SQL operators and expressions. However, in order to save disk space, the UUID value is saved in a compact format on disk. If the UUID value is the primary key, the canonical 36-byte string is converted to a 19-byte string, then is saved on disk. If the UUID value is a non-primary key, the canonical 36-byte string is converted to a 16-byte array, then is saved on disk.

A table column can be declared as having UUID type in a CREATE TABLE statement. The UUID data type is best-suited in situations where you need a globally unique identifier for the records in a table that span multiple regions since identity columns are only guaranteed to be unique within a NoSQL cluster in a region.

Using the UUID data type:

Declare a column with UUID data type. UUID is a subtype of the STRING data type. This UUID column can be defined as GENERATED BY DEFAULT. The system then automatically generates a value for the UUID column if you do not supply a value for it.

Syntax:

```
uuid_definition := AS UUID [GENERATED BY DEFAULT]
```

Semantics

Declares the type of a column to be the UUID type. If the GENERATED BY DEFAULT keywords are used, the system generates a value for the UUID column automatically, if the user does not supply one.

Copyright © 2011, 2026, Oracle and/or its affiliates.

UUID Column Characteristics :

- One table can have multiple columns defined as "STRING AS UUID". However, one table can have only one column defined as "STRING AS UUID GENERATED BY DEFAULT".
- Since the Identity column is also generated by the system, the Identity column and the UUID GENERATED BY DEFAULT columns are mutually exclusive. That means only one IDENTITY column or one "UUID GENERATED BY DEFAULT" can exist per table.
- You create a UUID column as part of a CREATE TABLE DDL statement or add a UUID column to an existing table with an ALTER TABLE DDL statement.
- You can also index UUID columns via secondary indexes.

Example 1: UUID Column without GENERATED BY DEFAULT

```
CREATE TABLE myTable (id STRING AS UUID,name STRING, PRIMARY KEY (id));
Statement completed successfully
```

In the above example, the id column has no "GENERATED BY DEFAULT" defined, therefore, whenever you insert a new row, you need to explicitly specify a value for the id column.

```
INSERT INTO myTable
        values("a81bc81b-dead-4e5d-abff-90865d1e13b1", "test1");
Statement completed successfully
```

Input format: The input string must conform to the format specified by RFC 4122. An IllegalArgumentException is thrown if the input string does not conform to the string representation as described at Class UUID.

Output format: The output is a UUID canonical format. This is 32 hexadecimal(base-16) digits, displayed in five groups separated by hyphens, in the form 8-4-4-4-12 for a total of 36 characters (32 hexadecimal characters and 4 hyphens).

The value for a UUID column can also be generated using the `random_uuid` function, which returns a randomly generated UUID, as a string of 36 characters. See Function to generate a UUID string.

Example 2: UUID Column using GENERATED BY DEFAULT

```
CREATE TABLE myTable (id STRING AS UUID GENERATED BY DEFAULT, name STRING,
PRIMARY KEY (id));
Statement completed successfully
```

In the above example, the id column has "GENERATED BY DEFAULT" defined, therefore, whenever you insert a new row without specifying the value for the id column, Oracle NoSQL Database automatically generates a value for it.

```
INSERT INTO myTable VALUES(default,"test1");
Statement completed successfully
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

Table 3-1 Comparison between Identity Column and UUID column

Identity Column	UUID column
Declare a column as Identity to have Oracle NoSQL Cluster automatically assign values to it	Declare a column as UUID if you need unique values to be assigned to a NoSQL Cluster column in a multi-region system
An INTEGER, LONG, or NUMBER column in a table can be defined as an Identity column	A UUID is a subtype of the STRING data type
An Identity column can be defined either as GENERATED ALWAYS or GENERATED BY DEFAULT	A UUID column can be defined as GENERATED BY DEFAULT or you can supply the value of the string while inserting or updating data
Ideal in a single cluster architecture	The UUID data type is best suited In situations where you need a globally unique identifier for the records in a table that span multiple regions since identity columns are only guaranteed to be unique within a NoSQL cluster in a region.
Costs less storage space than a corresponding UUID column.	Costs more storage space than a corresponding Identity column.
If LONG is the primary key, it costs a maximum of 10 bytes. If LONG is a non-primary key, it costs a maximum of 8 bytes.	If the UUID value is the primary key, it costs 19-bytes. If the UUID value is a non-primary key, it costs 16-bytes.

Inserting rows into a table with a UUID column

The system generates a UUID column value when the keyword `DEFAULT` is used as the insert_clause for the UUID column.

Here are a few examples that show INSERT statements for both flavors of the UUID column – GENERATED BY DEFAULT and when no DEFAULT CLAUSE is specified in a CREATE TABLE statement. The keyword `DEFAULT` in the INSERT statement applies only when the UUID column is declared as GENERATED BY DEFAULT.

Example : Inserting rows into a table with a UUID column without GENERATED BY DEFAULT clause

```
CREATE TABLE myTable (id STRING AS UUID, name STRING, PRIMARY KEY (id));
Statement completed successfully

INSERT INTO myTable values("a81bc81b-dead-4e5d-abff-90865d1e13b1","test1");
Statement completed successfully
```

In the above example, the id column in the table `myTable` has no "GENERATED BY DEFAULT" defined. Therefore, whenever you insert a new row, you need to explicitly specify the value for the id column.

Example : Inserting rows into a table with a UUID column using the `random_uuid` function

The value for a UUID column can also be generated using the `random_uuid` function. See Function to generate a UUID string.

```
sql-> INSERT INTO myTable values(random_uuid(),"test2");
{"NumRowsInserted":1}
1 row returned
Statement completed successfully
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
sql-> select * from myTable;
{"id":"d576ab3b-8a36-4dff-b50c-9d9d4ca6072c","name":"test2"}
{"id":"a81bc81b-dead-4e5d-abff-90865d1e13b1","name":"test1"}
2 rows returned
Statement completed successfully
```

In this example, a randomly generated UUID is fetched using the `random_uuid` function. This value is used in the INSERT statement.

Example : Inserting rows into a table with a UUID column with GENERATED BY DEFAULT clause

```
CREATE TABLE myTable (id STRING AS UUID GENERATED BY DEFAULT,name STRING,
PRIMARY KEY (id));
Statement completed successfully

INSERT INTO myTable VALUES(default,"test1") returning id;
{"id":"e7fbab63-7730-4ec9-be73-a62e33ea73c3"}
Statement completed successfully
```

In the above example, the id column in `myTable` has "GENERATED BY DEFAULT" defined. The system generates a UUID column value when the keyword `DEFAULT` is used in the insert_clause for the UUID column. The system generated UUID value is fetched using the `returning` clause.

Updating rows of a table with a UUID column

You can update a UUID column whether or not it is GENERATED BY DEFAULT. You can use the function `random_uuid` to generate a random UUID value to update the column. The function `random_uuid` returns a randomly generated UUID, as a string of 36 characters.

Example : Updating a UUID Column defined without GENERATED BY DEFAULT clause

```
CREATE TABLE myTable (tabId INTEGER, id STRING AS UUID, PRIMARY KEY (tabId));
Statement completed successfully

INSERT INTO myTable values(1,"a81bc81b-dead-4e5d-abff-90865d1e13b1");
Statement completed successfully

UPDATE myTable set id=random_uuid() where tabId=1;
Statement completed successfully
```

The above example shows how you can update a UUID column which is NOT GENERATED BY DEFAULT. To do so, the UUID column should not be part of the primary key, as NoSQL Primary key values are immutable. In the above example, `tabId` is the Primary key. So you can update the UUID column using the `random_uuid` function.

Add or Remove a UUID column

An existing table can be altered and a new UUID column can be added. The existing records in the table will have a NULL value for the newly added UUID column. An existing UUID column can also be removed from a table.

Copyright © 2011, 2026, Oracle and/or its affiliates.

Adding a UUID Column to an Existing Table

Use ALTER TABLE to add a UUID column to an existing table.

Create a table `test_alter` without a UUID column.

```
sql-> CREATE TABLE test_alter(id INTEGER,
                 name STRING, PRIMARY KEY(id));
Statement completed successfully
```

Use ALTER TABLE to add a UUID column to `test_alter`. You can specify the default clause, GENERATED BY DEFAULT.

```
sql-> ALTER TABLE test_alter
           (ADD new_id STRING AS UUID GENERATED BY DEFAULT );
Statement completed successfully
```

Dropping a UUID Column

To remove a UUID column from a table, use ALTER TABLE with a DROP id clause.

> **Note**
>
> You cannot drop a UUID column if it is the primary key, or if it participates in an index.

```
sql-> CREATE Table Test_alter ( name STRING ,
                 id STRING AS UUID GENERATED BY DEFAULT,
                 PRIMARY KEY (name));
Statement completed successfully

sql-> ALTER TABLE Test_alter (DROP id);
Statement completed successfully
```

Creating Indexes

Indexes represent an alternative way of retrieving table rows. Normally you retrieve table rows using the row's primary key. By creating an index, you can retrieve rows with dissimilar primary key values, but which share some other characteristic.

You can create indexes on any field that has a data type capable of indexing, including primary key fields. You can index table IDENTITY fields. For information on the types of fields that can be indexed, see Indexable Field Types.

For example, if you had a table representing types of automobiles, the primary keys for each row might be the automobile's manufacturer and model type. However, if you wanted to be able to query for all red automobiles, regardless of the manufacturer or model type, you could create an index on the field containing color information.

Copyright © 2011, 2026, Oracle and/or its affiliates.

> **Note**
>
> Indexes can take a long time to create because Oracle NoSQL Database must examine all of the data contained in the relevant table in your store. The smaller the data contained in the table, the faster index creation will complete. Conversely, if a table contains a lot of data, then it can take a long time to create indexes for it.

```
CREATE TABLE myInventory.itemDetails (
    itemSKU STRING,
    itemDescription STRING,
    price FLOAT,
    inventoryCount INTEGER,
    PRIMARY KEY (itemSKU)
)
```

To create an index, use the `CREATE INDEX` statement. See CREATE INDEX. For example:

```
CREATE INDEX inventoryIdx on myInventory.itemDetails(inventoryCount)
```

Similarly, to remove an index, use the `DROP INDEX` statement. See DROP INDEX. For example:

```
DROP INDEX inventoryIdx on myInventory.itemDetails
```

Be aware that adding and dropping indexes can be time consuming. You may want to run drop index operations asynchronously using the `KVStore.execute()` method.

```
package kvstore.basicExample;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

import oracle.kv.ExecutionFuture;
import oracle.kv.FaultException;
import oracle.kv.StatementResult;
import oracle.kv.KVStore;
import oracle.kv.KVStoreConfig;
import oracle.kv.KVStoreFactory;
import oracle.kv.table.TableAPI;

...
// Store open skipped
...

public void createIndex() {
    TableAPI tableAPI = store.getTableAPI();
    ExecutionFuture future = null;
    StatementResult result = null;
    String statement = null;

    try {
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
            statement = "CREATE INDEX inventoryIdx on " +
                        "myInventory.itemDetails(inventoryCount)"
            future = store.execute(statement);
            displayResult(future.getLastStatus(), statement);

            /*
             * Limit the amount of time to wait for the
             * operation to finish.
             */
            result = future.get(3, TimeUnit.SECONDS);
            displayResult(result, statement);

        } catch (IllegalArgumentException e) {
            System.out.println("Invalid statement:\n" + e.getMessage());
        } catch (FaultException e) {
            System.out.println
                ("Statement couldn't be executed, please retry: " + e);
            cleanupOperation(future);
        } catch (ExecutionException e) {
            System.out.println
                ("Problem detected while waiting for a DDL statement: " +
                 e.getCause());
            cleanupOperation(future);
        } catch (InterruptedException e) {
            System.out.println
                ("Interrupted while waiting for a DDL statement: " + e);
            cleanupOperation(future);
        } catch (TimeoutException e) {
            System.out.println("Statement execution took too long: " + e);
            cleanupOperation(future);
        }
    }

    private void cleanupOperation(ExecutionFuture future) {
        if (future == null) {
            /* nothing to do */
            return;
        }

        System.out.println("Statement:");
        System.out.println(future.getStatement());
        System.out.println("has status: ");
        System.out.println(future.getLastStatus());

        if (!future.isDone()) {
            future.cancel(true);
            System.out.println("Statement is cancelled");
        }
    }

    private void displayResult(StatementResult result, String statement) {
        System.out.println("============================");
        if (result.isSuccessful()) {
            System.out.println("Statement was successful:\n\t" +
                               statement);
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
            System.out.println("Results:\n\t" + result.getInfo());
        } else if (result.isCancelled()) {
            System.out.println("Statement was cancelled:\n\t" +
                               statement);
        } else {
            /*
             * statement wasn't successful: may be in error, or may still be
             * in progress.
             */
            if (result.isDone()) {
                System.out.println("Statement failed:\n\t" + statement);
                System.out.println("Problem:\n\t" + result.getErrorMessage());
            } else {
                System.out.println("Statement in progress:\n\t" + statement);
                System.out.println("Status:\n\t" + result.getInfo());
            }
        }
    }
```

For examples of how to index supported non-scalar types, see Indexing Non-Scalar Data Types.

Copyright © 2011, 2026, Oracle and/or its affiliates.

4

Introducing Oracle NoSQL Database Namespaces

You can create one or more global namespaces to extend table identification. Namespaces permit tables with the same name to exist in your database store. To access such tables from the command line, or with DDL commands, use a fully-qualified table name with the table preceded by its namespace, followed with a colon (:), such as `ns1:table1`.

As with tables, you grant authorization permissions to determine who can access both the namespace and the tables within them. After your namespaces exist, you can create any number of parent and child tables within the namespace, such as `ns1:table1.child1`.

There is a default Oracle NoSQL Database namespace, called `sysdefault`. For new installations, all tables are assigned to the default `sysdefault` namespace, until or unless you create other namespaces, and create new tables within them. After upgrading from an earlier Oracle NoSQL Database release, all existing tables become part of the default `sysdefault` namespace automatically. You cannot change an existing table's namespace.

Tables in the `sysdefault` namespace do not require further qualification for existing queries. For example, using a basic SQL table query does not require the default namespace prefix (`sysdefault:`*`tablename`*), for SQL access. Also, the `TableAPI.getTable( )` method does not require any updates at existing sites, since it accepts one argument for table name, or two when you are ready to use namespaces. For information see Using and Setting Namespaces.

```
select * from salesincome;
```

Referencing a table name without a namespace prefix (*`namespace`*`:`) implies that the table is part of the default, `sysdefault` namespace. However, by creating namespaces, and then tables within them, you can have same name tables, such as the following for a `support_ticket` table:

- `support_ticket`
- `acme:support_ticket`
- `international_abc:support_ticket`
- `international_xyz:support_ticket`

No additional permissions are required for tables in the default `sysdefault` namespace, so existing authentication remains. For example, if you grant user `Joe` permission to access tables `t1`, `t3`, and `t4`, but not `t2`, Joe can still access all tables except `t2` after they are subsumed into the `sysdefault` namespace.

Creating Namespaces

You can add and one or more namespaces to your store. Then, create tables within the namespaces, and grant various permissions to users to access namespaces and tables. For the security administrator, several new permissions are available for namespaces, described in Granting Authorization Access to Namespaces .

Copyright © 2011, 2026, Oracle and/or its affiliates.

You create a new namespace by using a `CREATE NAMESPACE` DDL statement, as follows, with whatever name you choose.

```
sql-> CREATE NAMESPACE [IF NOT EXISTS] namespace_name;
```

All namespace names use standard identifiers, with the same restrictions as tables and indexes:

- Names must begin with an alphabetic character (a-z, A-Z).
- Remaining characters are alphanumeric (a-z, A-Z, 0–9).
- Name characters can include period (.), and underscore (_) characters.
- The maximum name length for a namespace is 128 characters.

> **Note**
>
> You cannot use the prefix `sys` for any namespaces. The `sys` prefix is reserved. No other keywords are restricted.

Following is the namespace syntax showing the identifier rules similar to a table name:

```
table_name : (namespace ':' )? id_path;
-namespace :  id_path ;

id_path : id (DOT id)* ;
id : (... | ID );
ID : ALPHA (ALPHA | DIGIT | UNDER)* ;
```

Here are a couple of examples using DDL statements in the SQL CLI. The `SHOW NAMESPACES` directive in the SQL CLI lists the namespaces that currently exist:

```
sql-> CREATE NAMESPACE ns1;
Statement completed successfully

sql-> CREATE NAMESPACE IF NOT EXISTS ns2;
Statement completed successfully

sql-> SHOW NAMESPACES
namespaces
  ns1
  sysdefault
  ns2
```

Granting Authorization Access to Namespaces

You can manage permission for users or roles to access namespaces and tables. These are the applicable permissions given to the developers and other users:

Copyright © 2011, 2026, Oracle and/or its affiliates.

Table 4-1 Namespace Privileges and Permissions

Privilege	Description
CREATE_ANY_NAMESPACE DROP_ANY_NAMESPACE	Grant permission to a role to create or drop any namespace. GRANT CREATE_ANY_NAMESPACE TO <*Role*>; GRANT DROP_ANY_NAMESPACE TO <*Role*>;
CREATE_TABLE_IN_NAMESPACE DROP_TABLE_IN_NAMESPACE EVOLVE_TABLE_IN_NAMESPACE	Grant permission to a user or to a role to create, drop or evolve tables in a specific namespace. You can evolve tables to update table definitions, add or remove fields, or change field properties, such as a default value. You may even add a particular kind of column, like an IDENTITY column, to increment some value automatically. Only tables that already exist in the store are candidates for table evolution. For more details, see Alter Table. GRANT CREATE_TABLE_IN_NAMESPACE ON NAMESPACE *namespace_name* TO <*User*|*Role*>; GRANT DROP_TABLE_IN_NAMESPACE ON NAMESPACE *namespace_name* TO <*User*|*Role*>; GRANT EVOLVE_TABLE_IN_NAMESPACE ON NAMESPACE *namespace_name* TO <*User*|*Role*>*user_role*;
CREATE_INDEX_IN_NAMESPACE DROP_INDEX_IN_NAMESPACE	Grant permission to a user or to a role to create or drop an index in a specific namespace. GRANT CREATE_INDEX_IN_NAMESPACE ON NAMESPACE *namespace_name* TO <*User*|*Role*>; GRANT DROP_INDEX_IN_NAMESPACE ON NAMESPACE *namespace_name* TO <*User*|*Role*>;
READ_IN_NAMESPACE INSERT_IN_NAMESPACE DELETE_IN_NAMESPACE	Grant permission to a user or role to read, insert, or delete items in a specific namespace. GRANT READ_IN_NAMESPACE ON NAMESPACE *namespace_name* TO <*User*|*Role*>; GRANT INSERT_IN_NAMESPACE ON NAMESPACE *namespace_name* TO <*User*|*Role*>; GRANT DELETE_IN_NAMESPACE ON NAMESPACE *namespace_name* TO <*User*|*Role*>;
MODIFY_IN_NAMESPACE	Helper label for granting or revoking permissions to all DDL privileges for a specific namespace to a user or role. GRANT MODIFY_IN_NAMESPACE ON NAMESPACE *namespace_name* TO <*User*|*Role*>; REVOKE MODIFY_IN_NAMESPACE ON NAMESPACE *namespace_name* TO <*User*|*Role*>;

Copyright © 2011, 2026, Oracle and/or its affiliates.

Grant privileges on a namespace

You can grant permissions to a role or a user on a namespace. Following is the syntax for granting permissions on a namespace:

```
GRANT {Namespace-scoped privileges} ON NAMESPACE namespace_name TO <User|Role>
Namespace-scoped privileges ::= namespace_privilege [, namespace_privilege]
```

where,

- namespace_privilege

  The namespace privilege that can be granted to a user or a role. For more information on the applicable privileges, see the *Privilege* column in the Namespace Privileges and Permissions table.

- namespace_name

  The namespace that the user wishes to access.

- <User|Role>

  The name of the KVStore user or the role of a user.

For example, you can grant read access to a user for all the tables in the namespace.

```
GRANT READ_IN_NAMESPACE ON NAMESPACE ns1 TO Kate
```

Here, ns1 is the namespace and Kate is the user.

> **Note**
>
> The label MODIFY_IN_NAMESPACE can be used as a helper for granting or revoking permissions to all DDL privileges for a specific namespace to a user or role.

Revoke privileges on a namespace

You can revoke the permissions from a role or a user on a namespace. Following is the syntax for revoking the permissions on a namespace.

```
REVOKE {Namespace-scoped privileges} ON NAMESPACE namespace_name FROM <User|
Role>
Namespace-scoped privileges ::= namespace_privilege [, namespace_privilege]
```

where,

- namespace_privilege

  The namespace privilege that can be revoked from a user or a role. For more information on the applicable privileges, see the *Privilege* column in the Namespace Privileges and Permissions table.

- namespace_name

  The namespace that the user wishes to access.

- <User|Role>

Copyright © 2011, 2026, Oracle and/or its affiliates.

The name of the KVStore user or the role of a user.

For example, you can revoke the read access from a user for all the tables in the namespace.

```
REVOKE READ_IN_NAMESPACE ON NAMESPACE ns1 FROM Kate
```

Here, ns1 is the namespace and Kate is the user.

> **Note**
>
> The label MODIFY_IN_NAMESPACE can be used as a helper for granting or revoking permissions to all DDL privileges for a specific namespace to a user or role.

The following example shows a set of commands to grant and revoke privileges on a namespace to a user or role:

Example: Namespace Scoped Privileges

```
CREATE NAMESPACE IF NOT EXISTS ns1;
GRANT MODIFY_IN_NAMESPACE ON NAMESPACE ns1 TO <User|Role>;
CREATE TABLE ns1:t (id INTEGER, name STRING, primary key (id));
INSERT INTO ns1:t VALUES (1, 'Smith');
SELECT * FROM ns1:t;
REVOKE CREATE_TABLE_IN_NAMESPACE ON NAMESPACE ns1 FROM <User|Role>;
DROP NAMESPACE ns1 CASCADE;
```

> **Note**
>
> You can save all of the above commands as a **sql** script and execute it in a single command.

Using and Setting Namespaces

Once you have created one or more namespaces, and tables within them, you can fully qualify table names in any references. If your store has tables with the same name, the namespace differentiates them from each other.

Here is the syntax for specifying a fully qualified table, or child table name from the CLI:

```
namespace:tablename
namespace:tablename.child1
```

To reference a table in a namespace in a SELECT statement:

```
SELECT * FROM ns1:table1;
```

Set Namespace for Method Execution

You can use the `ExecuteOptions.setNamespace` method to set a default namespace for the duration of a `KVStore.execute()` method. While set, you do not need to qualify table and other

Copyright © 2011, 2026, Oracle and/or its affiliates.

object references. If you do not use `setNamespace`, or fully qualify table names, the store uses `sysdefault` as the default namespace.

```
ExecuteOptions.setNamespace("ns1");
SELECT * FROM table1;
```

Determine Table Namespace

To find out which namespace was set on an option object, use the `ExecuteOptions.getNamespace` method.

Get a Table in a Specific Namespace

You can call `TableAPI.getTable()` with two arguments:

```
TableAPI.getTable(String namespace, String tableFullName);
```

Here, the first argument for `TableAPI.getTable` method, *namespace*, is the namespace in which you created the table. If this argument is `NULL`, the method uses the default `sysdefault` namespace. This case is equivalent to calling the function with a single argument, described next.

The second argument, *tableFullName*, is the full table name. This interface retrieves only top-level tables, without parent tables. To retrieve child tables, use `TableAPI.getChildTable()`.

Get a Fully-Qualified Table

You can call `TableAPI.getTable()` with one argument:

```
TableAPI.getTable(String fullNamespaceName);
```

The *`fullNamespaceName`* argument indicates the full name or namespace-qualified name of the target table. If you supply an unqualified name, the method uses the `sysdefault` namespace. If you supply a namespace that contains a table name prefixed with a namespace followed with a colon (*namespace:*), this usage is equivalent to calling the function as `getTable(String,String)` with the *namespace*, and *TableFullName* described above.

Showing and Describing Namespaces

You can use the following ways to show namespaces and their tables from the SQL CLI:

- `SHOW`
- `DESCRIBE`

The next example shows creating a namespace (`ns1`), a table within that namespace (`ns1:foo`), and using `SHOW` namespaces and `SHOW table ns1:foo` to see the table hierarchy (that the table was created in the ns1 namespace). Finally, using `DESCRIBE table ns1:foo` to see more table details:

```
sql-> create namespace ns1;
Statement completed successfully

sql-> create table ns1:foo (id integer, primary key (id));
Statement completed successfully
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
sql-> show namespaces;
namespaces
  ns1
  sysdefault

sql-> show table ns1:foo;
tableHierarchy(namespace ns1)
  foo

sql-> describe table ns1:foo;
 === Information ===
 +-----------+------+-----+-------+----------+----------+--------+----------
+---------+-------------+
 | namespace | name | ttl | owner | sysTable | r2compat | parent | children |
indexes | description |
 +-----------+------+-----+-------+----------+----------+--------+----------
+---------+-------------+
 | ns1       | foo  |     |       | N        | N        |        |
|         |             |
 +-----------+------+-----+-------+----------+----------+--------+----------
+---------+-------------+

 === Fields ===
 +----+------+---------+----------+-----------+----------+------------
+----------+
 | id | name |  type   | nullable |  default  | shardKey | primaryKey |
identity |
 +----+------+---------+----------+-----------+----------+------------
+----------+
 |  1 | id   | Integer | N        | NullValue | Y        | Y
|          |
 +----+------+---------+----------+-----------+----------+------------
+----------+
```

Dropping Namespaces

You can drop a namespace only if you have been granted the DROP_ANY_NAMESPACE privilege, and the namespace has no associated tables. Also, you must have the appropriate privileges.

To drop a namespace:

```
DROP NAMESPACE [IF EXISTS] namespace_name [CASCADE]
```

Using the CASCADE option with DROP NAMESPACE lets you extend the activity to tables and other objects within the NAMESPACE.

Dropping a namespace is not an atomic operation, and completes the following steps:

- First check to make sure privileges to drop a namespace exist. Continue if privileges are in place.
- If no privileges exist, stop process with an error.
- If privileges are in place and CASCADE is *not* specified, check for tables, or other objects in the namespace. Drop the namespace if no objects exist.

Copyright © 2011, 2026, Oracle and/or its affiliates.

- If tables or other objects exist in the namespace, stop process with an error.
- If privileges are in place to drop the namespace and CASCADE is specified, the statement drops the namespace, removing all tables, indexes related to the tables, and table privileges.

> **Note**
>
> You cannot drop the default namespace, `sysdefault`.

Copyright © 2011, 2026, Oracle and/or its affiliates.

5

Primary and Shard Key Design

Primary keys and *shard keys* are important concepts for your table design. What you use for primary and shard keys has implications in terms of your ability to read multiple rows at a time. But beyond that, your key design has important performance implications.

Primary Keys

Every table must have one or more fields designated as the primary key. This designation occurs at the time that the table is created, and cannot be changed after the fact. Oracle NoSQL Database supports the following types for the primary key: INTEGER, LONG, FLOAT, DOUBLE, NUMBER, STRING, ENUM, BOOLEAN, and TIMESTAMP.

A table's primary key uniquely identifies every row in the table. In the simplest case, it is used to retrieve a specific row so that it can be examined and/or modified.

For example, a table might have five fields: `productName`, `productType`, `color`, `size`, and `inventoryCount`. To retrieve individual rows from the table, it might be enough to just know the product's name. In this case, you would set the primary key field as `productName` and then retrieve rows based on the product name that you want to examine/manipulate.

The table statement you use to define this table is:

```
CREATE TABLE myProducts (
    productName STRING,
    productType STRING,
    color ENUM (blue,green,red),
    size ENUM (small,medium,large),
    inventoryCount INTEGER,
    // Define the primary key. Every table must have one.
    PRIMARY KEY (productName)
)
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

Note

- If the primary key field is an INTEGER data type, you can apply a serialized size constraint to it. See Integer Serialized Constraints.
- If the primary key must represent an approximate numeric type, you have an option to choose between FLOAT and DOUBLE data types. When serialized as a primary key, the DOUBLE data type uses 10 bytes and stores up to 16 decimal places. The FLOAT data type uses 5 bytes and stores up to 8 decimal places. Therefore, DOUBLE data type is more suitable as it provides a higher precision and a wider range as compared to the FLOAT data type.
- If the primary key must represent a string data type, you have an option to choose between STRING and ENUM. The ENUM data type mandates choosing from a list of permitted values. When used as a primary key, the cardinality is small. Also, any addition or deletion to the ENUM values requires schema evolution. The primary keys can only be changed by dropping and recreating the table. Therefore, changing the ENUM values of a primary key is an invasive operation. The STRING data type is better suited for primary keys as its value is a sequence of zero or more Unicode characters

Composite Keys

You can use multiple fields for your primary key, which is termed as a composite key.

For example, consider the following table statement:

```
CREATE TABLE myProducts (
    productName STRING,
    productType STRING,
    color ENUM (blue,green,red),
    size ENUM (small,medium,large),
    inventoryCount INTEGER,
    // Define the primary key. Every table must have one.
    PRIMARY KEY (productName, productType)
)
```

Here, the columns `productName` and `productType` are both declared as primary key fields.

The composite keys are useful in cases where the keys conjointly help in identifying a unique row.

See Reading Table Rows to retrieve multiple rows from your table.

See Using multiDelete() to delete multiple rows at a time.

Data Type Limitations

Fields can be designated as primary keys only if they are declared to be one of the following types:

- Integer
- Long
- Number

Copyright © 2011, 2026, Oracle and/or its affiliates.

- Float
- Double
- String
- Timestamp
- Enum

Partial Primary Keys

Some of the methods you use to perform multi-row operations allow, or even require, a partial primary key. A partial primary key is, simply, a key where only some of the fields comprising the row's primary key are specified.

For example, the following example specifies three fields for the table's primary key:

```
CREATE TABLE myProducts (
    productName STRING,
    productType STRING,
    productClass STRING,
    color ENUM (blue,green,red),
    size ENUM (small,medium,large),
    inventoryCount INTEGER,
    // Define the primary key. Every table must have one.
    PRIMARY KEY (productName, productType, productClass)
)
```

In this case, a full primary key would be one where you provide value for all three primary key fields: `productName`, `productType`, and `productClass`. A partial primary key would be one where you provide values for only one or two of those fields.

Note that order matters when specifying a partial key. The partial key must be a subset of the full key, starting with the first field specified and then adding fields in order. So the following partial keys are valid:

- `productName`
- `productName`, `productType`

Shard Keys

Shard keys identify which primary key fields are meaningful in terms of shard storage. That is, rows that contain the same values for all the shard key fields are guaranteed to be stored on the same shard offering high-performance retrievals and horizontal scalability. This matters for some operations that promise atomicity of the results. (See Executing a Sequence of Operations for more information.)

For example, suppose you set the following primary keys:

```
PRIMARY KEY (productType, productName, productClass)
```

You can guarantee that rows are placed on the same shard using the values set for the `productType` and `productName` fields like this:

```
PRIMARY KEY (SHARD(productType, productName), productClass)
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

> **Note**
>
> - The order matters when it comes to the shard keys. The keys must be specified in the order that they are defined as primary keys, with no gaps in the key list. In other words, given the above example, it is impossible to set `productType` and `productClass` as shard keys without also specifying `productName` as a shard key.
> - The shard keys can't be declared in the create table statement of a non-root table (in cases of child tables). An error is returned in such scenarios.

For more details on table modeling and design using primary keys, see Choice of Keys in NoSQL Database.

Row Data

There are no restrictions on the size of your rows, or the amount of data that you store in a field. However, you should consider your store's performance when deciding how large you are willing to allow your individual tables and rows to become. As is the case with any data storage scheme, the larger your rows, the longer it takes to read the information from storage, and to write the information to storage.

On the other hand, every table row carries with it some amount of overhead. Also, as the number of your rows grows very large, search times may be adversely affected. As a result, choosing to use a large number of tables, each of which use rows with just a small handful of fields, can also harm your store's performance.

Therefore, when designing your tables' content, you must find the appropriate balance between a small number of tables, each of which uses very large rows; and a large number of tables, each of which uses very small rows. You should also consider how frequently any given piece of information will be accessed.

For example, suppose your table contains information about users, where each user is identified by their first and last names (surname and familiar name). There is a set of information that you want to maintain about each user. Some of this information is small in size, and some of it is large. Some of it you expect will be frequently accessed, while other information is infrequently accessed.

Small properties are:

- name
- gender
- address
- phone number

Large properties are:

- image file
- public key 1
- public key 2
- recorded voice greeting

There are several possible ways you can organize this data. How you should do it depends on your data access patterns.

Copyright © 2011, 2026, Oracle and/or its affiliates.

For example, suppose your application requires you to read and write all of the properties identified above every time you access a row. (This is unlikely, but it does represent the simplest case.) In that event, you might create a single table with rows containing fields for each of the properties you maintain for the users in your application.

However, the chances are good that your application will not require you to access *all* of a user's properties every time you access his information. While it is possible that you will always need to read all of the properties every time you perform a user look up, it is likely that on updates you will operate only on some properties.

Given this, it is useful to consider how frequently data will be accessed, and its size. Large, infrequently accessed properties should be placed in tables other than that used by the frequently accessed properties.

For example, for the properties identified above, suppose the application requires:

- all of the small properties to always be used whenever the user's record is accessed.
- all of the large properties to be read for simple user look ups.
- on user information updates, the public keys are always updated (written) at the same time.
- The image file and recorded voice greeting can be updated independently of everything else.

In this case, you might store user properties using a table and a child table. The parent table holds rows containing all the small properties, plus public keys. The child table contains the image file and voice greeting.

```
CREATE TABLE userInfo (
    surname STRING,
    familiarName STRING,
    gender ENUM (male,female),
    street STRING,
    city STRING,
    state STRING,
    zipcode STRING,
    userPhone STRING,
    publickey1 BINARY,
    publickey2 BINARY,
    PRIMARY KEY (SHARD(surname), familiarName)
)


CREATE TABLE userInfo.largeProps (
    propType STRING,
    voiceGreeting BINARY,
    imageFile BINARY,
    PRIMARY KEY (propType)
)
```

Because the parent table contains all the data that is accessed whenever user data is accessed, you can update that data all at once using a single atomic operation. At the same time, you avoid retrieving the big data values whenever you retrieve a row by splitting the image data and voice greeting into a child table.

Copyright © 2011, 2026, Oracle and/or its affiliates.

6

Writing and Deleting Table Rows

This chapter discusses two different write operations: putting table rows into the store, and then deleting them.

Write Exceptions

There are many exceptions that you should handle whenever you perform a write operation to the store. Some of the more common exceptions are described here. For simple cases where you use default policies or are not using a secure store, you can probably avoid explicitly handling these. However, as your code complexity increases, so too will the desirability of explicitly managing these exceptions.

The first of these is `DurabilityException`. This exception indicates that the operation cannot be completed because the durability policy cannot be met. For more information, see Durability Guarantees.

The second is `RequestTimeoutException`. This simply means that the operation could not be completed within the amount of time provided by the store's timeout property. This probably indicates an overloaded system. Perhaps your network is experiencing a slowdown, or your store's nodes are overloaded with too many operations (especially write operations) coming in too short of a period of time.

To handle a `RequestTimeoutException`, you could simply log the error and move on, or you could pause for a short period of time and then retry the operation. You could also retry the operation, but use a longer timeout value. (There is a version of the `TableAPI.put()` method that allows you to specify a timeout value for that specific operation.)

You can also receive an `IllegalArgumentException`, which will be thrown if a `Row` that you are writing to the store does not have a primary key or is otherwise invalid.

You can also receive a general `FaultException`, which indicates that some exception occurred which is neither a problem with durability nor a problem with the request timeout. Your only recourse here is to either log the error and move along, or retry the operation.

Finally, if you are using a secure store that requires authentication, you can receive `AuthenticationFailureException` or `AuthenticationRequiredException` if you do not provide the proper authentication credentials. When using a secure store, you can also see `UnauthorizedException`, which means you are attempting an operation for which the authenticated user does not have the proper permissions.

Writing Rows to a Table in the Store

Writing a new row to a table in the data store, and updating an existing row are similar operations. Later in this section, we describe methods that work only if a row is being updated, or only if you are creating a row. You can write data to a table only after it has been added to the store. See Introducing Oracle NoSQL Database Tables and Indexes for details.

Along with writing a row to the table, you can also add user-supplied metadata for each individual row. This metadata is stored along with the row but remains independent from your primary data model. You can use `setLastWriteMetadata()` from the `PutRequest` class to insert

Copyright © 2011, 2026, Oracle and/or its affiliates.

metadata along with the row. For more information with an example, see Using User-Supplied Metadata in Write Operations, in *Developers Guide*.

To write a row to a table in the store:

1. Construct a handle for the table to which are writing data. You do this by retrieving a `TableAPI` interface instance using `KVStore.getTableAPI()`. Use that instance to retrieve a handle for the table using the `TableAPI.getTable()`, which then returns a `Table` interface instance.

   > **Note**
   >
   > The `TableAPI.getTable()` method is an expensive call requiring server side access. For best performance, do not call this method each time you need a table handle. If possible, call this method for all relevant tables in the set up section of your code. Then, reuse the handles throughout your application.

2. Use the `Table.createRow()` method to create a Row interface instance, using the `Table` instance you retrieved in the previous step.
3. Using the `Row.put()` method, write to each field in the row.

   To write a `NULL` value, use `Row.putNull()`, rather than `Row.put()`.
4. Write the new row to the store using `TableAPI.put()`.

   > **Note**
   >
   > If the table you are writing to contains an IDENTITY column, the generated value from the sequence generator will be available in the row.

You can also load rows into the store using special purpose streams. For more information, see Bulk Put Operations.

The following example shows how to write a row to the store, assuming that you have already created the `KVStore` handle.

```
package kvstore.basicExample;

import oracle.kv.KVStore;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

// The name you give getTable() must be identical
// to the name of the table when you created it with
// the CREATE TABLE DDL statement (myTable in this example).
Table myTable = tableH.getTable("myTable");
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
// Get a Row instance
Row row = myTable.createRow();

// Use row.put to put all of the cells into the row.
// This does NOT actually write the data to the store.

row.put("item", "Bolts");
row.put("description", "Hex head, stainless");
row.put("count", 5);
row.put("percentage", 0.2173913);

// Now write the table to the store.
// "item" is the row's primary key. If we had not set that key and its value,
// this operation will result in an IllegalArgumentException.
tableH.put(row, null, null);
```

Writing Rows to a Child Table

To write to a child table, complete the tasks that you do for a parent table, except using the two-part table name, such as *parent-table.child-table*.

For example, in Defining Child Tables we showed how to create a child table. To write data to that table, do this:

```
package kvstore.basicExample;

import oracle.kv.KVStore;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

// Get the corresponding child table
Table myChildTable = tableH.getTable("myInventory.itemDetails");

// Get a row instance
Row childRow = myChildTable.createRow();

// Populate the rows. Because the parent table's "itemCategory"
// field is a primary key, this must be populated in addition
// to all of the child table's rows
childRow.put("itemCategory", "Bolts");
childRow.put("itemSKU", "1392610");
childRow.put("itemDescription", "1/4-20 x 1/2 Grade 8 Hex");
childRow.put("price", new Float(11.99));
childRow.put("inventoryCount", 1457);
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

Writing rows to an IDENTITY column

Special considerations arise when you are inserting values into an IDENTITY column programmatically. This section presents the issues that exist, and how to work around them using `put()` and other methods.

You create each IDENTITY column in a table with one of these choices:

- GENERATED ALWAYS AS IDENTITY
- GENERATED BY DEFAULT AS IDENTITY
- GENERATED BY DEFAULT ON NULL AS IDENTITY

Additionally, an identity column can be a primary key, which prevents you from changing the IDENTITY value.

Each of the ways in which you create your identity column affects activities when you add rows using the `put` function, with one of its variants:

- `put` (unconditional)
- `put if absent` (only if the row does not have values)
- `put if present` (only if the row has values)

This section describes the different effects of inserting and updating IDENTITY columns.

For example, create the following table with a column defined with GENERATED ALWAYS AS IDENTITY. The IDENTITY field is a primary key:

```
CREATE Table foo(
  idValue INTEGER GENERATED ALWAYS AS IDENTITY
  (START WITH 1 INCREMENT BY 1 MAXVALUE 2 NO CYCLE),
  name STRING,
  PRIMARY KEY(idValue));
```

Insert a row into the IDENTITY Column

To insert a row into the `foo` table, here's what to do in your application:

```
L1: TableAPI api = store.getTableAPI(); // Gets the TableAPI for the store
L2: Table table = api.getTable("foo"); // Gets the Table foo instance
L3: Row row = table.createRow(); // constructs an empty Row row for Table
foo.
L4: row.put("name", "joe"); // populates the values for the Row row
L5: api.put(row, null /* previous-row */, null /* write-options */);

// The client driver recognizes that the system must generate the id values
and \
generates value 1 for the id field in the row before putting it in the DB.
L6: System.out.println("Value of idValue: " + row.get("idValue")); // 1
L7: row.put("name", "smith");
L8: api.put(row, null /* previous-row */, null /* write-options */);
// driver sets id field to 2
L9: System.out.println("Value of id: " + row.get("idValue")); // 2
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

> **Note**
>
> To get the value of a generated IDENTITY column, use a `get()` call to the IDENTITY column, as shown in **L6** and `L9`.

Also, to return the `idValue` use the `RETURNING idValue` clause, as follows:

```
StatementResult sr = store.executeSync("INSERT INTO foo " + "(name) VALUES
('foe')
RETURNING idValue");
int id = sr.iterator().next().get("idValue").asInteger().get();
```

Updating an IDENTITY Column

When you define a column as `GENERATED ALWAYS AS IDENTITY` you cannot supply a value for the IDENTITY column, because the Sequence Generator(SG) must always supply the next value.

The following example illustrates what happens when you try to specify a value for the IDENTITY column. The first additions, `joe` and `john` are fine, and the SG supplies an `idValue` for both:

```
CREATE TABLE foo(
 idValue INTEGER GENERATED ALWAYS AS IDENTITY,
 name STRING, PRIMARY KEY (idValue))
 api.put('joe')
 api.put('john')
 get(idValue, name) or
select * from foo;
1, joe
2, john
```

Trying to update with any of the put() methods causes the following errors when a column is defined as `GENERATED ALWAYS AS IDENTITY`:

```
api.put(2,'dave') // error –- cannot specify a value for \
 a column defined as GENERATED ALWAYS AS IDENTITY

api.putIfPresent (2, 'dave') -- The following error occurs first in the code
path,
even though idValue = 2 is present
// error – user cannot specify a value for \
 IDENTITY column defined as GENERATED ALWAYS

api.putIfPresent (3,'cezar') -- The following error occurs, first in the code
path,
even though idValue = 3 is NOT present
// error - user cannot specify a value for \
 IDENTITY column defined as GENERATED ALWAYS

api.putIfPresent ('hema')
//error – a primary key is not provided to look up the record.

putIfAbsent (10, 'joe') -– is an insert
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
// error - user cannot specify a value for \
IDENTITY column defined as GENERATED ALWAYS
```

To use UPDATE on a column defined as `GENERATED ALWAYS AS IDENTITY`:

```
Create table foo( idValue INTEGER GENERATED ALWAYS AS IDENTITY,
 name STRING,
 PRIMARY KEY (idValue))

UPDATE foo SET idValue = 10 WHERE name='joe'
// error -  user cannot set a value for an IDENTITY column defined as
GENERATED ALWAYS
 UPDATE foo SET name=hema WHERE idValue=2
// Success! By using the Primary Key value (idValue=2)to locate its name
record,
// you can update the value and hema replaces john
select * from foo
1, joe
2, hema
```

To use `put`, `putIfPresent`, and `putIfAbsent` on an IDENTITY column that is not a PRIMARY KEY:

```
Create table Foo(idValue INTEGER GENERATED ALWAYS AS IDENTITY,
 acctNumber INTEGER,
 name STRING,
 PRIMARY KEY (acctNumber) )

//Put two acctNumber and name values.
api.put(100, 'joe')
api.put (200, 'john')

//SG increments the IDENTITY values, 1 and 2:
api.get(idValue, acctNumber, name)
1, 100, joe
2, 200, john

//Attempt to put an idValue
api.put (2, 200,'dave')
// error – Cannot specify a value for IDENTITY column defined as GENERATED
ALWAYS

api.putIfPresent(3, 200,'cezar')
//error – Cannot specify a value for IDENTITY column defined as GENERATED
ALWAYS

api.putIfPresent (400,'cezar')  // not IDENTITY column value error
// error - Cannot specify a primary key (400) that is not present

api.putIFPresent (200,'cezar')
1, 100, joe
2, 200, cezar
// Success! The IDENTITY value is updated.
The system generates a value on putIfPresent as the API semantics are to
update
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

the entire record, and not update fields within the record selectively.

```
api.putIfAbsent (300,'hema')
// Success! IDENTITY idValue was generated (3), and 300, hema were absent
get(idValue, acctNumber, name)
1, 100, joe
2, 200, cezar
3, 300, hema

api.putIfAbsent (20, 300,'hema')
// error – user cannot specify a value for IDENTITY column defined as
GENERATED ALWAYS

api.putIfAbsent (300,'hema')
//error - no row with primary key = 300 is present

api.putIfAbsent (3,400,'hema')
// error – user cannot specify a value for IDENTITY column defined as
GENERATED ALWAYS
```

To use `UPDATE` on an IDENTITY column that is not a PRIMARY KEY:

```
Create table Foo(idValue INTEGER GENERATED ALWAYS AS IDENTITY,
 acctNumber INTEGER,
 name STRING,
 PRIMARY KEY (acctNumber))

select * from foo
1, 100, joe
2, 200, cezar
3, 300, hema

UPDATE foo set name= dave where PRIMARY KEY = 200
// replaces (2, 200, cezar) with (2, 200, dave)

select * from foo;
1, 100, joe
2, 200, dave
3, 300, hema

UPDATE foo set name=george where acctNumber=100
// acctNumber is the PRIMARY KEY
// replaces (1, 100, joe) with (1, 100, george)
select * from foo;
1, 100, george
2, 200, dave
3, 300, hema

UPDATE foo set idValue=10 where acctNumber=100
// acctNumber is the PRIMARY KEY
// error - Cannot specify a value for IDENTITY column defined as GENERATED
ALWAYS
```

To use `put()` on a column defined as GENERATED BY DEFAULT AS IDENTITY, which is a PRIMARY KEY, review the following examples. In this case, not specifying a value for the

Copyright © 2011, 2026, Oracle and/or its affiliates.

IDENTITY column causes the SG to generate a value. Specifying a value, the system uses what you supply.

```
Create table foo( idValue INTEGER GENERATED BY DEFAULT AS IDENTITY,
 name STRING,
 PRIMARY KEY (idValue))

api.put('joe')
api.put('john')

//Since you supplied no idValue, SG supplies them:
get(idValue,name)
1, joe
2, john

//You supply 4 as the idValue, so system uses it
api.put (4,'george')

get(idValue, name)
1, joe
2, john
4, george

api.put (2,'sam') // replaces (2, john)  with (2, sam)
get(idValue,name)
1, joe
2, sam
4, george
```

To use `UPDATE()` on the column:

```
select * from foo;
1, joe
2, sam
4, george

UPDATE foo SET name='brian' where idValue=2
// Updates idValue 2 (2, sam) with 2, brian
select * from foo
1, joe
2, brian
4, george
```

You can update any column other than the primary key column using the UPDATE clause.

Deleting an IDENTITY Column
Deleting a row with an IDENTITY column follows the existing delete logic in the product. There is no change.

Other put Operations

Beyond the very simple usage of the `TableAPI.put()` method illustrated above, there are three other put operations that you can use:

- `TableAPI.putIfAbsent()`

Copyright © 2011, 2026, Oracle and/or its affiliates.

This method will only put the row if the row's primary key value DOES NOT currently exist in the table. That is, this method is successful only if it results in a *create* operation.

- `TableAPI.putIfPresent()`

  This method will only put the row if the row's primary key value already exists in the table. That is, this method is only successful if it results in an *update* operation.

- `TableAPI.putIfVersion()`

  This method will put the row only if the value matches the supplied version information. For more information, see Using Row Versions .

Bulk Put Operations

Bulk put operations allow you to load records supplied by special purpose streams into the store.

The bulk loading of the entries is optimized to make efficient use of hardware resources. As a result, this operation can achieve much higher throughput when compared with single put APIs.

The behavior of the bulk put operation with respect to duplicate entries contained in different streams is thus undefined. If the duplicate entries are just present in a single stream, then the first entry will be inserted (if it is not already present) and the second entry and subsequent entries will result in the invocation of `EntryStream.keyExists(E)` method. If duplicates exist across streams, then the first entry to win the race is inserted and subsequent duplicates will result in `EntryStream.keyExists(E)` being invoked on them.

To use bulk put, use one of the `TableAPI.put()` methods that provide bulk put. These accept a set of streams to bulk load data. The rows within each stream may be associated with different tables.

When using these methods, you can also optionally specify a `BulkWriteOptions` class instance which allows you to specify the durability, timeout, and timeout unit to configure the bulk put operation.

For example, suppose you are loading 1000 rows with 3 input streams:

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.atomic.AtomicLong;
import oracle.kv.BulkWriteOptions;
import oracle.kv.EntryStream;
import oracle.kv.FaultException;
import oracle.kv.KVStore;
import oracle.kv.KVStoreConfig;
import oracle.kv.KVStoreFactory;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;

...

// KVStore handle creation is omitted for brevity

...
Integer streamParallelism = 3;
Integer perShardParallelism = 3;
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
    Integer heapPercent = 30;
    // In this case, sets the amount of 1000 rows to load
    int nLoad = 1000;

    BulkWriteOptions bulkWriteOptions =
                              new BulkWriteOptions(null, 0, null);
    // Set the number of streams. The default is 1 stream.
    bulkWriteOptions.setStreamParallelism(streamParallelism);
    // Set the number of writer threads per shard.
    // The default is 3 writer threads.
    bulkWriteOptions.setPerShardParallelism(perShardParallelism);
    // Set the percentage of max memory used for bulk put.
    // The default is 40 percent.
    bulkWriteOptions.setBulkHeapPercent(heapPercent);

    System.err.println("Loading rows to " + TABLE_NAME + "...");

    final List<EntryStream<Row>> streams =
        new ArrayList<EntryStream<Row>>(streamParallelism);
    final int num = (nLoad + (streamParallelism - 1)) / streamParallelism;
    for (int i = 0; i < streamParallelism; i++) {
        final int min = num * i;
        final int max = Math.min((min + num) , nLoad);
        streams.add(new LoadRowStream(i, min, max));
    }

    final TableAPI tableImpl = store.getTableAPI();
    tableImpl.put(streams, bulkWriteOptions);

    long total = 0;
    long keyExists = 0;
    for (EntryStream<Row> stream: streams) {
        total += ((LoadRowStream)stream).getCount();
        keyExists += ((LoadRowStream)stream).getKeyExistsCount();
    }
    final String fmt = "Loaded %,d rows to %s, %,d pre-existing.";
    System.err.println(String.format(fmt, total, TABLE_NAME, keyExists));
```

You should implement the stream interface that supplies the data to be batched and loaded into the store. Entries are supplied by a list of `EntryStream` instances. Each stream is read sequentially, that is, each `EntryStream.getNext()` is allowed to finish before the next operation is issued. The load operation typically reads from these streams in parallel as determined by `BulkWriteOptions.getStreamParallelism()`.

```
private class LoadRowStream implements EntryStream<Row> {

    private final String name;
    private final long index;
    private final long max;
    private final long min;
    private long id;
    private long count;
    private final AtomicLong keyExistsCount;

    LoadRowStream(String name, long index, long min, long max) {
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
            this.index = index;
            this.max = max;
            this.min = min;
            this.name = name;
            id = min;
            count = 0;
            keyExistsCount = new AtomicLong();
        }

        @Override
        public String name() {
            return name + "-" + index + ": " + min + "~" + max;
        }

        @Override
        public Row getNext() {
            if (id++ == max) {
                return null;
            }
            final Row row = userTable.createRow();
            row.put("id", id);
            row.put("name", "name" + id);
            row.put("age", 20 + id % 50);
            count++;
            return row;
        }

        @Override
        public void completed() {
            System.err.println(name() + " completed, loaded: " + count);
        }

        @Override
        public void keyExists(Row entry) {
            keyExistsCount.incrementAndGet();
        }

        @Override
        public void catchException(RuntimeException exception, Row entry) {
            System.err.println(name() + " catch exception: " +
                               exception.getMessage() + ": " +
                               entry.toJsonString(false));
            throw exception;
        }

        public long getCount() {
            return count;
        }

        public long getKeyExistsCount() {
            return keyExistsCount.get();
        }
    }
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

Using Time to Live

Time to Live (TTL) is a mechanism that allows you to automatically expire table rows. TTL is expressed as the amount of time data is allowed to live in the store. Data which has reached its expiration timeout value can no longer be retrieved, and will not appear in any store statistics. Whether the data is physically removed from the store is determined by an internal mechanism that is not user-controllable.

TTL represents a minimum guaranteed time to live. Data expires on hour or day boundaries. This means that with a one hour TTL, there can be as much as two hours worth of unexpired data. For example (using a time format of hour:minute:second), given a one hour TTL, data written between 00:00:00.000 and 00:59:59.999 will expire at 02:00:00.000 because the data is guaranteed to expire no less than one hour from when it is written.

In case of MR Tables with TTL value defined, the rows replicated to other regions carry the expiration time when the row was written. This can be either the default table level TTL value or a row level override that is set by your application. Therefore, this row will expire in all the regions at the same time, irrespective of when they were replicated.

Expired data is invisible to queries and store statistics, but even so it is using disk space until it has been purged. Here, store statistics refer to the statistics related to your store's performance and availability. See Monitoring the Store. The expired data is purged from disk at some point in time after its expiration date. The exact time when the data is purged is driven by internal mechanisms and the workload on your store.

The TTL value for a table row can be updated at any time before the expiration value has been reached. Data that has expired can no longer be modified, and this includes its TTL value.

TTL is more efficient than manual user-deletion of the row because it avoids the overhead of writing a database log entry for the data deletion. The deletion also does not appear in the replication stream.

Specifying a TTL Value

TTL values are specified on a row by row basis using `Row.setTTL()`. This method accepts a `TimeToLive` class instance, which allows you to identify the number of days or hours the row will live in the store before expiring. A duration interval specified in days is recommended because this results in the least amount of storage consumed in the store. However, if you want a TTL value that is not an even multiple of days, then specify the TTL value in hours.

The code example from Writing Rows to a Table in the Store can be extended to specify a TTL value of 5 days like this:

```
package kvstore.basicExample;

import oracle.kv.KVStore;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TimeToLive;
import oracle.kv.table.TableAPI;

...

// KVStore handle creation is omitted for brevity

...
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
TableAPI tableH = kvstore.getTableAPI();
Table myTable = tableH.getTable("myTable");

// Get a Row instance
Row row = myTable.createRow();


// Add a TTL value to the row
row.setTTL(TimeToLive.ofDays(5));

// Now put all of the cells in the row.
row.put("item", "Bolts");
row.put("description", "Hex head, stainless");
row.put("count", 5);
row.put("percentage", 0.2173913);

// Now write the table to the store.
tableH.put(row, null, null);
```

Updating a TTL Value

To update the expiration time for a table row, you write the row as normal, and at the same time specify the new expiration time. However, you must also indicate that the expiration time is to be updated. By default, you can modify the row data and the expiration time will not be modified, even if you specify a new TTL value for the row.

To indicate that the the expiration time is to be updated, specify `true` to the `WriteOptions.setUpdateTTL()` method. For example, using the previous example, to change the TTL value to 10 days, do the following:

```
package kvstore.basicExample;

import oracle.kv.KVStore;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.Table.TimeToLive;
import oracle.kv.table.TableAPI;
import oracle.kv.table.WriteOptions;

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();
Table myTable = tableH.getTable("myTable");

// Get a Row instance
Row row = myTable.createRow();

// Change the TTL value for the row from 5 days to 10.
row.setTTL(TimeToLive.ofDays(10));
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
// Now put all of the cells in the row.
row.put("item", "Bolts");
row.put("description", "Hex head, stainless");
row.put("count", 5);
row.put("percentage", 0.2173913);

// Now write the table to the store.
tableH.put(row, null, new WriteOptions().setUpdateTTL(true));
```

Deleting TTL Expiration

If you have set a TTL value for a row and you later decide you do not want it to ever automatically expire, you can turn off TTL by setting a TTL value of `TimeToLive.DO_NOT_EXPIRE`:

```
package kvstore.basicExample;

import oracle.kv.KVStore;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.Table.TimeToLive;
import oracle.kv.table.TableAPI;
import oracle.kv.table.WriteOptions;

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();
Table myTable = tableH.getTable("myTable");

// Get a Row instance
Row row = myTable.createRow();


// Modify the row's TTL so that it will never expire
row.setTTL(TimeToLive.DO_NOT_EXPIRE);

// Now put all of the cells in the row.
row.put("item", "Bolts");
row.put("description", "Hex head, stainless");
row.put("count", 5);
row.put("percentage", 0.2173913);

// Now write the table to the store.
tableH.put(row, null, new WriteOptions().setUpdateTTL(true));
```

Setting Default Table TTL Values

You can set a default TTL value for the table when you define the table using the `USING TTL` DDL statement. It may be optionally applied when a table is created using `CREATE TABLE` or

Copyright © 2011, 2026, Oracle and/or its affiliates.

when a table is modified using one of the `ALTER TABLE` statements. See USING TTL for details on this statement.

For example:

```
CREATE TABLE myTable (
  item STRING,
  description STRING,
  count INTEGER,
  percentage DOUBLE,
  PRIMARY KEY (item) // Every table must have a primary key
) USING TTL 5 days
```

At program run time, you can examine the default TTL value for a table using the `Table.getDefaultTTL()` method.

Deleting Rows from the Store

You delete a single row from the store using the `TableAPI.delete()` method. Rows are deleted based on a `PrimaryKey`, which you obtain using the `Table.createPrimaryKey()` method. You can also require a row to match a specified version before it will be deleted. To do this, use the `TableAPI.deleteIfVersion()` method. Versions are described in Using Row Versions .

When you delete a row, you must handle the same exceptions as occur when you perform any write operation on the store. See Write Exceptions for a high-level description of these exceptions.

If you delete a row that has associated user-supplied metadata, both the row data and its metadata are permanently removed from the table. This metadata will be included in the change stream event, allowing the change stream subscriber to track the cause of deletion. For more information, see Using User-Supplied Metadata in Write Operations in *Developers Guide*.

```
package kvstore.basicExample;

import oracle.kv.KVStore;
import oracle.kv.table.PrimaryKey;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

// The name you give to getTable() must be identical
// to the name that you gave the table when you created
// the table using the CREATE TABLE DDL statement.
Table myTable = tableH.getTable("myTable");

// Get the primary key for the row that we want to delete
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
PrimaryKey primaryKey = myTable.createPrimaryKey();
primaryKey.put("item", "Bolts");

// Delete the row
// This performs a store write operation
tableH.delete(primaryKey, null, null);
```

Using multiDelete()

You can delete multiple rows at once in a single atomic operation, as long as they all share the shard key values. Recall that shard keys are at least a subset of your primary keys. This results in using a partial primary key, which is the shard key, to perform a multi-delete.

To delete multiple rows at once, use the `TableAPI.multiDelete()` method.

For example, suppose you create a table like this:

```
CREATE TABLE myTable (
    itemType STRING,
    itemCategory STRING,
    itemClass STRING,
    itemColor STRING,
    itemSize STRING,
    price FLOAT,
    inventoryCount INTEGER,
    PRIMARY KEY (SHARD(itemType, itemCategory, itemClass), itemColor,
    itemSize)
)
```

With tables containing data like this:

- Row 1:
  - itemType: Hats
  - itemCategory: baseball
  - itemClass: longbill
  - itemColor: red
  - itemSize: small
  - price: 12.07
  - inventoryCount: 127
- Row 2:
  - itemType: Hats
  - itemCategory: baseball
  - itemClass: longbill
  - itemColor: red
  - itemSize: medium
  - price: 13.07
  - inventoryCount: 201

Copyright © 2011, 2026, Oracle and/or its affiliates.

- Row 3:
  - itemType: Hats
  - itemCategory: baseball
  - itemClass: longbill
  - itemColor: red
  - itemSize: large
  - price: 14.07
  - inventoryCount: 39

In this case, you can delete all the rows sharing the partial primary key `Hats`, `baseball`, `longbill` as follows:

```
package kvstore.basicExample;

import oracle.kv.KVStore;
import oracle.kv.table.PrimaryKey;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

// The name you give to getTable() must be identical
// to the name that you gave the table when you created
// it using the CREATE TABLE DDL statement.
Table myTable = tableH.getTable("myTable");

// Get the primary key for the row that we want to delete
PrimaryKey primaryKey = myTable.createPrimaryKey();
primaryKey.put("itemType", "Hats");
primaryKey.put("itemCategory", "baseball");
primaryKey.put("itemClass", "longbill");

// Exception handling omitted
tableH.multiDelete(primaryKey, null, null);
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

7

Reading Table Rows

There are several ways to retrieve table rows from the store. You can:

1. Retrieve a single row at a time using the `TableAPI.get()` method.
2. Retrieve rows associated with a shard key (which is based on at least part of your primary keys) using either the `TableAPI.multiGet()` or `TableAPI.multiGetIterator()` methods.
3. Retrieve table rows that share a shard key, or an index key, using the `TableAPI.tableIterator()` method.
4. Retrieve and process records from each shard in parallel using a single key as the retrieval criteria. Use one of the `TableAPI.tableIterator()` or `TableAPI.tableKeysIterator()` methods that provide parallel scans.
5. Retrieve and process records from each shard in parallel using a set of keys as the retrieval criteria. Use one of the `TableAPI.tableIterator()` or `TableAPI.tableKeysIterator()` methods that provide bulk retrievals.

Each of these are described in the following sections.

Read Exceptions

Several exceptions can occur when you attempt a read operation in the store. The first of these is `ConsistencyException`. This exception indicates that the operation cannot be completed because the consistency policy cannot be met. For more information, see Consistency Guarantees.

The second exception is `RequestTimeoutException`. This means that the operation could not be completed within the amount of time provided by the store's timeout property. This probably indicates a store that is attempting to service too many read requests all at once. Remember that your data is partitioned across the shards in your store, with the partitioning occurring based on your shard keys. If you designed your keys such that a large number of read requests are occurring against a single key, you could see request timeouts even if some of the shards in your store are idle.

A request timeout could also be indicative of a network problem that is causing the network to be slow or even completely unresponsive.

To handle a `RequestTimeoutException`, you could simply log the error and move on, or you could pause for a short period of time and then retry the operation. You could also retry the operation, but use a longer timeout value.

You can also receive an `IllegalArgumentException`, which will be thrown if a `Row` that you are writing to the store does not have a primary key or is otherwise invalid.

You can also receive a general `FaultException`, which indicates that some exception occurred which is neither a problem with consistency nor a problem with the request timeout. Your only recourse here is to either log the error and move along, or retry the operation.

You can also receive a `MetadataNotFoundException`, which indicates that a client's metadata may be out of sync. It extends `FaultException` and can be caught by applications to trigger

Copyright © 2011, 2026, Oracle and/or its affiliates.

the need for a refresh of their metadata, and in particular, Table handles obtained via TableAPI.getTable().

Finally, if you are using a secure store that requires authentication, you can receive `AuthenticationFailureException` or `AuthenticationRequiredException` if you do not provide the proper authentication credentials. When using a secure store, you can also see `UnauthorizedException`, which means you are attempting an operation for which the authenticated user does not have the proper permissions.

Retrieving a Single Row

To retrieve a single row from the store:

1. Construct a handle for the table from which you want to read. You do this by retrieving a `TableAPI` class instance using `KVStore.getTableAPI()`. You then use that instance to retrieve the desired table handle using `TableAPI.getTable()`. This returns a `Table` class instance.

   > **Note**
   >
   > `TableAPI.getTable()` is an expensive call that requires server side access. From a performance point of view, it is a mistake to call this method whenever you need a table handle. Instead, call this method for all relevant tables in the set up section of your code, and then reuse those handles throughout your application.

2. Use the `Table` instance retrieved in the previous step to create a `PrimaryKey` class instance. In this case, the key you create must be the entire primary key.
3. Retrieve the row using `TableAPI.get()`. This performs a store read operation.
4. Retrieve individual fields from the row using the `Row.get()` method.

For example, in Writing Rows to a Table in the Store we showed a trivial example of storing a table row to the store. The following trivial example shows how to retrieve that row.

```
package kvstore.basicExample;

import oracle.kv.KVStore;
import oracle.kv.table.PrimaryKey;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

// The name you give to getTable() must be identical
// to the name that you gave the table when you created
// the table using the CREATE TABLE DDL statement.
Table myTable = tableH.getTable("myTable");
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
// Construct the PrimaryKey. This is driven by your table
// design, which designated one or more fields as
// being part of the table's primary key. In this
// case, we have a single field primary key, which is the
// 'item' field. Specifically, we want to retrieve the
// row where the 'item' field contains 'Bolts'.
PrimaryKey key = myTable.createPrimaryKey();
key.put("item", "Bolts");

// Retrieve the row. This performs a store read operation.
// Exception handling is skipped for this trivial example.
Row row = tableH.get(key, null);

// Now retrieve the individual fields from the row.
String item = row.get("item").asString().get();
String description = row.get("description").asString().get();
Integer count = row.get("count").asInteger().get();
Double percentage = row.get("percentage").asDouble().get();
```

Retrieve a Child Table

In Writing Rows to a Child Table we showed how to populate a child table with data. To retrieve that data, you must specify the primary key used for the parent table row, as well as the primary key for the child table row. For example:

```
package kvstore.basicExample;

import oracle.kv.KVStore;
import oracle.kv.table.PrimaryKey;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

// We omit retrieval of the parent table because it is not required.
Table myChildTable = tableH.getTable("myInventory.itemDetails");

// Construct the PrimaryKey. This key must contain the primary key
// from the parent table row, as well as the primary key from the
// child table row that you want to retrieve.
PrimaryKey key = myChildTable.createPrimaryKey();
key.put("itemCategory", "Bolts");
key.put("itemSKU", "1392610");

// Retrieve the row. This performs a store read operation.
// Exception handling is skipped for this trivial example.
Row row = tableH.get(key, null);
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
// Now retrieve the individual fields from the row.
String description = row.get("itemDescription").asString().get();
Float price = row.get("price").asFloat().get();
Integer invCount = row.get("inventoryCount").asInteger().get();
```

For information on how to iterate over nested tables, see Iterating with Nested Tables.

Using multiGet()

`TableAPI.multiGet()` allows you to retrieve multiple rows at once, so long as they all share the same shard keys. You must specify a full set of shard keys to this method.

Use `TableAPI.multiGet()` only if your retrieval set will fit entirely in memory.

For example, suppose you have a table that stores information about products, which is designed like this:

```
CREATE TABLE myTable (
    itemType STRING,
    itemCategory STRING,
    itemClass STRING,
    itemColor STRING,
    itemSize STRING,
    price FLOAT,
    inventoryCount INTEGER,
    PRIMARY KEY (SHARD(itemType, itemCategory, itemClass), itemColor,
    itemSize)
)
```

With tables containing data like this:

- Row 1:
  - itemType: Hats
  - itemCategory: baseball
  - itemClass: longbill
  - itemColor: red
  - itemSize: small
  - price: 12.07
  - inventoryCount: 127
- Row 2:
  - itemType: Hats
  - itemCategory: baseball
  - itemClass: longbill
  - itemColor: red
  - itemSize: medium
  - price: 13.07
  - inventoryCount: 201

Copyright © 2011, 2026, Oracle and/or its affiliates.

- Row 3:
  - itemType: Hats
  - itemCategory: baseball
  - itemClass: longbill
  - itemColor: red
  - itemSize: large
  - price: 14.07
  - inventoryCount: 39

In this case, you can retrieve all of the rows with their `itemType` field set to `Hats` and their `itemCategory` field set to `baseball`. Notice that this represents a partial primary key, because `itemClass`, `itemColor` and `itemSize` are not used for this query.

```
package kvstore.basicExample;

...

import java.util.List;
import java.util.Iterator;
import oracle.kv.ConsistencyException;
import oracle.kv.KVStore;
import oracle.kv.RequestTimeoutException;
import oracle.kv.table.PrimaryKey;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

// The name you give to getTable() must be identical
// to the name that you gave the table when you created
// the table using the CREATE TABLE DDL statement.
Table myTable = tableH.getTable("myTable");

// Construct the PrimaryKey. In this case, we are
// using a partial primary key.
PrimaryKey key = myTable.createPrimaryKey();
key.put("itemType", "Hats");
key.put("itemCategory", "baseball");
key.put("itemClass", "longbill");

List<Row> myRows = null;

try {
    myRows = tableH.multiGet(key, null, null);
} catch (ConsistencyException ce) {
    // The consistency guarantee was not met
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
} catch (RequestTimeoutException re) {
    // The operation was not completed within the
    // timeout value
}
```

You can then iterate over the resulting list as follows:

```
for (Row theRow: myRows) {
    String itemType = theRow.get("itemType").asString().get();
    String itemCategory = theRow.get("itemCategory").asString().get();
    String itemClass = theRow.get("itemClass").asString().get();
    String itemColor = theRow.get("itemColor").asString().get();
    String itemSize = theRow.get("itemSize").asString().get();
    Float price = theRow.get("price").asFloat().get();
    Integer price = theRow.get("itemCount").asInteger().get();
}
```

Iterating over Table Rows

`TableAPI.tableIterator()` provides non-atomic table iteration. Use this method to iterate over indexes. This method performs a parallel scan of your tables if you set a concurrent request size other than 1.

`TableAPI.tableIterator()` does not return the entire set of rows all at once. Instead, it batches the fetching of rows in the iterator, to minimize the number of network round trips, while not monopolizing the available bandwidth. Also, the rows returned by this method are in unsorted order.

Note that this method does not result in a single atomic operation. Because the retrieval is batched, the return set can change over the course of the entire retrieval operation. As a result, you lose the atomicity of the operation when you use this method.

This method provides for an unsorted traversal of rows in your table. If you do not provide a key, then this method will iterate over all of the table's rows.

When using this method, you can optionally specify:

- A `MultiRowOptions` class instance. This class allows you to specify a field range, and the ancestor and parent tables you want to include in this iteration.
- A `TableIteratorOptions` class instance. This class allows you to identify the suggested number of keys to fetch during each network round trip. If you provide a value of 0, an internally determined default is used. You can also use this class to specify the traversal order (`FORWARD`, `REVERSE`, and `UNORDERED` are supported).

  This class also allows you to control how many threads are used to perform the store read. By default this method determines the degree of concurrency based on the number of available processors. You can tune this concurrency by explicitly stating how many threads to use for table retrieval. See Parallel Scans for more information.

  Finally, you use this class to specify a consistency policy. See Consistency Guarantees for more information.

Copyright © 2011, 2026, Oracle and/or its affiliates.

> **Note**
>
> When using `TableAPI.tableIterator()`, it is important to call `TableIterator.close()` when you are done with the iterator to avoid resource leaks. This is especially true for long-running applications, especially if you do not iterate over the entire result set.

For example, suppose you have a table that stores information about products, which is designed like this:

```
CREATE TABLE myTable (
    itemType STRING,
    itemCategory STRING,
    itemClass STRING,
    itemColor STRING,
    itemSize STRING,
    price FLOAT,
    inventoryCount INTEGER,
    PRIMARY KEY (SHARD(itemType, itemCategory, itemClass), itemColor,
    itemSize)
)
```

With tables containing data like this:

- Row 1:
  - itemType: Hats
  - itemCategory: baseball
  - itemClass: longbill
  - itemColor: red
  - itemSize: small
  - price: 12.07
  - inventoryCount: 127
- Row 2:
  - itemType: Hats
  - itemCategory: baseball
  - itemClass: longbill
  - itemColor: red
  - itemSize: medium
  - price: 13.07
  - inventoryCount: 201
- Row 3:
  - itemType: Hats
  - itemCategory: baseball
  - itemClass: longbill

Copyright © 2011, 2026, Oracle and/or its affiliates.

  - itemColor: red
  - itemSize: large
  - price: 14.07
  - inventoryCount: 39
- Row *n*:
  - itemType: Coats
  - itemCategory: Casual
  - itemClass: Winter
  - itemColor: red
  - itemSize: large
  - price: 247.99
  - inventoryCount: 9

Then in the simplest case, you can retrieve all of the rows related to 'Hats' using `TableAPI.tableIterator()` as follows. Note that this simple example can also be accomplished using the `TableAPI.multiGet()` method. If you have a complete shard key, and if the entire results set will fit in memory, then `multiGet()` will perform much better than `tableIterator()`. However, if the results set cannot fit entirely in memory, or if you do not have a complete shard key, then `tableIterator()` is the better choice. Note that reads performed using `tableIterator()` are non-atomic, which may have ramifications if you are performing a long-running iteration over records that are being updated.

```
package kvstore.basicExample;

...

import oracle.kv.KVStore;
import oracle.kv.table.PrimaryKey;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;
import oracle.kv.table.TableIterator;

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

// The name you give to getTable() must be identical
// to the name that you gave the table when you created
// the table using the CREATE TABLE DDL statement.
Table myTable = tableH.getTable("myTable");

// Construct the PrimaryKey. In this case, we are
// using a partial primary key.
PrimaryKey key = myTable.createPrimaryKey();
key.put("itemType", "Hats");
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
// Exception handling is omitted, but in production code
// ConsistencyException, RequestTimeException, and FaultException
// would have to be handled.
TableIterator<Row> iter = tableH.tableIterator(key, null, null);
try {
    while (iter.hasNext()) {
        Row row = iter.next();
        // Examine your row's fields here
    }
} finally {
    if (iter != null) {
        iter.close();
    }
}
```

Specifying Field Ranges

When performing multi-key operations in the store, you can specify a range of rows to operate upon. You do this using the `FieldRange` class, which is accepted by any of the methods which perform bulk reads. This class is used to restrict the selected rows to those matching a range of field values.

For example, suppose you defined a table like this:

```
CREATE TABLE myTable (
    surname STRING,
    familiarName STRING,
    userID STRING,
    phonenumber STRING,
    address STRING,
    email STRING,
    dateOfBirth STRING,
    PRIMARY KEY (SHARD(surname, familiarName), userID)
)
```

The `surname` contains a person's family name, such as `Smith`. The `familiarName` contains their common name, such as `Bob`, `Patricia`, `Robert`, and so forth.

Given this, you could perform operations for all the rows related to users with a surname of `Smith`, but we can limit the result set to just those users with familiar names that fall alphabetically between `Bob` and `Patricia` by specifying a field range.

A `FieldRange` is created using `Table.createFieldRange()`. This method takes just one argument — the name of the primary key for which you want to set the range.

In this case, we will define the start of the key range using the string "Bob" and the end of the key range to be "Patricia". Both ends of the key range will be inclusive.

In this example, we use `TableIterator`, but we could just as easily use this range on any multi-row read operation, such as the `TableAPI.multiGet()` or `TableAPI.multiGetKeys()` methods. The `FieldRange` object is passed to these methods using a `MultiRowOptions` class

Copyright © 2011, 2026, Oracle and/or its affiliates.

instance, which we construct using the `FieldRange.createMultiRowOptions()` convenience method.

```
package kvstore.basicExample;

...

import oracle.kv.KVStore;
import oracle.kv.table.FieldRange;
import oracle.kv.table.MultiRowOptions;
import oracle.kv.table.PrimaryKey;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;
import oracle.kv.table.TableIterator;

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

// The name you give to getTable() must be identical
// to the name that you gave the table when you created
// the table using the CREATE TABLE DDL statement.
Table myTable = tableH.getTable("myTable");

// Construct the PrimaryKey. In this case, we are
// using a partial primary key.
PrimaryKey key = myTable.createPrimaryKey();
key.put("surname", "Smith");

// Create the field range.
FieldRange fh = myTable.createFieldRange("familiarName");
fh.setStart("Bob", true);
fh.setEnd("Patricia", true);
MultiRowOptions mro = fh.createMultiRowOptions();

// Exception handling is omitted, but in production code
// ConsistencyException, RequestTimeException, and FaultException
// would have to be handled.
TableIterator<Row> iter = tableH.tableIterator(key, mro, null);
try {
    while (iter.hasNext()) {
        Row row = iter.next();
        // Examine your row's fields here
    }
} finally {
    if (iter != null) {
        iter.close();
    }
}
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

Iterating with Nested Tables

When you are iterating over a table, or performing a multi-get operation, by default only rows are retrieved from the table on which you are operating. However, you can use `MultiRowOptions` to specify that parent and child tables are to be retrieved as well.

When you do this, parent tables are retrieved first, then the table you are operating on, then child tables. In other words, the tables' hierarchical order is observed.

The parent and child tables retrieved are identified by specifying a `List` of `Table` objects to the `ancestors` and `children` parameters on the class constructor. You can also specify these using the `MultiRowOptions.setIncludedChildTables()` or `MultiRowOptions.setIncludedParentTables()` methods.

When operating on rows retrieved from multiple tables, it is your responsibility to determine which table the row belongs to.

For example, suppose you create a table with a child and grandchild table like this:

```
CREATE TABLE prodTable (
    prodType STRING,
    typeDescription STRING,
    PRIMARY KEY (prodType)
)


CREATE TABLE prodTable.prodCategory (
    categoryName STRING,
    categoryDescription STRING,
    PRIMARY KEY (categoryName)
)


CREATE TABLE prodTable.prodCategory.item (
    itemSKU STRING,
    itemDescription STRING,
    itemPrice FLOAT,
    vendorUID STRING,
    inventoryCount INTEGER,
    PRIMARY KEY (itemSKU)
)
```

With tables containing data like this:

- Row 1:
  - prodType: Hardware
  - typeDescription: Equipment, tools and parts
  - Row 1.1:
    - categoryName: Bolts
    - categoryDescription: Metric & US Sizes
    - Row 1.1.1:
      - itemSKU: 1392610

Copyright © 2011, 2026, Oracle and/or its affiliates.

        - itemDescription: 1/4-20 x 1/2 Grade 8 Hex
        - itemPrice: 11.99
        - vendorUID: A8LN99
        - inventoryCount: 1457
- Row 2:
  - prodType: Tools
  - typeDescription: Hand and power tools
  - Row 2.1:
    - categoryName: Handtools
    - categoryDescription: Hammers, screwdrivers, saws
    - Row 2.1.1:
      - itemSKU: 1582178
      - itemDescription: Acme 20 ounce claw hammer
      - itemPrice: 24.98
      - vendorUID: D6BQ27
      - inventoryCount: 249

In this case, you can display all of the data contained in these tables in the following way.

Start by getting all our table handles:

```
package kvstore.tableExample;

import java.util.Arrays;

import oracle.kv.KVStore;
import oracle.kv.KVStoreConfig;
import oracle.kv.KVStoreFactory;

import oracle.kv.table.PrimaryKey;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;

import oracle.kv.table.TableIterator;
import oracle.kv.table.MultiRowOptions;

...

private static Table prodTable;
private static Table categoryTable;
private static Table itemTable;

private static TableAPI tableH;

...

// KVStore handle creation is omitted for brevity
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
...

tableH = kvstore.getTableAPI();
prodTable = tableH.getTable("prodTable");
categoryTable = tableH.getTable("prodTable.prodCategory");
itemTable = tableH.getTable("prodTable.prodCategory.item");
```

Now we need the `PrimaryKey` and the `MultiRowOptions` that we will use to iterate over the top-level table. Because we want all the rows in the top-level table, we create an empty `PrimaryKey`.

The `MultiRowOptions` identifies the two child tables in the constructor's `child` parameter. This causes the iteration to return all the rows from the top-level table, as well as all the rows from the nested children tables.

```
// Construct a primary key
PrimaryKey key = prodTable.createPrimaryKey();

// Get a MultiRowOptions and tell it to look at both the child
// tables
MultiRowOptions mro = new MultiRowOptions(null, null,
        Arrays.asList(categoryTable, itemTable));
```

Now we perform the iteration:

```
// Get the table iterator
// Exception handling is omitted, but in production code
// ConsistencyException, RequestTimeException, and FaultException
// would have to be handled.
TableIterator<Row> iter = tableH.tableIterator(key, mro, null);
try {
    while (iter.hasNext()) {
        Row row = iter.next();
        displayRow(row);
    }
} finally {
    if (iter != null) {
        iter.close();
    }
}
```

Our `displayRow()` method is used to determine which table a row belongs to, and then display it in the appropriate way.

```
private static void displayRow(Row row) {
    // Display the row depending on which table it belongs to
    if (row.getTable().equals(prodTable)) {
        displayProdTableRow(row);
    } else if (row.getTable().equals(categoryTable)) {
        displayCategoryTableRow(row);
    } else {
        displayItemTableRow(row);
    }
}
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

Finally, we just need the methods used to display each row. These are trivial, but in a more sophisticated application they could be used to do more complex things, such as construct HTML pages or write XSL-FO for the purposes of generating PDF copies of a report.

```
private static void displayProdTableRow(Row row) {
    System.out.println("\nType: " +
        row.get("prodType").asString().get());
    System.out.println("Description: " +
        row.get("typeDescription").asString().get());
}

private static void displayCategoryTableRow(Row row) {
    System.out.println("\tCategory: " +
        row.get("categoryName").asString().get());
    System.out.println("\tDescription: " +
        row.get("categoryDescription").asString().get());
}

private static void displayItemTableRow(Row row) {
    System.out.println("\t\tSKU: " +
        row.get("itemSKU").asString().get());
    System.out.println("\t\tDescription: " +
        row.get("itemDescription").asString().get());
    System.out.println("\t\tPrice: " +
        row.get("itemPrice").asFloat().get());
    System.out.println("\t\tVendorUID: " +
        row.get("vendorUID").asString().get());
    System.out.println("\t\tInventory count: " +
        row.get("inventoryCount").asInteger().get());
    System.out.println("\n");
}
```

Note that the retrieval order remains the top-most ancestor to the lowest child, even if you retrieve by lowest child. For example, you can retrieve all the Bolts, and all of their parent tables, like this:

```
// Get all the table handles
prodTable = tableH.getTable("prodTable");
categoryTable = tableH.getTable("prodTable.prodCategory");
itemTable = tableH.getTable("prodTable.prodCategory.item");

// Construct a primary key
PrimaryKey key = itemTable.createPrimaryKey();
key.put("prodType", "Hardware");
key.put("categoryName", "Bolts");

// Get a MultiRowOptions and tell it to look at both the ancestor
// tables
MultiRowOptions mro = new MultiRowOptions(null,
        Arrays.asList(prodTable, categoryTable), null);

// Get the table iterator
// Exception handling is omitted, but in production code
// ConsistencyException, RequestTimeException, and FaultException
// would have to be handled.
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
TableIterator<Row> iter = tableH.tableIterator(key, mro, null);
try {
    while (iter.hasNext()) {
        Row row = iter.next();
        displayRow(row);
    }
} finally {
    if (iter != null) {
        iter.close();
    }
}
```

Reading Indexes

You use `TableIterator` to retrieve table rows using a table's indexes. Just as when you use `TableIterator` to read table rows using a table's primary key(s), when reading using indexes you can set options such as field ranges, traversal direction, and so forth. By default, index scans return entries in forward order.

In this case, rather than provide `TableIterator` with a `PrimaryKey` instance, you use an instance of `IndexKey`.

For example, suppose you defined a table like this:

```
CREATE TABLE myTable (
    surname STRING,
    familiarName STRING,
    userID STRING,
    phonenumber STRING,
    address STRING,
    email STRING,
    dateOfBirth STRING,
    PRIMARY KEY (SHARD(surname, familiarName), userID)
)


CREATE INDEX DoB ON myTable (dateOfBirth)
```

This creates an index named `DoB` for table `myTable` based on the value of the `dateOfBirth` field. To read using that index, you use `Table.getIndex()` to retrieve the index named `Dob`. You then create an `IndexKey` from the `Index` object. For example:

```
package kvstore.basicExample;

...

import oracle.kv.KVStore;
import oracle.kv.table.Index;
import oracle.kv.table.IndexKey;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;
import oracle.kv.table.TableIterator;
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myTable");

// Construct the IndexKey. The name we gave our index when
// we created it was 'DoB'.
Index dobIdx = myTable.getIndex("DoB");
IndexKey dobIdxKey = dobIdx.createIndexKey();

// Exception handling is omitted, but in production code
// ConsistencyException, RequestTimeException, and FaultException
// would have to be handled.
TableIterator<Row> iter = tableH.tableIterator(dobIdxKey, null, null);
try {
    while (iter.hasNext()) {
        Row row = iter.next();
        // Examine your row's fields here
    }
} finally {
    if (iter != null) {
        iter.close();
    }
}
```

If you want to return entries that match a specific field name and field value, then use the `IndexKey.put()` method:

```
// Construct the IndexKey. The name we gave our index when
// we created it was 'DoB'.
Index dobIdx = myTable.getIndex("DoB");
IndexKey dobIdxKey = dobIdx.createIndexKey();
// Return only those entries with a dateOfBirth equal to
// "1991-08-23"
dobIdxKey.put("dateOfBirth", "1991-08-23");

// Exception handling is omitted, but in production code
// ConsistencyException, RequestTimeException, and FaultException
// would have to be handled.
TableIterator<Row> iter = tableH.tableIterator(dobIdxKey, null, null);
try {
    while (iter.hasNext()) {
        Row row = iter.next();
        // Examine your row's fields here
    }
} finally {
    if (iter != null) {
        iter.close();
    }
}
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

If you want to return all the entries with a null value for the field, use the `IndexKey.putNull()` method:

```
// Construct the IndexKey. The name we gave our index when
// we created it was 'DoB'.
Index dobIdx = myTable.getIndex("DoB");
IndexKey dobIdxKey = dobIdx.createIndexKey();
// Return only those entries with a NULL dateOfBirth
// value.
dobIdxKey.putNull("dateOfBirth");

// Exception handling is omitted, but in production code
// ConsistencyException, RequestTimeException, and FaultException
// would have to be handled.
TableIterator<Row> iter = tableH.tableIterator(dobIdxKey, null, null);
try {
    while (iter.hasNext()) {
        Row row = iter.next();
        // Examine your row's fields here
    }
} finally {
    if (iter != null) {
        iter.close();
    }
}
```

In the previous example, the code examines every row indexed by the `DoB` index. A more likely, and useful, example in this case would be to limit the rows returned through the use of a field range. You do that by using `Index.createFieldRange()` to create a `FieldRange` object. When you do this, you must specify the field to base the range on. Recall that an index can be based on more than one table field, so the field name you give the method must be one of the indexed fields.

For example, if the rows hold dates in the form of `yyyy-mm-dd`, you could retrieve all the people born in the month of May, 1994 in the following way. This index only examines one field, `dateOfBirth`, so we give that field name to `Index.createFieldRange()`:

```
package kvstore.basicExample;

...

import oracle.kv.KVStore;
import oracle.kv.table.FieldRange;
import oracle.kv.table.Index;
import oracle.kv.table.IndexKey;
import oracle.kv.table.MultiRowOption;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;
import oracle.kv.table.TableIterator;

...

// KVStore handle creation is omitted for brevity
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
...

TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myTable");

// Construct the IndexKey. The name we gave our index when
// we created it was 'DoB'.
Index dobIdx = myTable.getIndex("DoB");
IndexKey dobIdxKey = dobIdx.createIndexKey();

// Create the field range.
FieldRange fh = dobIdx.createFieldRange("dateOfBirth");
fh.setStart("1994-05-01", true);
fh.setEnd("1994-05-30", true);
MultiRowOptions mro = fh.createMultiRowOptions();

// Exception handling is omitted, but in production code
// ConsistencyException, RequestTimeException, and FaultException
// would have to be handled.
TableIterator<Row> iter = tableH.tableIterator(dobIdxKey, mro, null);
try {
    while (iter.hasNext()) {
        Row row = iter.next();
        // Examine your row's fields here
    }
} finally {
    if (iter != null) {
        iter.close();
    }
}
```

Parallel Scans

By default, store reads are performed using multiple threads, the number of which is chosen by the number of cores available to your code. You can configure the maximum number of client-side threads to be used for the scan, as well as the number of results per request and the maximum number of result batches that the Oracle NoSQL Database client can hold before the scan pauses. To do this, use the `TableIteratorOptions` class. You pass this to `TableAPI.tableIterator()`. This creates a `TableIterator` that uses the specified parallel scan configuration.

> **Note**
>
> You cannot configure the number of scans you use for your reads if you are using indexes.

For example, to retrieve all of the records in the store using 5 threads in parallel, you would do this:

```
package kvstore.basicExample;
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
...

import oracle.kv.Consistency;
import oracle.kv.Direction;
import oracle.kv.KVStore;
import oracle.kv.table.FieldRange;
import oracle.kv.table.PrimaryKey;
import oracle.kv.table.MultiRowOption;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;
import oracle.kv.table.TableIterator;
import oracle.kv.table.TableIteratorOptions;

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myTable");

// Construct the PrimaryKey.
PrimaryKey key = myTable.createPrimaryKey();
key.put("itemType", "Hats");
key.put("itemCategory", "baseball");

TableIteratorOptions tio =
    new TableIteratorOptions(Direction.UNORDERED,
                             Consistency.NONE_REQUIRED,
                             0,     // timeout
                             null,  // timeout units
                             5,     // number of concurrent
                                    // threads
                             0,     // results per request
                             0);    // max result sets
// Exception handling is omitted, but in production code
// ConsistencyException, RequestTimeException, and FaultException
// would have to be handled.
TableIterator<Row> iter =
    tableH.tableIterator(key, null, tio);
try {
    while (iter.hasNext()) {
        Row row = iter.next();
        // Examine your row's fields here
    }
} finally {
    if (iter != null) {
        iter.close();
    }
}
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

Bulk Get Operations

Bulk get operations allow you to retrieve and process records from each shard in parallel, like a parallel scan, but using a set of keys instead of a single key as retrieval criteria.

A bulk get operation does not return the entire set of rows all at once. Instead, it batches the fetching of rows in the iterator, to minimize the number of network round trips, while not monopolizing the available bandwidth. Batches are fetched in parallel across multiple Replication Nodes. If more threads are specified on the client side, then the user can expect better retrieval performance – until processor or network resources are saturated.

To use bulk get, use one of the `TableAPI.tableIterator()` or `TableAPI.tableKeysIterator()` methods that provide bulk retrievals. These accept a set of keys instead of a single key as the retrieval criteria. The set is provided using either an `Iterator<Key>` or `List<Iterator<Key>>` value.

The methods retrieve the rows or primary keys matching the keys supplied by the iterator(s).

> **Note**
>
> If the iterator yields duplicate keys, the row associated with the duplicate keys will be returned at least once and potentially multiple times.

The supplied keys should follow these rules:

1. All supplied primary keys should belong to the same table.
2. The input key must be a complete shard key.
3. If a field range is specified, then the partial primary keys should be uniform. That is, they should have the same number of components. Also, the field range must be the first unspecified field of the supplied key.

When using these methods, you can also optionally specify:

- A `MultiRowOptions` class instance which allows you to specify a field range, as well as the ancestor and parent tables you want to include in the iteration.
- The number of keys to fetch during each network round trip using a `TableIteratorOptions` class instance. If you provide a value of 0, an internally determined default is used. You can also specify the traversal order (`UNORDERED` is supported).

  You can control how many threads are used to perform the store read using the `MaxConcurrentRequests` parameter.

  Finally, you can specify a consistency policy. See Consistency Guarantees for more information.

For example, suppose you have a table that stores information about products, which is designed like this:

```
CREATE TABLE myTable (
    itemType STRING,
    itemCategory STRING,
    itemClass STRING,
    itemColor STRING,
    itemSize STRING,
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
price FLOAT,
inventoryCount INTEGER,
PRIMARY KEY (SHARD(itemType, itemCategory), itemClass, itemColor,
itemSize))
```

With tables containing data like this:

- Row 1:
  - itemType: Hats
  - itemCategory: baseball
  - itemClass: longbill
  - itemColor: red
  - itemSize: small
  - price: 12.07
  - inventoryCount: 127
- Row 2:
  - itemType: Hats
  - itemCategory: baseball
  - itemClass: longbill
  - itemColor: red
  - itemSize: medium
  - price: 13.07
  - inventoryCount: 201
- Row 3:
  - itemType: Pants
  - itemCategory: baseball
  - itemClass: Summer
  - itemColor: red
  - itemSize: large
  - price: 14.07
  - inventoryCount: 39
- Row 4:
  - itemType: Pants
  - itemCategory: baseball
  - itemClass: Winter
  - itemColor: white
  - itemSize: large
  - price: 16.99
  - inventoryCount: 9
- Row *n*:

Copyright © 2011, 2026, Oracle and/or its affiliates.

- itemType: Coats
- itemCategory: Casual
- itemClass: Winter
- itemColor: red
- itemSize: large
- price: 247.99
- inventoryCount: 13

If you want to locate all the Hats and Pants used for baseball, using nine threads in parallel, you can retrieve all of the records as follows:

```
package kvstore.basicExample;

...
import java.util.ArrayList;
import java.util.List;
import oracle.kv.Consistency;
import oracle.kv.Direction;
import oracle.kv.table.PrimaryKey;
import oracle.kv.table.Row;
import oracle.kv.table.TableAPI;
import oracle.kv.table.TableIterator;
import oracle.kv.table.TableIteratorOptions;

...

// KVStore handle creation is omitted for brevity

...

// Construct the Table Handle
TableAPI tableH = store.getTableAPI();
Table table = tableH.getTable("myTable");


// Use multi-threading for this store iteration and limit the number
// of threads (degree of parallelism) to 9.
final int maxConcurrentRequests = 9;
final int batchResultsSize = 0;
final TableIteratorOptions tio =
   new TableIteratorOptions(Direction.UNORDERED,
                  Consistency.NONE_REQUIRED,
                  0, null,
                  maxConcurrentRequests,
                  batchResultsSize);

// Create retrieval keys
PrimaryKey myKey = table.createPrimaryKey();
myKey.put("itemType", "Hats");
myKey.put("itemCategory", "baseball");
PrimaryKey otherKey = table.createPrimaryKey();
otherKey.put("itemType", "Pants");
otherKey.put("itemCategory", "baseball");
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
List<PrimaryKey> searchKeys = new ArrayList<PrimaryKey>();

// Add the retrieval keys to the list.
searchKeys.add(myKey);
searchKeys.add(otherKey);


final TableIterator<Row> iterator = tableH.tableIterator(
                                        searchKeys.iterator(), null, tio);

// Now retrieve the records.
try {
    while (iterator.hasNext()) {
    Row row = (Row) iterator.next();
    // Do some work with the Row here
    }
} finally {
   if (iterator != null) {
   iterator.close();
   }
}
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

8
Using Data Types

Many of the types that Oracle NoSQL Database offers are easy to use. Examples of their usage has been scattered throughout this manual. However, some types are a little more complicated to use because they use container methods. This chapter describes their usage.

The types described in this chapter are: Arrays, Maps, Records, Enums, and Byte Arrays. This chapter shows how to read and write values of each of these types.

Using Arrays

Arrays are a sequence of values all of the same type.

When you declare a table field as an array, you use the `ARRAY()` statement.

To define a simple two-field table where the primary key is a UID and the second field contains array of strings, you use the following DDL statement:

```
CREATE TABLE myTable (
    uid INTEGER,
    myArray ARRAY(STRING),
    PRIMARY KEY(uid)
)
```

`DEFAULT` and `NOT NULL` constraints are not supported for arrays.

To write the array, use `Row.putArray()`, which returns an `ArrayValue` class instance. You then use `ArrayValue.put()` to write elements to the array:

```
TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myTable");

Row row = myTable.createRow();
row.put("uid", 12345);

ArrayValue av = row.putArray("myArray");
av.add("One");
av.add("Two");
av.add("Three");

tableH.put(row, null, null);
```

Note that `ArrayValue` has methods that allow you to add multiple values to the array by appending an array of values to the array. This assumes the array of values matches the array's schema. For example, the previous example could be done in the following way:

```
TableAPI tableH = kvstore.getTableAPI();
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
Table myTable = tableH.getTable("myTable");

Row row = myTable.createRow();
row.put("uid", 12345);

ArrayValue av = row.putArray("myArray");
String myStrings[] = {"One", "Two", "Three"};
av.add(myStrings);

tableH.put(row, null, null);
```

To read the array, use `Row.get().asArray()`. This returns an `ArrayValue` class instance. You can then use `ArrayValue.get()` to retrieve an element of the array from a specified index, or you can use `ArrayValue.toList()` to return the array as a Java `List`. In either case, the retrieved values are returned as a `FieldValue`, which allows you to retrieve the encapsulated value using a cast method such as `FieldValue.asString()`.

For example, to iterate over the array created in the previous example:

```
TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myTable");

/* Create a primary key for user id 12345 and get a row */
PrimaryKey key = myTable.createPrimaryKey();
key.put("uid", 12345);
Row row = tableH.get(key, null);

/* Iterate over the array, displaying each element as a string */
ArrayValue av = row.get("myArray").asArray();
for (FieldValue fv: av.toList()) {
    System.out.println(fv.asString().get()); }
```

Using Binary

You can declare a field as binary using the `BINARY` statement. You then read and write the field value using a Java byte array.

If you want to store a large binary object, then you should use the LOB APIs rather than a binary field.

Note that fixed binary should be used over the binary datatype any time you know that all the field values will be of the same size. Fixed binary is a more compact storage format because it does not need to store the size of the array. See Using Fixed Binary for information on the fixed binary datatype.

To define a simple two-field table where the primary key is a UID and the second field contains a binary field, you use the following statement:

```
CREATE TABLE myTable (
    uid INTEGER,
    myByteArray BINARY,
    PRIMARY KEY(uid)
)
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

DEFAULT and NOT NULL constraints are not supported for binary values.

To write the byte array, use Row.put().

```
TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myTable");

Row row = myTable.createRow();
row.put("uid", 12345);

String aString = "The quick brown fox.";
try {
    row.put("myByteArray", aString.getBytes("UTF-8"));
} catch (UnsupportedEncodingException uee) {
    uee.printStackTrace();
}

tableH.put(row, null, null);
```

To read the binary field, use Row.get().asBinary(). This returns a BinaryValue class instance. You can then use BinaryValue.get() to retrieve the stored byte array.

For example:

```
TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myTable");

/* Create a primary key for user id 12345 and get a row */
PrimaryKey key = myTable.createPrimaryKey();
key.put("uid", 12345);
Row row = tableH.get(key, null);

byte[] b = row.get("myByteArray").asBinary().get();
String aString = new String(b);
System.out.println("aString: " + aString);
```

Using Enums

Enumerated types are declared using the ENUM() statement. You must declare the acceptable enumeration values when you use this statement.

To define a simple two-field table where the primary key is a UID and the second field contains an enum, you use the following DDL statement:

```
CREATE TABLE myTable (
    uid INTEGER,
    myEnum ENUM (Apple,Pears,Oranges),
    PRIMARY KEY (uid)
)
```

DEFAULT and NOT NULL constraints are supported for enumerated fields. See DEFAULT for more information.

Copyright © 2011, 2026, Oracle and/or its affiliates.

To write the enum, use `Row.putEnum()`. If the enumeration value that you use with this method does not match a value defined on the `-enum-values` parameter during table definition, an `IllegalArgumentException` is thrown.

```
TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myTable");

Row row = myTable.createRow();
row.put("uid", 12345);

row.putEnum("myEnum", "Pears");

tableH.put(row, null, null);
```

To read the enum, use `Row.get().asEnum()`. This returns a `EnumValue` class instance. You can then use `EnumValue.get()` to retrieve the stored enum value's name as a string. Alternatively, you can use `EnumValue.getIndex()` to retrieve the stored value's index position.

For example:

```
TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myTable");

/* Create a primary key for user id 12345 and get a row */
PrimaryKey key = myTable.createPrimaryKey();
key.put("uid", 12345);
Row row = tableH.get(key, null);

EnumValue ev = row.get("testEnum").asEnum();
System.out.println("enum as string: " +
                   ev.get()); // returns "Pears"
System.out.println("enum index: " +
                   ev.getIndex()); // returns '1'
```

Using Fixed Binary

You can declare a fixed binary field using the `BINARY()` statement. When you do this, you must also specify the field's size in bytes. You then read and write the field value using Java byte arrays. However, if the byte array does not equal the specified size, then `IllegalArgumentException` is thrown when you attempt to write the field. Write the field value using a Java byte array.

If you want to store a large binary object, then you should use the LOB APIs rather than a binary field.

Fixed binary should be used over the binary datatype any time you know that all the field values will be of the same size. Fixed binary is a more compact storage format because it does not need to store the size of the array. See Using Binary for information on the binary datatype.

Copyright © 2011, 2026, Oracle and/or its affiliates.

To define a simple two-field table where the primary key is a UID and the second field contains a fixed binary field, you use the following DDL statement:

```
CREATE TABLE myTable (
    uid INTEGER,
    myByteArray BINARY(20),
    PRIMARY KEY (uid)
)
```

`DEFAULT` and `NOT NULL` constraints are not supported for binary values.

To write the byte array, use `Row.putFixed()`. Again, if the byte array does not match the size defined for this field, then `IllegalArgumentException` is thrown.

```
TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myTable");

Row row = myTable.createRow();
row.put("uid", 12345);

String aString = "The quick brown fox.";
try {
    row.putFixed("myByteArray", aString.getBytes("UTF-8"));
} catch (UnsupportedEncodingException uee) {
    uee.printStackTrace();
}

tableH.put(row, null, null);
```

To read the fixed binary field, use `Row.get().asFixedBinary()`. This returns a `FixedBinaryValue` class instance. You can then use `FixedBinaryValue.get()` to retrieve the stored byte array.

For example:

```
TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myTable");

/* Create a primary key for user id 12345 and get a row */
PrimaryKey key = myTable.createPrimaryKey();
key.put("uid", 12345);
Row row = tableH.get(key, null);

byte[] b = row.get("myByteArray").asFixedBinary().get();
String aString = new String(b);
System.out.println("aString: " + aString);
```

Using JSON

The JSON datatype cannot be used as part of a primary or shard key.

Copyright © 2011, 2026, Oracle and/or its affiliates.

To define a simple two-field table where the primary key is a UID and the second field contains a JSON data field, you use the following DDL statement:

```
CREATE TABLE myJsonTable (
    uid INTEGER,
    myJSON JSON,
    PRIMARY KEY (uid)
)
```

The data that you write for this datatype can be any valid JSON stored as a string. For example, all of the following are valid:

```
final String jsonNumber = "2";
final String jsonString = "\"a json string\"";
final String jsonObject_null = "{}";
final String jsonObject = "{\"a\": 1.006, \"b\": null," +
            "\"bool\" : true, \"map\": {\"m1\": 5}," +
            "\"ar\" : [1,2.7,3]}";
final String jsonNull = "null";
```

To store a JSON value in the table that we defined, above:

```
    TableAPI tableH = kvstore.getTableAPI();

    Table myJsonTable = tableH.getTable("myJsonTable");
    Row row = myTable.createRow();
    row.put("uid", 12345);
    String jsonArray="[1,5,11.1,88]";
    row.putJson("myJSON", jsonArray);
    tableH.put(row, null, null);
```

To retrieve it:

```
    TableAPI tableH = kvstore.getTableAPI();

    Table myTable = tableH.getTable("myJsonTable");
    PrimaryKey pkey = myTable.createPrimaryKey();
    pkey.put("uid", 12345);

    Row row = tableH.get(pkey, null);
    int uid = row.get("uid").asInteger().get();
    String jsonStr = row.get("myJSON").toString();

    System.out.println("uid: " + uid + " JSON: " + jsonStr);
```

Be aware that a version of `Row.putJson()` exists that allows you to use Java Readers to stream JSON data from I/O locations (such as files on disk). For example, to stream a small file from disk use `java.io.FileReader`:

```
    TableAPI tableH = kvstore.getTableAPI();

    Table myTable = tableH.getTable("myJsonTable");
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
Row row = myTable.createRow();
row.put("uid", 666);

try {
   FileReader fr = new FileReader("myJsonFile.txt");
   row.putJson("myJson", fr);
   tableH.put(row, null, null);
} catch (FileNotFoundException fnfe) {
   System.out.println("File not found: " + fnfe);
}
```

For a more complete example of using JSON data fields, see JSON By Example.

Using Maps

All map entries must be of the same type. Regardless of the type of the map's values, its keys are always strings.

The string "[]" is reserved and must not be used for key names.

When you declare a table field as a map, you use the `MAP()` statement. You must also declare the map element's data types.

To define a simple two-field table where the primary key is a UID and the second field contains a map of integers, you use the following DDL statement:

```
CREATE TABLE myTable (
    uid INTEGER,
    myMap MAP(INTEGER),
    PRIMARY KEY (uid)
)
```

`DEFAULT` and `NOT NULL` constraints are not supported for map fields.

To write the map, use `Row.putMap()`, which returns a `MapValue` class instance. You then use `MapValue.put()` to write elements to the map:

```
TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myTable");

Row row = myTable.createRow();
row.put("uid", 12345);

MapValue mv = row.putMap("myMap");
mv.put("field1", 1);
mv.put("field2", 2);
mv.put("field3", 3);

tableH.put(row, null, null);
```

To read the map, use `Row.get().asMap()`. This returns a `MapValue` class instance. You can then use `MapValue.get()` to retrieve an map value. The retrieved value is returned as a `FieldValue`, which allows you to retrieve the encapsulated value using a cast method such as `FieldValue.asInteger()`.

Copyright © 2011, 2026, Oracle and/or its affiliates.

For example, to retrieve elements from the map created in the previous example:

```
TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myTable");

/* Create a primary key for user id 12345 and get a row */
PrimaryKey key = myTable.createPrimaryKey();
key.put("uid", 12345);
Row row = tableH.get(key, null);

MapValue mv = row.get("testMap").asMap();
FieldValue fv = mv.get("field3");
System.out.println("fv: " + fv.asInteger().get());
```

Using Embedded Records

A record entry can contain fields of differing types. However, embedded records should be used only when the data is relatively static. In general, child tables provide a better solution over embedded records, especially if the child dataset is large or is likely to change in size.

Use the `RECORD()` statement to declare a table field as a record.

To define a simple two-field table where the primary key is a UID and the second field contains a record, you use the following DDL statement:

```
CREATE TABLE myTable (
    uid INTEGER,
    myRecord RECORD(firstField STRING, secondField INTEGER),
    PRIMARY KEY (uid)
)
```

`DEFAULT` and `NOT NULL` constraints are not supported for embedded record fields. However, these constraints can be applied to the individual fields in an embedded record. See Field Constraints for more information.

To write the record, use `Row.putRecord()`, which returns a `RecordValue` class instance. You then use `RecordValue.put()` to write fields to the record:

```
TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myTable");

Row row = myTable.createRow();
row.put("uid", 12345);

RecordValue rv = row.putRecord("myRecord");
rv.put("firstField", "An embedded record STRING field");
rv.put("secondField", 3388);

tableH.put(row, null, null);
```

To read the record, use `Row.get().asRecord()`. This returns a `RecordValue` class instance. You can then use `RecordValue.get()` to retrieve a field from the record. The retrieved value is

Copyright © 2011, 2026, Oracle and/or its affiliates.

returned as a `FieldValue`, which allows you to retrieve the encapsulated value using a cast method such as `FieldValue.asInteger()`.

For example, to retrieve field values from the embedded record created in the previous example:

```
TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myTable");

/* Create a primary key for user id 12345 and get a row */
PrimaryKey key = myTable.createPrimaryKey();
key.put("uid", 12345);
Row row = tableH.get(key, null);

RecordValue rv = row.get("myRecord").asRecord();
FieldValue fv = rv.get("firstField");
System.out.println("firstField: " + fv.asString().get());
fv = rv.get("secondField");
System.out.println("secondField: " + fv.asInteger().get());
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

9

Indexing Non-Scalar Data Types

We describe how to index scalar data types in Creating Indexes, and we show how to read using indexes in Reading Indexes. However, non-scalar data types (Arrays, Maps and Records) require more explanation, which we give here.

Index creation is accomplished using the `CREATE INDEX` statement. See CREATE INDEX for details on this statement.

Indexing Arrays

You can create an index on an array field (or a field of an array nested inside another array).

Be aware that indexing an array potentially results in multiple index entries for each row, which can lead to very large indexes.

To create the index, first create the table:

```
CREATE TABLE myArrayTable (
    uid INTEGER,
    testArray ARRAY(STRING),
    PRIMARY KEY(uid)
)
```

Once the table has been added to the store, create the index. Be sure to use `[]` with the field name to indicate that it is an array:

```
CREATE INDEX arrayFieldIndex on myArrayTable (testArray[])
```

In the case of arrays, the field can be indexed only if the array contains values that are of one of the other indexable types.

An index on an array is a multikey index. An index is called a multikey index if for each row of data in the table, there are multiple entries created in the index. In a multikey index there is at least one index path that uses `[]` steps. Any such index path will be called a multikey index path.

In a multikey index, for each table row, index entries are created on all the elements in arrays that are being indexed. If the evaluation returns an empty result, the special value EMPTY is used as the index entry. Any duplicate index entries are then eliminated.

To retrieve data using an index of arrays, you first retrieve the index using its name, and create an instance of `IndexKey` that you will use to perform the index lookup:

```
Index arrayIndex = myTable.getIndex("arrayFieldIndex");
IndexKey indexKey = arrayIndex.createIndexKey();
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

Next you assign the array field name and its value to the `IndexKey` that you created using the `IndexKey.put()` method:

```
indexKey.put("testArray[]", "One");
```

When you perform the index lookup, the only records that will be returned will be those which have an array with at least one item matching the value set for the `IndexKey` object. For example, if you have individual records that contain arrays like this:

```
Record 1: ["One," "Two", "Three"]
Record 2: ["Two", "Three", "One"]
Record 3: ["One", "Three", "One"]
Record 4: ["Two", "Three", "Four"]
```

and you then perform an array lookup on the array value "One", then Records 1 - 3 will be returned, but not 4.

After that, you retrieve the matching table rows, and iterate over them in the same way you would any other index type. For example:

```
TableIterator<Row> iter = tableH.tableIterator(indexKey, null, null);
System.out.println("Results for Array value 'One' : ");
try {
    while (iter.hasNext()) {
        Row rowRet = iter.next();
        int uid = rowRet.get("uid").asInteger().get();
        System.out.println("uid: " + uid);
        ArrayValue avRet = rowRet.get("testArray").asArray();
        for (FieldValue fv: avRet.toList()) {
            System.out.println(fv.asString().get());
        }
    }
} finally {
    if (iter != null) {
        iter.close();
    }
}
```

Indexing JSON Fields

You can create an index on a JSON field. To create the index, specify it as you would any other index, except that you must define the data type of the JSON field you are indexing.

Note that there are some restrictions on the data type of the JSON field that you can index. See JSON Indexes for more information.

To create the index, first create the table:

```
CREATE Table JSONPersons (
    id INTEGER,
    person JSON,
    PRIMARY KEY (id)
)
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

To create the index, you must specify the JSON field to be indexed using dot notation. Suppose your table rows look like this:

```
  "id":1,
  "person" : {
      "firstname":"David",
      "lastname":"Morrison",
      "age":25,
      "income":100000,
      "lastLogin" : "2016-10-29T18:43:59.8319",
      "address":{"street":"150 Route 2",
                 "city":"Antioch",
                 "state":"TN",
                 "zipcode" : 37013,
                 "phones":[{"type":"home", "areacode":423,
                            "number":8634379}]
                },
      "connections":[2, 3],
      "expenses":{"food":1000, "gas":180}
  }
```

Then once the table has been added to the store, you can create an index for one of the JSON fields like this:

```
CREATE INDEX idx_json_income on JSONPersons (person.income AS integer)
```

To retrieve data using a JSON index, you first retrieve the index using its name, and create an instance of `IndexKey` that you will use to perform the index lookup. The following is used to retrieve all table rows where the `person.income` field is 100000:

```
Index jsonIndex = myTable.getIndex("idx_json_income");
IndexKey indexKey = jsonIndex.createIndexKey();
indexKey.put("person.income", 100000);
```

When you perform the index lookup, the only rows returned will be those which have a JSON field with the specified field value. You then retrieve the matching table rows, and iterate over them in the same way you would any other index type. For example:

```
TableIterator<Row> iter = tableH.tableIterator(indexKey, null, null);
System.out.println("Results for person.income, value 100000: ");
try {
    while (iter.hasNext()) {
        Row rowRet = iter.next();
        int id = rowRet.get("id").asInteger().get();
        System.out.println("id: " + id);
        MapValue mapRet = rowRet.get("person").asMap();
        System.out.println("person: " + mapRet.toString());
    }
} finally {
    if (iter != null) {
        iter.close();
    }
}
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

For a more complete example of using JSON data fields, including a JSON index, see JSON By Example.

Indexing Maps

You can create an index on a map field (or a field of a map nested inside another array or map). An index created on a map field inside another array or map is a multi key index.

An index is called a multikey index if for each row of data in the table, there are multiple entries created in the index. In a multikey index, there is at least one index path that uses `.keys(), .values()`. Any such index path will be called a multikey index path. In a multikey index, for each table row, index entries are created on entries in maps that are being indexed. If the evaluation returns an empty result, the special value EMPTY is used as the index entry. Any duplicate index entries are then eliminated.

To create the index, define the map as normal. Once the map is defined for the table, there are several different ways to index it:

- Based on the map's keys without regard to the actual key values.
- Based on the map's values, without regard to the actual key used.
- By a specific map key. To do this, you specify the name of the map field *and* the name of a map key using dot notation. If the map key is ever created using your client code, then it will be indexed.
- Based on the map's key and value without identifying a specific value (such as is required by the previous option in this list).

Indexing by Map Keys

You can create indexes based on a map's keys without regard to the corresponding values.

Be aware that creating an index like this can potentially result in multiple index entries for each row, which can lead to very large indexes.

First create the table:

```
CREATE TABLE myMapTable (
    uid INTEGER,
    testMap MAP(INTEGER),
    PRIMARY KEY(uid)
)
```

Once the table has been added to the store, create the index using the `.keys()` path step:

```
CREATE INDEX mapKeyIndex on myMapTable (testMap.keys())
```

Data is retrieved if the table row contains the identified map with the identified key. So, for example, if you create a series of table rows like this:

```
TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myMapTable");

Row row = myTable.createRow();
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
row.put("uid", 12345);
MapValue mv = row.putMap("testMap");
mv.put("field1", 1);
mv.put("field2", 2);
mv.put("field3", 3);
tableH.put(row, null, null);

row = myTable.createRow();
row.put("uid", 12);
mv = row.putMap("testMap");
mv.put("field1", 1);
mv.put("field2", 2);
tableH.put(row, null, null);

row = myTable.createRow();
row.put("uid", 666);
mv = row.putMap("testMap");
mv.put("field1", 1);
mv.put("field3", 4);
tableH.put(row, null, null);
```

then you can retrieve any table rows that contain the map with any key currently in use by the map. For example, "field3".

To retrieve data using a map index, you first retrieve the index using its name, and create an instance of `IndexKey` that you will use to perform the index lookup:

```
Index mapIndex = myTable.getIndex("mapKeyIndex");
IndexKey indexKey = mapIndex.createIndexKey();
```

Next, you populate the `IndexKey` instance with the field name that you want to retrieve. Use the `keys()` path step to indicate that you want to retrieve using the field name without regard for the field value. When you perform the index lookup, the only records that will be returned will be those which have a map with the specified key name:

```
indexKey.put("testMap.keys()", "field3");
```

After that, you retrieve the matching table rows, and iterate over them in the same way you would any other index type. For example:

```
TableIterator<Row> iter = tableH.tableIterator(indexKey, null, null);
System.out.println("Results for testMap field3: ");
try {
    while (iter.hasNext()) {
        Row rowRet = iter.next();
        int uid = rowRet.get("uid").asInteger().get();
        System.out.println("uid: " + uid);
        MapValue mapRet = rowRet.get("testMap").asMap();
        System.out.println("testMap: " + mapRet.toString());
    }
} finally {
    if (iter != null) {
        iter.close();
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
    }
}
```

Indexing by Map Values

You can create indexes based on the values contained in a map without regard to the keys in use.

Be aware that creating an index like this can potentially result in multiple index entries for each row, which can lead to very large indexes.

First create the table:

```
CREATE TABLE myMapTable (
    uid INTEGER,
    testMap MAP(INTEGER),
    PRIMARY KEY(uid)
)
```

Once the table has been added to the store, create the index using the `.values()` path step:

```
CREATE INDEX mapElementIndex on myMapTable (testMap.values())
```

Data is retrieved if the table row contains the identified map with the identified value. So, for example, if you create a series of table rows like this:

```
TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myMapTable");

Row row = myTable.createRow();
row.put("uid", 12345);
MapValue mv = row.putMap("testMap");
mv.put("field1", 1);
mv.put("field2", 2);
mv.put("field3", 3);
tableH.put(row, null, null);

row = myTable.createRow();
row.put("uid", 12);
mv = row.putMap("testMap");
mv.put("field1", 1);
mv.put("field2", 2);
tableH.put(row, null, null);

row = myTable.createRow();
row.put("uid", 666);
mv = row.putMap("testMap");
mv.put("field1", 1);
mv.put("field3", 4);
tableH.put(row, null, null);
```

then you can retrieve any table rows that contain the map with any value currently in use by the map. For example, a value of "2".

Copyright © 2011, 2026, Oracle and/or its affiliates.

To retrieve data using a map index, you first retrieve the index using its name, and create an instance of `IndexKey` that you will use to perform the index lookup:

```
Index mapIndex = myTable.getIndex("mapElementIndex");
IndexKey indexKey = mapIndex.createIndexKey();
```

Next, you populate the `IndexKey` instance with the field value (2) that you want to retrieve. Use the `values()` path step with the field name to indicate that you want to retrieve entries based on the value only. When you perform the index lookup, the only records that will be returned will be those which have a map with a value of 2.

```
indexKey.put("testMap.values()", 2);
```

After that, you retrieve the matching table rows, and iterate over them in the same way you would any other index type. For example:

```
TableIterator<Row> iter = tableH.tableIterator(indexKey, null, null);
System.out.println("Results for testMap value 2: ");
try {
    while (iter.hasNext()) {
        Row rowRet = iter.next();
        int uid = rowRet.get("uid").asInteger().get();
        System.out.println("uid: " + uid);
        MapValue mapRet = rowRet.get("testMap").asMap();
        System.out.println("testMap: " + mapRet.toString());
    }
} finally {
    if (iter != null) {
        iter.close();
    }
}
```

Indexing by a Specific Map Key Name

You can create an index based on a specified map key name. Any map entries containing the specified key name are indexed. This can create a small and very efficient index because the index does not contain every key/value pair contained by the map fields. Instead, it just contains those map entries using the identified key, which results in at most a single index entry per row.

To create the index, first create the table:

```
CREATE TABLE myMapTable (
    uid INTEGER,
    testMap MAP(INTEGER),
    PRIMARY KEY(uid)
)
```

Once the table has been added to the store, create the index by specifying the key name you want indexed using dot notation. In this example, we will index the key name of "field3":

```
CREATE INDEX mapField3Index on myMapTable (testMap.field3)
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

Data is retrieved if the table row contains the identified map with the indexed key and a specified value. So, for example, if you create a series of table rows like this:

```
TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myMapTable");

Row row = myTable.createRow();
row.put("uid", 12345);
MapValue mv = row.putMap("testMap");
mv.put("field1", 1);
mv.put("field2", 2);
mv.put("field3", 3);
tableH.put(row, null, null);

row = myTable.createRow();
row.put("uid", 12);
mv = row.putMap("testMap");
mv.put("field1", 1);
mv.put("field2", 2);
tableH.put(row, null, null);

row = myTable.createRow();
row.put("uid", 666);
mv = row.putMap("testMap");
mv.put("field1", 1);
mv.put("field3", 4);
tableH.put(row, null, null);
```

then you can retrieve any table rows that contain the map with key "field3" (because that is what you indexed) when "field3" maps to a specified value — such as "3". If you try to do an index lookup on, for example, "field2" then that will fail because you did not index "field2".

To retrieve data using a map index, you first retrieve the index using its name and create an instance of `IndexKey` that you will use to perform the index lookup:

```
Index mapIndex = myTable.getIndex("mapField3Index");
IndexKey indexKey = mapIndex.createIndexKey();
```

Then you populate the map field name (using dot notation) and the desired value using `IndexKey.put()`. When you perform the index lookup, the only records that will be returned will be those which have a map with the matching key name and corresponding value.

```
indexKey.put("testMap.field3", 3);
```

After that, you retrieve the matching table rows, and iterate over them in the same way you would any other index type. For example:

```
TableIterator<Row> iter = tableH.tableIterator(indexKey, null, null);
System.out.println("Results for testMap field3, value 3: ");
try {
    while (iter.hasNext()) {
        Row rowRet = iter.next();
        int uid = rowRet.get("uid").asInteger().get();
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
            System.out.println("uid: " + uid);
            MapValue mapRet = rowRet.get("testMap").asMap();
            System.out.println("testMap: " + mapRet.toString());
        }
    } finally {
        if (iter != null) {
            iter.close();
        }
    }
```

Indexing by Map Key and Value

In the previous section, we showed how to create a map index by specifying a pre-determined key name. This allows you to perform map index look ups by providing both key and value, but the index lookup will only be successful if the specified key is the key that you indexed.

You can do the same thing in a generic way by indexing every key/value pair in your map. The result is a more flexible index, but also an index that is potentially much larger than the previously described method. It is likely to result in multiple index entries per row.

To create an index based on every key/value pair used by the map field, first create the table:

```
CREATE TABLE myMapTable (
    uid INTEGER,
    testMap MAP(INTEGER),
    PRIMARY KEY(uid)
)
```

Once the table has been added to the store, create the index by using the `.keys()` and `.values()` path steps:

```
CREATE INDEX mapKeyValueIndex on myMapTable
(testMap.keys(),testMap.values())
```

Data is retrieved if the table row contains the identified map with the identified key and the identified value. So, for example, if you create a series of table rows like this:

```
TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myMapTable");

Row row = myTable.createRow();
row.put("uid", 12345);
MapValue mv = row.putMap("testMap");
mv.put("field1", 1);
mv.put("field2", 2);
mv.put("field3", 3);
tableH.put(row, null, null);

row = myTable.createRow();
row.put("uid", 12);
mv = row.putMap("testMap");
mv.put("field1", 1);
mv.put("field2", 2);
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
tableH.put(row, null, null);

row = myTable.createRow();
row.put("uid", 666);
mv = row.putMap("testMap");
mv.put("field1", 1);
mv.put("field3", 4);
tableH.put(row, null, null);
```

then you can retrieve any table rows that contain the map with specified key/value pairs — for example, key "field3" and value "3".

To retrieve data using a map index, you first retrieve the index using its name and create an instance of `IndexKey` that you will use to perform the index lookup:

```
Index mapIndex = myTable.getIndex("mapKeyValueIndex");
IndexKey indexKey = mapIndex.createIndexKey();
```

Next, you populate the `IndexKey` class instance with the field name and value you want to retrieve. In this case, you must specify two sets of information, using two calls to `IndexKey.put()`:

- The name of the field. Here, use the `keys()` path step with the field name.
- The field value you want to retrieve. Here, use the `values()` path step the field name.

For example:

```
indexKey.put("testMap.keys()", "field3");
indexKey.put("testMap.values()", 3);
```

When you perform the index lookup, the only records that will be returned will be those which have a map with the matching key/value pair. Once you have performed the index lookup, you retrieve the matching table rows, and iterate over them in the same way you would any other index type. For example:

```
TableIterator<Row> iter = tableH.tableIterator(indexKey, null, null);
System.out.println("Results for testMap field3, value 3: ");
try {
    while (iter.hasNext()) {
        Row rowRet = iter.next();
        int uid = rowRet.get("uid").asInteger().get();
        System.out.println("uid: " + uid);
        MapValue mapRet = rowRet.get("testMap").asMap();
        System.out.println("testMap: " + mapRet.toString());
    }
} finally {
    if (iter != null) {
        iter.close();
    }
}
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

Indexing Embedded Records

You can create an index on an embedded record field so long as the record field contains scalar data. To create the index, define the record as normal. To index the field, you specify the name of the embedded record *and* the name of the field using dot notation.

To create the index, first create the table:

```
CREATE Table myRecordTable (
    uid INTEGER,
    myRecord RECORD (firstField STRING, secondField INTEGER),
    PRIMARY KEY (uid)
)
```

Once the table has been added to the store, create the index:

```
CREATE INDEX recordFieldIndex on myRecordTable (myRecord.secondField)
```

Data is retrieved if the table row contains the identified record field with the specified value. So, for example, if you create a series of table rows like this:

```
TableAPI tableH = kvstore.getTableAPI();
Table myTable = tableH.getTable("myRecordTable");

Row row = myTable.createRow();
row.put("uid", 12345);
RecordValue rv = row.putRecord("myRecord");
rv.put("firstField", "String field for 12345");
rv.put("secondField", 3388);
tableH.put(row, null, null);

row = myTable.createRow();
row.put("uid", 345);
rv = row.putRecord("myRecord");
rv.put("firstField", "String field for 345");
rv.put("secondField", 3388);
tableH.put(row, null, null);

row = myTable.createRow();
row.put("uid", 111);
rv = row.putRecord("myRecord");
rv.put("firstField", "String field for 111");
rv.put("secondField", 12);
tableH.put(row, null, null);
```

then you can retrieve any table rows that contain the embedded record where "secondField" is set to a specified value. (The embedded record index that we specified, above, indexed myRecord.secondField.)

Copyright © 2011, 2026, Oracle and/or its affiliates.

To retrieve data using a record index, you first retrieve the index using its name, and create an instance of `IndexKey` that you will use to perform the index lookup:

```
Index recordIndex = myTable.getIndex("recordFieldIndex");
IndexKey indexKey = recordIndex.createIndexKey();
indexKey.put(“myRecord.secondField", 3388);
```

When you perform the index lookup, the only records returned will be those which have an embedded record with the specified field and field value. You then retrieve the matching table rows, and iterate over them in the same way you would any other index type. For example:

```
TableIterator<Row> iter = tableH.tableIterator(indexKey, null, null);
System.out.println("Results for testRecord.secondField, value 3388: ");
try {
    while (iter.hasNext()) {
        Row rowRet = iter.next();
        int uid = rowRet.get("uid").asInteger().get();
        System.out.println("uid: " + uid);
        RecordValue recordRet = rowRet.get("myRecord").asRecord();
        System.out.println("myRecord: " + recordRet.toString());
    }
} finally {
    if (iter != null) {
        iter.close();
    }
}
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

10

Using Row Versions

When a row is initially inserted in the store, and each time it is updated, it is assigned a unique version token. The version is always returned by the method that wrote to the store (for example, `TableAPI.put()`). The version information is also returned by methods that retrieve rows from the store.

There are two reasons why versions might be important.

1. When an update or delete is to be performed, it may be important to perform the operation only if the row's value has not changed. This is particularly interesting in an application where there can be multiple threads or processes simultaneously operating on the row. In this case, read the row, examining its version when you do so. You can then perform a put operation, but only allow the put to proceed if the version has not changed (this is often referred to as a *Compare and Set* (CAS) or *Read, Modify, Write* (RMW) operation). You use `TableAPI.putIfVersion()` or `TableAPI.deleteIfVersion()` to guarantee this.
2. When a client reads data that was previously written, it may be important to ensure that the Oracle NoSQL Database node servicing the read operation has been updated with the information previously written. This can be accomplished by passing the version of the previously written data as a consistency parameter to the read operation. For more information on using consistency, see Consistency Guarantees.

Versions are managed using the `Version` class. In some situations, it is returned as part of another encapsulating class, such as the `Row` class.

The following code fragment retrieves a row, and then writes that row back to the store only if the version has not changed:

```
package kvstore.basicExample;

...

import oracle.kv.Version;
import oracle.kv.KVStore;
import oracle.kv.table.Index;
import oracle.kv.table.IndexKey;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;
import oracle.kv.table.TableIterator;

...

// Retrieve the row. Note that we do not show the creation of
// the kvstore handle here.

TableAPI tableH = kvstore.getTableAPI();
Table myTable = tableH.getTable("myTable");

// Construct the IndexKey. The name we gave our index when
// we created it was 'DoB'.
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
Index dobIdx = myTable.getIndex("DoB");
IndexKey dobIdxKey = dobIdx.createIndexKey();

TableIterator<Row> iter =
    tableH.tableIterator(dobIdxKey, null, null);

while (iter.hasNext()) {
    Row aRow = iter.next();

    // Retrieve the row's version information
    Version rowVersion = aRow.getVersion();

    //////////////////////////
    // Do work on the row here
    //////////////////////////

    // Put if the version is correct. Notice that here we examine
    // the return code. If it is null, that means that the put was
    // unsuccessful, probably because the row was changed elsewhere.

    Version newVersion =
        tableH.putIfVersion(row, rowVersion, null, null);
    if (newVersion == null) {
        // Unsuccessful. Someone else probably modified the record.
    }
}
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

11

Consistency Guarantees

A Oracle NoSQL Database store is built from some number of computers (generically referred to as *nodes*) that are working together using a network. All data in your store is first written to a master node. The master node then copies that data to other nodes in the store. Nodes which are not master nodes are referred to as *replicas.*

Because of the nature of distributed systems, there is a possibility that, at any given moment, a write operation that was performed on the master node will not yet have been performed on some other node in the store.

Consistency, then, is the policy describing whether it is possible for a row on Node A to be different from the same row on Node B.

When there is a high likelihood that a row stored on one node is identical to the same row stored on another node, we say that we have a *high consistency guarantee*. Likewise, a *low consistency guarantee* means that there is a good possibility that a row on one node differs in some way from the same row stored on another node.

You can control how high you want your consistency guarantee to be. Note that the trade-off in setting a high consistency guarantee is that your store's read performance might not be as high as if you use a low consistency guarantee.

There are several different forms of consistency guarantees that you can use. They are described in the following sections.

Note that by default, Oracle NoSQL Database uses the lowest possible consistency possible.

Specifying Consistency Policies

To specify a consistency policy, you use one of the static instances of the `Consistency` class, or one of its nested classes.

Once you have selected a consistency policy, you can put it to use in one of two ways. First, you can use it to define a default consistency policy using the `KVStoreConfig.setConsistency()` method. Specifying a consistency policy in this way means that all store operations will use that policy, unless they are overridden on an operation by operation basis.

The second way to use a consistency policy is to override the default policy using a `ReadOption` class instance you provide to the `TableAPI` method that you are using to perform the store read operation.

The following example shows how to set a default consistency policy for the store. We will show the per-operation method of specifying consistency policies in the following sections.

```
package kvstore.basicExample;

import oracle.kv.Consistency;
import oracle.kv.KVStore;
import oracle.kv.KVStoreConfig;
import oracle.kv.KVStoreFactory;
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
...

KVStoreConfig kconfig = new KVStoreConfig("exampleStore",
    "node1.example.org:5088, node2.example.org:4129");

kconfig.setConsistency(Consistency.NONE_REQUIRED);

KVStore kvstore = KVStoreFactory.getStore(kconfig);
```

Using Simple Consistency

You can use static instances of the `Consistency` base class to specify certain rigid consistency guarantees. There are two such instances that you can use:

1. `Consistency.ABSOLUTE`

   Requires that the operation be serviced at the master node. In this way, the row(s) will always be consistent with the master.

   This is the strongest possible consistency guarantee that you can require, but it comes at the cost of servicing all read and write requests at the master node. If you direct all your traffic to the master node (which is just one machine for each partition), then you will not be distributing your read operations across your replicas. You also will slow your write operations because your master will be busy servicing read requests. For this reason, you should use this consistency guarantee sparingly.

2. `Consistency.NONE_REQUIRED`

   Allows the store operation to proceed regardless of the state of the replica relative to the master. This is the most relaxed consistency guarantee that you can require. It allows for the maximum possible store performance, but at the high possibility that your application will be operating on stale or out-of-date information.

For example, suppose you are performing a critical read operation that you know must absolutely have the most up-to-date data. Then do this:

```
package kvstore.basicExample;

import oracle.kv.Consistency;
import oracle.kv.ConsistencyException;
import oracle.kv.KVStore;
import oracle.kv.table.PrimaryKey;
import oracle.kv.table.ReadOptions;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();

// The name you give to getTable() must be identical
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
// to the name that you gave the table when you created
// the table using the CREATE TABLE DDL statement.
Table myTable = tableH.getTable("myTable");

// Construct the PrimaryKey.
PrimaryKey key = myTable.createPrimaryKey();
key.put("item", "Bolts");

// Create the ReadOption with our Consistency policy
ReadOptions ro = new ReadOptions(Consistency.ABSOLUTE,
                                 0,     // Timeout parameter.
                                        // 0 means use the default.
                                 null); // Timeout units. Null because
                                        // the Timeout is 0.


// Retrieve the row. This performs a store read operation.
// Exception handling is skipped for this trivial example.
try {
    Row row = tableH.get(key, ro);
} catch (ConsistencyException ce) {
    // The consistency guarantee was not met
}
```

Using Time-Based Consistency

A time-based consistency policy describes the amount of time that a replica node is allowed to lag behind the master node. If the replica's data is more than the specified amount of time out-of-date relative to the master, then a `ConsistencyException` is thrown. In that event, you can either abandon the operation, retry it immediately, or pause and then retry it.

In order for this type of a consistency policy to be effective, the clocks on all the nodes in the store must be synchronized using a protocol such as NTP.

In order to specify a time-based consistency policy, you use the `Consistency.Time` class. The constructor for this class requires the following information:

- `permissibleLag`

  A `long` that describes the number of `TimeUnits` the replica is allowed to lag behind the master.

- `permissibleLagUnits`

  A `TimeUnit` that identifies the units used by `permissibleLag`. For example: `TimeUnit.MILLISECONDS`.

- `timeout`

  A `long` that describes how long the replica is permitted to wait in an attempt to meet the permissible lag limit. That is, if the replica cannot immediately meet the permissible lag requirement, then it will wait this amount of time to see if it is updated with the required data from the master. If the replica cannot meet the permissible lag requirement within the timeout period, a `ConsistencyException` is thrown.

- `timeoutUnit`

  A `TimeUnit` that identifies the units used by `timeout`. For example: `TimeUnit.SECONDS`.

Copyright © 2011, 2026, Oracle and/or its affiliates.

The following sets a default time-based consistency policy of 2 seconds. The timeout is 4 seconds.

```
package kvstore.basicExample;

import oracle.kv.Consistency;
import oracle.kv.KVStore;
import oracle.kv.KVStoreConfig;
import oracle.kv.KVStoreFactory;

import java.util.concurrent.TimeUnit;

...

KVStoreConfig kconfig = new KVStoreConfig("exampleStore",
    "node1.example.org:5088, node2.example.org:4129");

Consistency.Time cpolicy =
        new Consistency.Time(2, TimeUnit.SECONDS,
                             4, TimeUnit.SECONDS);
kconfig.setConsistency(cpolicy);

KVStore kvstore = KVStoreFactory.getStore(kconfig);
```

Using Version-Based Consistency

Version-based consistency is used on a per-operation basis. It ensures that a read performed on a replica is at least as current as some previous write performed on the master.

An example of how this might be used is a web application that collects some information from a customer (such as her name). It then customizes all subsequent pages presented to the customer with her name. The storage of the customer's name is a write operation that can only be performed by the master node, while subsequent page creation is performed as a read-only operation that can occur at any node in the store.

Use of this consistency policy might require that version information be transferred between processes in your application.

To create a version-based consistency policy, use the `Consistency.Version` class. When you do this, you must provide the following information:

- `version`

  The `Version` that the read must match. The value returned is either equal or newer than the version specified in the policy.

- `timeout`

  A `long` that describes how long the replica is permitted to wait in an attempt to meet the version requirement. That is, if the replica cannot immediately meet the version requirement, then it will wait this amount of time to see if it is updated with the required data from the master. If the replica cannot meet the requirement within the timeout period, a `ConsistencyException` is thrown.

- `timeoutUnit`

  A `TimeUnit` that identifies the units used by `timeout`. For example: `TimeUnit.SECONDS`.

Copyright © 2011, 2026, Oracle and/or its affiliates.

For example, the following code performs a store write, collects the version information, then uses it to construct a version-based consistency policy.

```
package kvstore.basicExample;

import oracle.kv.KVStore;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;
import oracle.kv.Version;

...

// KVStore handle creation is omitted for brevity

...

TableAPI tableH = kvstore.getTableAPI();
Table myTable = tableH.getTable("myTable");

// Get a Row instance
Row row = myTable.createRow();

// Now put all of the cells in the row.

row.put("item", "Bolts");
row.put("count1", 5);
row.put("count2", 23);
row.put("percentage", 0.2173913);

// Now write the table to the store, capturing the
// Version information as we do.

Version matchVersion = tableH.put(row, null, null);

Version matchVersion = kvstore.put(myKey, myValue);
```

At some other point in this application's code, or perhaps in another application entirely, we use the `matchVersion` captured above to create a version-based consistency policy.

```
package kvstore.basicExample;

import oracle.kv.Consistency;
import oracle.kv.ConsistencyException;
import oracle.kv.KVStore;
import oracle.kv.table.PrimaryKey;
import oracle.kv.table.ReadOptions;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;

import java.util.concurrent.TimeUnit;

...
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
// KVStore handle creation is omitted for brevity

...

// Construct the PrimaryKey.

PrimaryKey key = myTable.createPrimaryKey();
key.put("item", "Bolts");

// Create the consistency policy, using the
// Version object we captured, above.
Consistency.Version versionConsistency =
        new Consistency.Version(matchVersion,
                                200,
                                TimeUnit.NANOSECONDS);

// Create a ReadOptions using our new consistency policy.
ReadOptions ro = new ReadOptions(versionConsistency, 0, null);

// Now perform the read.
try {

    Row row = tableH.get(key, ro);

    // Do work with the row here
} catch (ConsistencyException ce) {
    // The consistency guarantee was not met
}
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

12

Durability Guarantees

Writes are performed in the Oracle NoSQL Database store by performing the write operation (be it a creation, update, or delete operation) on a master node. As a part of performing the write operation, the master node will usually make sure that the operation has made it to stable storage before considering the operation complete.

The master node will also transmit the write operation to the replica nodes in its shard. It is possible to ask the master node to wait for acknowledgments from its replicas before considering the operation complete.

> **Note**
>
> If your store is configured such that secondary zones are in use, then write acknowledgements are never required for the replicas in the secondary zones. That is, write acknowledgements are only returned by replicas in primary zones. For more information on zones, see *Administrator's Guide*.

The replicas, in turn, will not acknowledge the write operation until they have applied the operation to their own database.

A *durability guarantee,* then, is a policy which describes how strongly persistent your data is in the event of some kind of catastrophic failure within the store. (Examples of a catastrophic failure are power outages, disk crashes, physical memory corruption, or even fatal application programming errors.)

A high durability guarantee means that there is a very high probability that the write operation will be retained in the event of a catastrophic failure. A low durability guarantee means that the write is very unlikely to be retained in the event of a catastrophic failure.

The higher your durability guarantee, the slower your write-throughput will be in the store. This is because a high durability guarantee requires a great deal of disk and network activity.

Usually you want some kind of a durability guarantee, although if you have highly transient data that changes from run-time to run-time, you might want the lowest possible durability guarantee for that data.

Durability guarantees include two types of information: acknowledgment guarantees and synchronization guarantees. These two types of guarantees are described in the next sections. We then show how to set a durability guarantee.

Note that by default, Oracle NoSQL Database uses a low durability guarantee.

Setting Acknowledgment-Based Durability Policies

Whenever a master node performs a write operation (create, update or delete), it must send that operation to its various replica nodes. The replica nodes then apply the write operation(s) to their local databases so that the replicas are consistent relative to the master node.

Copyright © 2011, 2026, Oracle and/or its affiliates.

Upon successfully applying write operations to their local databases, replicas in primary zones send an *acknowledgment message* back to the master node. This message simply says that the write operation was received and successfully applied to the replica's local database. Replicas in secondary zones do not send these acknowledgement messages.

> **Note**
>
> The exception to this are replicas in secondary zones, which will never acknowledge write operations.

An acknowledgment-based durability policy describes whether the master node will wait for these acknowledgments before considering the write operation to have completed successfully. You can require the master node to not wait for acknowledgments, or to wait for acknowledgments from a simple majority of replica nodes in primary zones, or to wait for acknowledgments from all replica nodes in primary zones.

The more acknowledgments the master requires, the slower its write performance will be. Waiting for acknowledgments means waiting for a write message to travel from the master to the replicas, then for the write operation to be performed at the replica (this may mean disk I/O), then for an acknowledgment message to travel from the replica back to the master. From a computer application's point of view, this can all take a long time.

When setting an acknowledgment-based durability policy, you can require acknowledgment from:

- All replicas. That is, all of the replica nodes in the shard that reside in a primary zone. Remember that your store has more than one shard, so the master node is not waiting for acknowledgments from every machine in the store.
- No replicas. In this case, the master returns with normal status from the write operation as soon as it has met its synchronization-based durability policy. These are described in the next section.
- A simple majority of replicas in primary zones. That is, if the shard has 5 replica nodes residing in primary zones, then the master will wait for acknowledgments from 3 nodes.

Setting Synchronization-Based Durability Policies

Whenever a node performs a write operation, the node must know whether it should wait for the data to be written to stable storage before successfully returning from the operation.

As a part of performing a write operation, the data modification is first made to an in-memory cache. It is then written to the filesystem's data buffers. And, finally, the contents of the data buffers are synchronized to stable storage (typically, a hard drive).

You can control how much of this process the master node will wait to complete before it returns from the write operation with a normal status. There are three different levels of synchronization durability that you can require:

- NO_SYNC

  The data is written to the host's in-memory cache, but the master node does not wait for the data to be written to the file system's data buffers, or for the data to be physically transferred to stable storage. This is the fastest, but least durable, synchronization policy.

- WRITE_NO_SYNC

Copyright © 2011, 2026, Oracle and/or its affiliates.

The data is written to the in-memory cache, and then written to the file system's data buffers, but the data is not necessarily transferred to stable storage before the operation completes normally.

- SYNC

  The data is written to the in-memory cache, then transferred to the file system's data buffers, and then synchronized to stable storage before the write operation completes normally. This is the slowest, but most durable, synchronization policy.

Notice that in all cases, the data is eventually written to stable storage (assuming some failure does not occur to prevent it). The only question is, how much of this process will be completed before the write operation returns and your application can proceed to its next operation.

See the next section for an example of setting durability policies.

Setting Durability Guarantees

To set a durability guarantee, use the `Durability` class. When you do this, you must provide three pieces of information:

- The acknowledgment policy.
- A synchronization policy at the master node.
- A synchronization policy at the replica nodes.

The combination of policies that you use is driven by how sensitive your application might be to potential data loss, and by your write performance requirements.

For example, the fastest possible write performance can be achieved through a durability policy that requires:

- No acknowledgments.
- NO_SYNC at the master.
- NO_SYNC at the replicas.

However, this durability policy also leaves your data with the greatest risk of loss due to application or machine failure between the time the operation returns and the time when the data is written to stable storage.

On the other hand, if you want the highest possible durability guarantee, you can use:

- All replicas must acknowledge the write operation.
- SYNC at the master.
- SYNC at the replicas.

Of course, this also results in the slowest possible write performance.

Most commonly, durability policies attempt to strike a balance between write performance and data durability guarantees. For example:

- Simple majority (> 50%) of replicas must acknowledge the write.
- SYNC at the master.
- NO_SYNC at the replicas.

Note that you can set a default durability policy for your `KVStore` handle, but you can also override the policy on a per-operation basis for those situations where some of your data need not be as durable (or needs to be MORE durable) than the default.

Copyright © 2011, 2026, Oracle and/or its affiliates.

For example, suppose you want an intermediate durability policy for most of your data, but sometimes you have transient or easily re-created data whose durability really is not very important. Then you would do something like this:

First, set the default durability policy for the `KVStore` handle:

```
package kvstore.basicExample;

import oracle.kv.Durability;
import oracle.kv.KVStore;
import oracle.kv.KVStoreConfig;
import oracle.kv.KVStoreFactory;

...

KVStoreConfig kconfig = new KVStoreConfig("exampleStore",
    "node1.example.org:5088, node2.example.org:4129");

Durability defaultDurability =
    new Durability(Durability.SyncPolicy.SYNC,    // Master sync
                   Durability.SyncPolicy.NO_SYNC, // Replica sync
                   Durability.ReplicaAckPolicy.SIMPLE_MAJORITY);
kconfig.setDurability(defaultDurability);

KVStore kvstore = KVStoreFactory.getStore(kconfig);
```

In another part of your code, for some unusual write operations, you might then want to relax the durability guarantee so as to speed up the write performance for those specific write operations:

```
package kvstore.basicExample;

...

import oracle.kv.Durability;
import oracle.kv.DurabilityException;
import oracle.kv.KVStore;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;

...

TableAPI tableH = kvstore.getTableAPI();

// The name you give to getTable() must be identical
// to the name that you gave the table when you created
// the table using the CREATE TABLE DDL statement.
Table myTable = tableH.getTable("myTable");

// Get a Row instance
Row row = myTable.createRow();

// Now put all of the cells in the row.

row.put("item", "Bolts");
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
row.put("description", "Hex head, stainless");
row.put("count", 5);
row.put("percentage", 0.2173913);

// Construct a durability policy
Durability durability =
    new Durability(Durability.SyncPolicy.NO_SYNC, // Master sync
                   Durability.SyncPolicy.NO_SYNC, // Replica sync
                   Durability.ReplicaAckPolicy.NONE);

// Construct a WriteOptions object using the durability policy.
WriteOptions wo = new WriteOptions(durability, 0, null);

// Now write the table to the store using the durability policy
// defined, above.
tableH.put(row, null, wo);
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

13

Executing a Sequence of Operations

You can execute a sequence of write operations as a single atomic unit so long as all the rows that you are operating upon share the same shard key. By *atomic unit*, we mean all of the operations will execute successfully, or none of them will.

Also, the sequence is performed in isolation. This means that if you have a thread running a particularly long sequence, then another thread cannot intrude on the data in use by the sequence. The second thread will not be able to see any of the modifications made by the long-running sequence until the sequence is complete. The second thread also will not be able to modify any of the data in use by the long-running sequence.

Be aware that sequences only support write operations. You can perform puts and deletes, but you cannot retrieve data when using sequences.

When using a sequence of operations:

- All of the keys in use by the sequence must share the same shard key.
- Operations are placed into a list, but the operations are not necessarily executed in the order that they appear in the list. Instead, they are executed in an internally defined sequence that prevents deadlocks.

The rest of this chapter shows how to use `TableOperationFactory` and `TableAPI.execute()` to create and run a sequence of operations.

Sequence Errors

If any operation within the sequence experiences an error, then the entire operation is aborted. In this case, your data is left in the same state it would have been in if the sequence had never been run at all — no matter how much of the sequence was run before the error occurred.

Fundamentally, there are two reasons why a sequence might abort:

1. An internal operation results in an exception that is considered a fault. For example, the operation throws a `DurabilityException`. Also, if there is an internal failure due to message delivery or a networking error.
2. An individual operation returns normally but is unsuccessful as defined by the particular operation. (For example, you attempt to delete a row that does not exist). If this occurs AND you specified `true` for the `abortIfUnsuccessful` parameter when the operation was created using TableOperationFactory, then an `OperationExecutionException` is thrown. This exception contains information about the failed operation.

Creating a Sequence

You create a sequence by using the `TableOperationFactory` class to create `TableOperation` class instances, each of which represents exactly one operation in the store. You obtain an instance of `TableOperationFactory` by using `TableAPI.getTableOperationFactory()`.

Copyright © 2011, 2026, Oracle and/or its affiliates.

For example, suppose you are using a table defined like this:

```
CREATE TABLE myTable (
    itemType STRING,
    itemCategory STRING,
    itemClass STRING,
    itemColor STRING,
    itemSize STRING,
    price FLOAT,
    inventoryCount INTEGER,
    PRIMARY KEY (SHARD(itemType, itemCategory, itemClass), itemColor,
    itemSize)
)
```

With tables containing data like this:

- Row 1:
  - itemType: Hats
  - itemCategory: baseball
  - itemClass: longbill
  - itemColor: red
  - itemSize: small
  - price: 12.07
  - inventoryCount: 127
- Row 2:
  - itemType: Hats
  - itemCategory: baseball
  - itemClass: longbill
  - itemColor: red
  - itemSize: medium
  - price: 13.07
  - inventoryCount: 201
- Row 3:
  - itemType: Hats
  - itemCategory: baseball
  - itemClass: longbill
  - itemColor: red
  - itemSize: large
  - price: 14.07
  - inventoryCount: 39

And further suppose that this table has rows that require an update (such as a price and inventory refresh), and you want the update to occur in such a fashion as to ensure it is performed consistently for all the rows.

Copyright © 2011, 2026, Oracle and/or its affiliates.

Then you can create a sequence in the following way:

```
package kvstore.basicExample;

import java.util.ArrayList;

import oracle.kv.KVStore;
import oracle.kv.KVStoreConfig;
import oracle.kv.KVStoreFactory;

import oracle.kv.DurabilityException;
import oracle.kv.FaultException;
import oracle.kv.OperationExecutionException;
import oracle.kv.RequestTimeoutException;

import oracle.kv.table.PrimaryKey;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;
import oracle.kv.table.TableOperationFactory;
import oracle.kv.table.TableOperation;

...

// kvstore handle creation omitted.

...

TableAPI tableH = kvstore.getTableAPI();

Table myTable = tableH.getTable("myTable");

// We use TableOperationFactory to create items for our
// sequence.
TableOperationFactory tof = tableH.getTableOperationFactory();

// This ArrayList is used to contain each item in our sequence.
ArrayList<TableOperation> opList = new ArrayList<TableOperation>();

// Update each row, adding each to the opList as we do.
Row row = myTable.createRow();
row.put("itemType", "Hats");
row.put("itemCategory", "baseball");
row.put("itemClass", "longbill");
row.put("itemColor", "red");
row.put("itemSize", "small");
row.put("price", new Float(13.07));
row.put("inventoryCount", 107);
opList.add(tof.createPut(row, null, true));

row = myTable.createRow();
row.put("itemType", "Hats");
row.put("itemCategory", "baseball");
row.put("itemClass", "longbill");
row.put("itemColor", "red");
row.put("itemSize", "medium");
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
row.put("price", new Float(14.07));
row.put("inventoryCount", 198);
opList.add(tof.createPut(row, null, true));

row = myTable.createRow();
row.put("itemType", "Hats");
row.put("itemCategory", "baseball");
row.put("itemClass", "longbill");
row.put("itemColor", "red");
row.put("itemSize", "large");
row.put("price", new Float(15.07));
row.put("inventoryCount", 139);
opList.add(tof.createPut(row, null, true));
```

Note in the above example that we update only those rows that share the same shard key. In this case, the shard key includes the `itemType`, `itemCategory`, and `itemClass` fields. If the value for any of those fields is different from the others, we could not successfully execute the sequence.

Executing a Sequence

To execute the sequence we created in the previous section, use the `TableAPI.execute()` method:

```
package kvstore.basicExample;
try {
    tableH.execute(opList, null);
} catch (OperationExecutionException oee) {
    // Some error occurred that prevented the sequence
    // from executing successfully. Use
    // oee.getFailedOperationIndex() to determine which
    // operation failed. Use oee.getFailedOperationResult()
    // to obtain an OperationResult object, which you can
    // use to troubleshoot the cause of the execution
    // exception.
} catch (IllegalArgumentException iae) {
    // An operation in the list was null or empty.

    // Or at least one operation operates on a row
    // with a shard key that is different
    // than the others.

    // Or more than one operation uses the same key.
} catch (DurabilityException de) {
    // The durability guarantee could not be met.
} catch (RequestTimeoutException rte) {
    // The operation was not completed inside of the
    // default request timeout limit.
} catch (FaultException fe) {
    // A generic error occurred
}
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

Note that if any of the above exceptions are thrown, then the entire sequence is aborted, and your data will be in the state it would have been in if you had never executed the sequence at all.

`TableAPI.execute()` can optionally take a `WriteOptions` class instance. This class instance allows you to specify:

- The durability guarantee that you want to use for this sequence. If you want to use the default durability guarantee, pass `null` for this parameter.
- A timeout value that identifies the upper bound on the time interval allowed for processing the entire sequence. If you provide `0`, the default request timeout value is used.
- A `TimeUnit` that identifies the units used by the timeout value. For example: `TimeUnit.MILLISECONDS`.

For an example of using `WriteOptions`, see Durability Guarantees.

Copyright © 2011, 2026, Oracle and/or its affiliates.

14

Introduction to SQL for Oracle NoSQL Database

SQL for Oracle NoSQL Database is an easy to use SQL-like language that supports read-only queries and data definition (DDL) statements. This chapter focuses on the query part of the language.

For a detailed description of the language (both DDL and query statements), see *SQL Reference Guide*.

To follow along query examples run with the interactive shell, see *SQL Beginner's Guide*.

This section talks about using SQL through the JAVA API.

Running a simple query

Before running a query, perform store access as usual by obtaining a `KVStore` handle using the `KVStoreFactory.getStore()` method and a `KVStoreConfig` object.

To create the query, use `KVStore.executeSync()` This returns a `StatementResult` instance, which represents the result of an execution of a statement. There are two types of results, results of DDL statements and results of DML statements. DDL statements modify the database schema. `CREATE TABLE`, `ALTER TABLE`, and `DROP TABLE` are examples of DDL statements. DDL statements do not return data records, so `iterator()` and `next()` will return as if there was an empty result.

DML statements are non-updating queries. SQL SELECT-FROM-WHERE(SFW) statements are an example of a DML statement. DML statements may contain a set of records. Objects of `StatementResult` are not intended to be used across several threads.

For example, to run a simple query:

```
// Setup Store
String[] hhosts = {"n1.example.org:5088", "n2.example.org:4129"};
KVStoreConfig kconfig = new KVStoreConfig("exampleStore", hhosts);
KVStore store = KVStoreFactory.getStore(kconfig);

// Compile and Execute the SELECT statement
StatementResult result = store.executeSync("SELECT firstName,
age FROM Users");

// Get the results
for( RecordValue record : result ) {
     System.out.println("nameFirst: " +
     record.get("firstName").asString().get());
     System.out.println("age: " +
     record.get("age").asInteger().get());
}
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

where the query SELECTS the firstname and age from the table Users. Then, the results are displayed.

Using binding variables

To declare a binding variable, you need to create an instance of `PreparedStatement`. An instance of PreparedStatement can be created through the `KVStore.prepare()` method.

You can specify zero or more variable declarations. The syntax for a variable is:

```
DECLARE $varname vartype;
```

If the DML statement contains external variables, the `PreparedStatement` can be executed multiple times by creating an instance of `BoundStatement`. The external variables must be bound to specific values before the statement can be executed. To allow for the potentially concurrent execution of the same `PreparedStatement` multiple times with different bind values each time, binding of external variables must be done through one or more instances of `BoundStatement`. Such instances are created using the `createBoundStatement()` method.

This instance can then be executed multiple times using the `KVStore.execute()` or `KVStore.executeSync()` methods.

For example:

```
// store handle creation omitted.

...

// Compile the statement.
PreparedStatement pStmt = store.prepare(
    "DECLARE $minAge integer; $maxAge integer;  " +
    "SELECT id, firstName FROM Users WHERE
    age >= $minAge and age < $maxAge "
);

// Iterate decades
for( int age = 0; age <= 100; age = age + 10 ) {
    int maxAge = age + ( age < 100 ? 10 : 1000 );
    System.out.println("Persons with ages between " + age +
        " and " + maxAge + ".");

    // Bind variables, reuse the same pStmt
    BoundStatement bStmt = pStmt.createBoundStatement();
    bStmt.setVariable("$minAge", age);
    bStmt.setVariable("$maxAge", maxAge);

    // Execute the statement
    StatementResult result = store.executeSync(bStmt);

    // Get the results in the current decade
    for( RecordValue record : result ) {
        System.out.println("id: " +
            record.get("id").asInteger().get() );
        System.out.println("firstName: " +
            record.get("firstName").asString().get());
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
    }
}
```

Accessing metadata

You can access the metadata of a BoundStatement, PreparedStatement or StatementResult by using the `getResultDef()` method.

Additionally, you can use the `getFields().size()`, `getFieldsName()`, and `getField()` `RecordDef` methods to obtain the number of fields, field name, and field type respectively.

For example:

```
// store handle creation omitted.

...

// Access metadata on PreparedStatement or BoundStatement
PreparedStatement pStmt = store.prepare(
   "DECLARE $minAge integer; $maxAge integer;  " +
   "SELECT id, firstName FROM users WHERE age >= $minAge
   and age < $maxAge ");

RecordDef recodDef = pStmt.getResultDef();
int noOfFields = recodDef.getFields().size();
String fieldName = recodDef.getFieldName(0);  // fieldName is "$minAge";
FieldDef fieldType = recodDef.getField(0);    // feldType is IntegerDef


// Access metadata on StatementResult
StatementResult result = store.executeSync("SELECT * FROM Users WHERE
(age > 18 and age < 30)");

recordDef = result.getResultDef();
```

> **Note**
>
> DDL operations do not have metadata.

Using a query to update data

You can form queries to UPDATE a row in an Oracle NoSQL Database table. The WHERE clause must specify an exact primary key as only single row updates are allowed.

For example, to update a field using the UPDATE statement:

```
// store handle creation omitted.

...

// Updates the age for User with id=2
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
StatementResult result = store.executeSync("UPDATE Users SET age=20
WHERE id=2");
```

To update multiple rows, you must first form a query to SELECT records. You then use the result of the SELECT query to update or insert data.

For example, to update a field using a result record from the SELECT statement:

```
// store handle creation omitted.

...

TableAPI tableAPI = store.getTableAPI();
Table table = tableAPI.getTable("Users");

StatementResult result = store.executeSync("SELECT * FROM Users WHERE
(age > 13 and age < 17)");

for( RecordValue record : result ) {

    // Update a field
    Row row = table.createRow(record);
    row.put("age", record.get("age").asInteger().get() + 1);
    tableAPI.put(row, null, null);
}
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

A
JSON By Example

This appendix contains a complete Java example of how to use JSON data in a Oracle NoSQL Database store.

The example loads a series of table rows, using JSON objects to represent each row. The example then updates all table rows that contain a home address in Boston so that the zip code for that address is updated from `02102` to `02102-1000`.

Our sample data deliberately contains some table rows with null and missing fields so as to illustrate some (but by no means all) of the error handling that is required when working with JSON data. It is possible to be endlessly creative when providing broken JSON to the store. Any production code would have to be a great deal more robust than what is shown here.

The update operation is shown three different ways in the following example. While the actual update is always the same (see the UpdateJSON.updateZipCode() method), there are three different ways to seek out rows with a home address in Boston:

- No query.

  This simply iterates over the entire table, examining each row in turn. See UpdateJSON.updateTableWithoutQuery().

- With an index.

  This uses a JSON index to retrieve all table rows where the home address is in Boston. See UpdateJSON.updateTableWithIndex().

- With a SQL Query.

  This uses a SQL statement with a `executeSync()` method to retrieve all relevant table rows. See UpdateJSON.updateTableUsingSQLQuery().

The next section shows some of the sample data used by this example. The description of the example itself begins with UpdateJSON.

If you want to follow along with the example, and see all of the sample data, you can find this example in the `Examples` download from here. The example and its sample data can be found in the `Table` folder.

When compiling the example, make sure that `kvclient.jar` is in your classpath. For example:

```
javac -d . -cp <KVHOME>/lib/kvclient.jar UpdateJSON.java
```

You can run this program against a store or a kvlite instance that does not have security enabled.

```
java -cp .:<KVHOME>/lib/kvclient.jar table.UpdateJSON
```

By default, this example uses `localhost:5000`, but you can set the helper host and port at the command line using the `-hostport` parameter.

Copyright © 2011, 2026, Oracle and/or its affiliates.

Sample Data

Our sample data is contained in `person_contacts.json`. We use it create a simple two-column table to hold our JSON data.

The first column is an account ID, and it serves as the primary key. At a minimum, every table will always have a single non-JSON field that serves as the primary key. If you wish to use compound primary keys, or one or more shard keys, then the number of non-JSON fields will expand.

Our second field is a JSON field. Like all such fields, it can contain any valid JSON data. This provides extreme flexibility for your table schema, which is particularly useful when it is necessary to evolve your data's schema. However, it comes at the cost of requiring more error checking to ensure that your JSON contains the data you expect it to contain.

```
CREATE TABLE personContacts (account INTEGER,
                             person JSON,
                             PRIMARY KEY(account))
```

We load this table with 23 rows, some of which are deliberately incomplete. Each row is represented as a single JSON object. We show a representative section of the sample data file, below, for your reference.

In the following listing, notice that Account 3 only provides a work address — there is no home address. Account 4 provides no address information at all. Account 5 fails to provide any data at all for the `person` field. Account 22 explicitly sets the address object to `null`. Account 23 explicitly sets both the home and work addresses to null. All of this is valid JSON and all of it should be handled by our code.

```
{
"account" : 1,
"person" : {
      "lastName" : "Jones",
      "firstName" : "Joe",
      "address" : {
        "home" : {
          "street" : "15 Elm",
          "city" : "Lakeville",
          "zip" : "12345"
        },
        "work" : {
          "street" : "12 Main",
          "city" : "Lakeville",
          "zip" : "12345"
        }
      },
      "phone" : {
        "home" : "800-555-1234",
       "work" : "877-123-4567"
      }
   }
}

{
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
"account" : 2,
"person" : {
      "lastName" : "Anderson",
      "firstName" : "Nick",
      "address" : {
        "home" : {
          "street" : "4032 Kenwood Drive",
          "city" : "Boston",
          "zip" : "02102"
        },
        "work" : {
          "street" : "541 Bronx Street",
          "city" : "Boston",
          "zip" : "02102"
        }
      },
      "phone" : {
        "home" : "800-555-9201",
       "work" : "877-123-8811"
      }
   }
}

{
"account" : 3,
"person" : {
      "lastName" : "Long",
      "firstName" : "Betty",
      "address" : {
        "work" : {
          "street" : "10 Circle Drive",
          "city" : "Minneapolis",
          "zip" : "55111"
        }
      },
      "phone" : {
        "home" : "800-555-2701",
       "work" : "877-181-4912"
      }
   }
}

{
"account" : 4,
"person" : {
      "lastName" : "Brown",
      "firstName" : "Harrison",
      "phone" : {
        "home" : "800-555-3838",
       "work" : "877-753-4110"
      }
   }
}

{
"account" : 5
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
}

{
"account" : 6,
"person" : {
      "lastName" : "Abrams",
      "firstName" : "Cynthia",
      "address" : {
        "home" : {
          "street" : "2 Fairfield Drive",
          "city" : "San Jose",
          "zip" : "95054"
        }
      },
      "phone" : {
        "home" : "800-528-4897",
       "work" : "877-180-5287"
      }
   }
}

# ...
# sample data removed for the book. See person_contact.json
# in ..../examples/table for the complete data
# file.
# ...

{
"account" : 21,
"person" : {
      "lastName" : "Blase",
      "firstName" : "Lisa",
      "address" : {
        "home" : {
          "street" : "72 Rutland Circle",
          "city" : "Boston",
          "zip" : "02102"
        },
        "work" : {
          "street" : "541 Bronx Street",
          "city" : "Boston",
          "zip" : "02102"
        }
      },
      "phone" : {
        "home" : "800-555-4404",
       "work" : "877-123-2277"
      }
   }
}

{
"account" : 22,
"person" : {
      "address" : null,
      "phone" : {
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
        "home" : "800-555-1234",
       "work" : "877-123-4567"
      }
   }
}

{
"account" : 23,
"person" : {
      "address" : {
        "home" : null,
        "work" : null
      },
      "phone" : {
        "home" : "800-555-1234",
       "work" : "877-123-4567"
      }
   }
}
```

UpdateJSON

The example program is called `UpdateJSON`. We deliberately avoid using Java JSON APIs in this example so as to show how to perform these operations using Oracle NoSQL Database APIs only. Our imports are therefore limited to `oracle.kv`, `oracle.kv.table`, `java.io`, and `java.util`.

```
package table;

import oracle.kv.FaultException;
import oracle.kv.KVStore;
import oracle.kv.KVStoreConfig;
import oracle.kv.KVStoreFactory;
import oracle.kv.StatementResult;

import oracle.kv.table.FieldValue;
import oracle.kv.table.Index;
import oracle.kv.table.IndexKey;
import oracle.kv.table.MapValue;
import oracle.kv.table.PrimaryKey;
import oracle.kv.table.RecordValue;
import oracle.kv.table.Row;
import oracle.kv.table.Table;
import oracle.kv.table.TableAPI;
import oracle.kv.table.TableIterator;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

import java.util.ArrayList;


public class UpdateJSON {
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
private String dataFile = "person_contacts.json";
private String defaulthost = "localhost:5000";
private String helperhosts[];
private String storeName = "kvstore";


private static void usage() {
    String msg = "Creates a table and loads data into it from\n";
    msg += "an external file containing one or more JSON\n";
    msg += "objects. The objects must conform to the table\n";
    msg += "schema. Table rows are then updated so that\n";
    msg += "zipcodes for all home addresses in Boston are\n";
    msg += "modified updated. Update is performed 3 different\n";
    msg += "ways so as to illustrate the ways to query JSON\n";
    msg += "data in Oracle NoSQL Database.\n";
    msg += "\nCommand line options: \n";
    msg += "-store <storename>\n";
    msg += "\tName of the store. Defaults to 'kvstore'\n";
    msg += "-hostport <hostname>:<port>\n";
    msg += "\tStore location. Defaults to 'localhost:5000'\n";
    msg += "-file <filename>\n";
    msg += "\tFile containing row data. Defaults to ";
    msg += "person_contacts.json";


    System.out.println(msg);
    System.exit(0);
}

public static void main(String args[]) {
    UpdateJSON uj = new UpdateJSON();

    uj.run(args);
}
```

UpdateJSON.run()

The `UpdateJSON.run()` method parses our command line arguments, sets up and opens our `KVStore` handle, and then calls each of the methods that provide individual steps in this example.

Notice that there are three different `updateTable...` methods. Each provides the same functionality as the next, but reads are performed in different ways. Once data is loaded into the table, they can be run independently of the others. The only other dependency is that UpdateJSON.createIndex() must be run before UpdateJSON.updateTableWithIndex() is run.

```
private void run(String args[]) {
    parseArgs(args);

    KVStoreConfig kconfig =
        new KVStoreConfig(storeName,
                          helperhosts);
    KVStore kvstore = KVStoreFactory.getStore(kconfig);

    defineTable(kvstore);
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
        loadTable(kvstore, dataFile);
        displayTable(kvstore);
        updateTableWithoutQuery(kvstore);
        createIndex(kvstore);
        updateTableWithIndex(kvstore);
        updateTableUsingSQLQuery(kvstore);
        displayTable(kvstore);
    }
```

UpdateJSON.defineTable()

The `defineTable()` method drops (deletes) the `personContacts` table if it exists. Dropping a table deletes all of the table data from the store. The `defineTable()` method then creates the new table without data.

As always, we can write no data to the store until the table has been defined in the store using the appropriate DDL statement.

This method relies on the UpdateJSON.runDDL() method, which we show later in this appendix.

```
    // Drops the example table if it exists. This removes all table
    // data and indexes. The table is then created in the store.
    // The loadTable() method is used to populate the newly created
    // table with data.
    private void defineTable(KVStore kvstore) {
        System.out.println("Dropping table....");
        String statement = "DROP TABLE IF EXISTS personContacts";
        boolean success = runDDL(kvstore, statement);

        if (success) {
            statement =
                  "CREATE TABLE personContacts (" +
                  "account INTEGER," +
                  "person JSON," +
                  "PRIMARY KEY(account))";
            System.out.println("Creating table....");
            success = runDDL(kvstore, statement);
            if (!success) {
                System.out.println("Table creation failed.");
                System.exit(-1);
            }
        }

    }
```

UpdateJSON.createIndex()

The `UpdateJSON.createIndex()` method creates a JSON index in the store. It must be run before UpdateJSON.updateTableWithIndex() is run.

Copyright © 2011, 2026, Oracle and/or its affiliates.

For information on JSON indexes, see JSON Indexes.

```
// Creates a JSON index. This method must be
// run before updateTableWithIndex() is run.
private void createIndex(KVStore kvstore) {
    System.out.println("Creating index....");
    String statement = "CREATE INDEX IF NOT EXISTS ";
    statement += "idx_home_city on personContacts ";
    statement += "(person.address.home.city AS String)";
    runDDL(kvstore, statement);
}
```

UpdateJSON.runDDL()

The `UpdateJSON.runDDL()` method is a utility that executes DDL statements against the store using `KVStore.executeSync()`.

This method relies on the UpdateJSON.displayResult() method, which simply writes the results of the DDL execution to the command line. It is shown later in this example.

```
// Executes DDL statements (such as are found in defineTable()
// and createIndex()) in the store.
private boolean runDDL(KVStore kvstore, String statement) {
    StatementResult result = null;
    boolean success = false;

    try {
        result = kvstore.executeSync(statement);
        displayResult(result, statement);
        success = true;
    } catch (IllegalArgumentException e) {
        System.out.println("Invalid statement:\n" + e.getMessage());
    } catch (FaultException e) {
        System.out.println
            ("Statement couldn't be executed, please retry: " + e);
    }

    return success;
}
```

UpdateJSON.updateTableWithoutQuery()

The `UpdateJSON.updateTableWithoutQuery()` method iterates over every row in our table looking for the proper rows to update.

This is by far the most complicated of the update methods due to the requirement to continually check for null fields. Notice that all of the following code is used to simply retrieve table rows. The actual update operation is performed by UpdateJSON.updateZipCode().

```
// Utility method. Given a MapValue and a field name,
// return the field as a MapValue. Used by
// updateTableWithoutQuery()
private MapValue getMV(MapValue mv, String field) {
    FieldValue fv = null;
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
    if ((fv = mv.get(field)) != null)
        return fv.asMap();
    return null;
}

// Update the zip code found on all Boston home addresses
// to "02102-1000"
//
// Because we are not using an index, we must iterate over
// every row in the table, modifying the rows with Boston home
// addresses.
private void updateTableWithoutQuery(KVStore kvstore) {
    TableAPI tableH = kvstore.getTableAPI();
    Table myTable = tableH.getTable("personContacts");

    PrimaryKey pkey = myTable.createPrimaryKey();
    TableIterator<Row> iter =
        tableH.tableIterator(pkey, null, null);
    try {
        while (iter.hasNext()) {
            int account = 0;
            Row row = iter.next();
            FieldValue fv = null;
            try {
                account = row.get("account").asInteger().get();
                MapValue mv = row.get("person").asMap();

                MapValue mvaddress = getMV(mv, "address");
                if (mvaddress != null) {
                    MapValue mvhome = getMV(mvaddress, "home");
                    if (mvhome != null) {
                        fv = mvhome.get("city");
                        if (fv != null) {
                            if (fv.toString()
                                    .equalsIgnoreCase("Boston"))
                                updateZipCode(tableH,
                                              row,
                                              "home",
                                              "02102-1000");
                        }
                    }
                }
            } catch (ClassCastException cce) {
                System.out.println("Data error: ");
                System.out.println("Account " + account +
                     "has a missing or incomplete person field");
                // If this is thrown, then the "person" field
                // doesn't exist for the row.
            }
        }
    } finally {
        if (iter != null) {
            iter.close();
        }
    }
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
        System.out.println("Updated a table without using a query.");
    }
```

UpdateJSON.updateTableWithIndex()

The `UpdateJSON.updateTableWithIndex()` method performs the update using an index.

This read operation is considerably easier to implement than the previous method because we do not need to perform all the error checking. This method is also more efficient because only the table rows identified by the index are returned, resulting in less network traffic and fewer rows for our code to examine. However, if the required index does not exist, then this method will fail.

```
    // Update the zip code found on all Boston home addresses
    // to "02102-1000"
    //
    // Because we have an index available to us, we only have to look
    // at those rows which have person.address.home.city = Boston.
    // All other rows are skipped during the read operation.
    private void updateTableWithIndex(KVStore kvstore) {
        TableAPI tableH = kvstore.getTableAPI();
        Table myTable = tableH.getTable("personContacts");

        // Construct the IndexKey.
        Index homeCityIdx = myTable.getIndex("idx_home_city");
        IndexKey homeCityIdxKey = null;

        // If NullPointerException is thrown by createIndexKey(),
        // it means that the required index has not been created.
        // Run the createIndex() method before running this method.
        homeCityIdxKey = homeCityIdx.createIndexKey();

        // Return only those entries with a home city of "Boston"
        homeCityIdxKey.put("person.address.home.city", "Boston");

        // Iterate over the returned table rows. Because we're
        // using an index, we're guaranteed that
        // person.address.home.city exists and equals Boston
        // for every table row seen here.
        TableIterator<Row> iter =
            tableH.tableIterator(homeCityIdxKey, null, null);
        try {
            while (iter.hasNext()) {
                Row row = iter.next();
                updateZipCode(tableH, row, "home", "02102-1000");
            }
        } finally {
            if (iter != null) {
            iter.close();
            }
        }
        System.out.println("Updated a table using an index.");
    }
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

UpdateJSON.updateTableUsingSQLQuery()

The `UpdateJSON.updateTableUsingSQLQuery()` method uses a Oracle NoSQL Database SQL query to retrieve the required table rows.

This third and final query method is the easiest to implement, and it has the advantage of not requiring an index. If an appropriate index is available, it will be used — with all the advantages that an index offers — but the index is not required as it was for the previous method.

```
// Update the zip code found on all Boston home addresses
// to "02102-1000"
//
// This query works with or without an index. If an index is
// available, it is automatically used. For larger datasets,
// the read operation will be faster with an index because only
// the rows where person.address.home.city=Boston are returned
// for the read.
private void updateTableUsingSQLQuery(KVStore kvstore) {
    TableAPI tableH = kvstore.getTableAPI();
    Table myTable = tableH.getTable("personContacts");

    String query = "select * from personContacts p ";
    query += "where p.person.address.home.city=\"Boston\"";

    StatementResult result = kvstore.executeSync(query);

    for (RecordValue rv : result) {
        Row row = myTable.createRowFromJson(rv.toString(),
                  false);
        updateZipCode(tableH, row, "home", "02102-1000");
    }
    System.out.println("Updated a table using a SQL Query to " +
    "read.");
}
```

UpdateJSON.updateZipCode()

The `UpdateJSON.updateZipCode()` method performs the actual update operation in the store.

Because JSON can be returned as a `MapValue`, it is simple and efficient to update the JSON data field.

```
// Updates the zipcode for the proper address (either "home"
// or "work" in this example).
//
// The calling method must guarantee that this row contains a
// home address which refers to the correct city.
private void updateZipCode(TableAPI tableH, Row row,
        String addrType, String newzip) {

    MapValue homeaddr = row.get("person").asMap()
                           .get("address").asMap()
                           .get(addrType).asMap();
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
        // If the zip field does not exist in the home address,
        // it is created with the newzip value. If it currently
        // exists, it is updated with the new value.
        homeaddr.put("zip", newzip);

        // Write the updated row back to the store.
        // Note that if this was production code, we
        // should be using putIfVersion() when
        // performing this write to ensure that the row
        // has not been changed since it was originally read.
        tableH.put(row, null, null);
    }
```

UpdateJSON.loadTable()

The `UpdateJSON.loadTable()` method loads our sample date into the Oracle NoSQL Database store.

As Sample Data shows, all of our table rows are represented as JSON data in a single text file. Each row is a single JSON object in the file. Typically JSON files are expected to contain one and only one JSON object. While there are third party libraries which will iterate over multiple JSON objects found in a stream, we do not want to rely on them for this example. (In the interest of simplicity, we avoid adding a third party dependency.) Consequently, this method provides a primitive custom parser to load the example data into the store.

```
    // Loads the contents of the sample data file into
    // the personContacts table. The defineTable() method
    // must have been run at least once (either in this
    // runtime, or in one before it) before this method
    // is run.
    //
    // JSON parsers ordinarily expect one JSON Object per file.
    // Our sample data contains multiple JSON Objects, each of
    // which represents a single table row. So this method
    // implements a simple, custom, not particularly robust
    // parser to read the input file, collect JSON Objects,
    // and load them into the table.
    private void loadTable(KVStore kvstore, String file2load) {
        TableAPI tableH = kvstore.getTableAPI();
        Table myTable = tableH.getTable("personContacts");

        BufferedReader br = null;
        FileReader fr = null;

        try {
          String jObj = "";
          String currLine;
          int pCount = 0;
          boolean buildObj = false;
          boolean beganParsing = false;

          fr = new FileReader(file2load);
          br = new BufferedReader(fr);

          // Parse the example data file, loading each JSON object
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
        // found there into the table.
        while ((currLine = br.readLine()) != null) {
            pCount += countParens(currLine, '{');

            // Empty line in the data file
            if (currLine.length() == 0)
                continue;

            // Comments must start at column 0 in the
            // data file.
            if (currLine.charAt(0) == '#')
                continue;

            // If we've found at least one open paren, it's time to
            // start collecting data
            if (pCount > 0) {
                buildObj = true;
                beganParsing = true;
            }

            if (buildObj) {
                jObj += currLine;
            }

            // If our open and closing parens balance (the count
            // is zero) then we've collected an entire object
            pCount -= countParens(currLine, '}');
            if (pCount < 1)
                buildObj = false;
            // If we started parsing data, but buildObj is false
            // then that means we've reached the end of a JSON
            // object in the input file. So write the object
            // to the table, which means it is written to the
            // store.
            if (beganParsing && !buildObj) {
                Row row = myTable.createRowFromJson(jObj, false);
                tableH.put(row, null, null);
                jObj = "";
            }

        }

        System.out.println("Loaded sample data " + file2load);

    } catch (FileNotFoundException fnfe) {
        System.out.println("File not found: " + fnfe);
        System.exit(-1);
    } catch (IOException ioe) {
        System.out.println("IOException: " + ioe);
        System.exit(-1);
    } finally {
        try {
            if (br != null)
                br.close();
            if (fr != null)
                fr.close();
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
            } catch (IOException iox) {
                System.out.println("IOException on close: " + iox);
            }
        }
    }

    // Used by loadTable() to know when a JSON object
    // begins and ends in the input data file.
    private int countParens(String line, char p) {
        int c = 0;
        for( int i=0; i < line.length(); i++ ) {
            if( line.charAt(i) == p ) {
                    c++;
            }
        }

        return c;
    }
```

UpdateJSON.displayTable()

The `UpdateJSON.displayTable()` method simply writes the entire table to the command line.

This method does not format the table's contents in any significant way. It is simply provided as a convenience to allow the user to see that data has in fact been modified in the store.

```
    // Dumps the entire table to the command line.
    // Output is unformatted.
    private void displayTable(KVStore kvstore) {
        TableAPI tableH = kvstore.getTableAPI();
        Table myTable = tableH.getTable("personContacts");

        PrimaryKey pkey = myTable.createPrimaryKey();
        TableIterator<Row> iter = tableH.tableIterator(pkey, null,
                null);
        try {
            while (iter.hasNext()) {
                Row row = iter.next();
                System.out.println("\nAccount: " +
                        row.get("account").asInteger());
                if (row.get("person").isNull()) {
                    System.out.println("No person field");
                } else {
                    System.out.println(row.get("person").asMap());
                }
            }
        } finally {
            if (iter != null) {
                iter.close();
            }
        }
    }
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

UpdateJSON.displayResult()

The `UpdateJSON.displayResult()` method shows the contents of a `StatementResult` object returned by a `KVStore.executeSync()` method call. It is used by this example as a convenience to help the user see that a DDL statement has executed correctly in the store.

```
// Displays the results of an executeSync() call.
private void displayResult(StatementResult result,
        String statement) {
    System.out.println("============================");
    if (result.isSuccessful()) {
        System.out.println("Statement was successful:\n\t" +
                statement);
        System.out.println("Results:\n\t" + result.getInfo());
    } else if (result.isCancelled()) {
        System.out.println("Statement was cancelled:\n\t" +
                statement);
    } else {
         // statement wasn't successful: may be in error, or may
         // still be in progress.
        if (result.isDone()) {
            System.out.println("Statement failed:\n\t" +
                    statement);
            System.out.println("Problem:\n\t" +
                    result.getErrorMessage());
        } else {
            System.out.println("Statement in progress:\n\t" +
                    statement);
            System.out.println("Status:\n\t" +
                    result.getInfo());
        }
    }
}
```

UpdateJSON.parseArgs()

The `UpdateJSON.parseArgs()` method is used to parse the command line arguments used with this class at run time.

It is unlikely that this method holds any surprises for Java programmers. It is included here purely for the sake of completeness.

```
// Parse command line arguments
private void parseArgs(String[] args)
{
    final int nArgs = args.length;
    int argc = 0;
    ArrayList<String> hhosts = new ArrayList<String>();

    while (argc < nArgs) {
        final String thisArg = args[argc++];

        if (thisArg.equals("-store")) {
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
                    if (argc < nArgs) {
                        storeName = args[argc++];
                    } else {
                        usage();
                    }
                } else if (thisArg.equals("-hostport")) {
                    if (argc < nArgs) {
                        hhosts.add(args[argc++]);
                    } else {
                        usage();
                    }
                } else if (thisArg.equals("-file")) {
                    if (argc < nArgs) {
                        dataFile = args[argc++];
                    } else {
                        usage();
                    }
                } else if (thisArg.equals("?") ||
                        thisArg.equals("help")) {
                    usage();
                } else {
                    usage();
                }
            }

            if (hhosts.isEmpty()) {
                helperhosts = new String [] {defaulthost};
            } else {
                helperhosts = new String[hhosts.size()];
                helperhosts = hhosts.toArray(helperhosts);
            }
        }
    }
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

B

Table Data Definition Language Overview

Before you can write data to tables in the store, you must provide a definition of the tables you want to use. This definition includes information such as the table's name, the name of its various rows and the data type contained in those rows, identification of the primary and (optional) shard keys, and so forth. To perform these definitions, Oracle NoSQL Database provides a Data Definition Language (DDL) that you use to form table and index statements. The best way to run DDL statements is from the SQL Shell as described in Running the SQL Shell in the *SQL Beginner's Guide*.

From the SQL, you can use these statements:

- Define tables and sub-tables.
- Modify table definitions.
- Delete table definitions.
- Define indexes.
- Delete index definitions.
- Set and modify default Time-to-Live values for tables.

Table and index statements take the form of ordinary strings, which are then transmitted to the Oracle NoSQL Database store using the appropriate method or function. For example, to define a simple user table, the table statement might look like this:

```
SQL-> CREATE TABLE Users (
    id INTEGER,
    firstName STRING,
    lastName STRING,
    contactInfo JSON,
    PRIMARY KEY (id)
)
```

For information on how to transmit these statements to the store, see Introducing Oracle NoSQL Database Tables and Indexes.

For overview information on primary and shard keys, see Primary and Shard Key Design.

For overview information on indexes, see Creating Indexes.

The remainder of this appendix describes in detail the DDL statements that you use to manipulate table and index definitions in the store.

Name Constraints

Throughout this document, and elsewhere in the documentation, using uppercase text signifies DDL keywords (such as STRING, CREATE TABLE, and so on). These keywords are actually case-insensitive and you can enter them in lowercase characters. However, all DDL keywords shown here are reserved words. You cannot use keywords as table, index, or field names.

Copyright © 2011, 2026, Oracle and/or its affiliates.

Table, index, and field names are case-preserving, but case-insensitive. For example, you can create a field named `MY_NAME`, and later reference it as `my_name` without error. However, whenever you display the field name, it displays in the way you created it, `MY_NAME`, in this case.

Table names are limited to 256 characters. Namespace names, index names, and field names are limited to 64 characters. All table, index, and field names must begin with a letter (A-Z, a-z), and are restricted to alphanumeric characters (A-Z, a-z, 0–9), plus underscore (_) and a period (.) character.

DDL Comments

You can include comments in your DDL statements using one of the following constructs:

```
id INTEGER, /* this is a comment */
firstName STRING, // this is a comment
lastName STRING, # this is a comment
```

CREATE TABLE

To create a table definition, use a `CREATE TABLE` statement, as follows:

```
CREATE TABLE [IF NOT EXISTS] [namespace:]table-name
 [COMMENT "comment string"]
 (field-definition, field-definition-2 [,...]
    PRIMARY KEY (field-name, field-name-2 [,...] ),
) [USING TTL ttl]
  [IN REGIONS region-name,region-name-2 [,...]]
```

where:

- `IF NOT EXISTS`

  Optional clause. If you use this clause and a table of the same name and definition already exists in the current namespace, the statement neither creates a new table, nor returns an error. No action occurs.

  If you do not use `IF NOT EXISTS`, and a table of the same name and definition already exists in the current namespace, the statement attempts to create the table, and fails. You cannot have two tables with the same name in one namespace.

- *`table-name`*

  Required. Specifies the full table name, regardless of where it exists in the table hierarchy. A table can be a top-level table created in the default namespace (`sysdefault`), a table in a non-default namespace, or a child or grandchild table of a parent table. Specify a fully-qualified table name as follows:

  - *`parent-table.child-table`* – Specifies a new child table of an existing parent. To create a child table, specify the parent table followed by a period (.) before the child name. For example, if the parent table is `Users`, define a child table named `MailingAddress` as `Users.MailingAddress`.
  - *`parent-table.child-table.grandchild-table`* – Specifies a child table of an existing table. You must also specify the parent table, for example `Users.MailingAddress.Zip`.

Copyright © 2011, 2026, Oracle and/or its affiliates.

  - *namespace-name*:*table-name* – Identifies a new table in a specific namespace, either user-created or in the default namespace (`sysdefault:`). Use the namespace followed by a colon (:) as a prefix to the new table name. For a new child table of a table in a namespace, or any further generation, use a fully qualified table name such as `Sales:Users.MailingAddress`, or `Sales:Users.MailingAddress.Zip`.

- `COMMENT`

  Optional. Use a comment to provide a brief description of the table. The comment is not interpreted at runtime, but becomes part of the table's metadata.

- *field-definition*

  Required. A comma-separated list of fields. There are one or more field definitions for every table. Field definitions are described next in this section.

- `PRIMARY KEY`

  Required for every table. Identifies at least one field in the table that is the primary key. For information on primary keys, see Primary Keys.

  > **Note**
  >
  > If the primary key field is an INTEGER data type, you can apply a serialized size constraint to it. See Integer Serialized Constraints for details.

  To optionally define a shard key, use the `SHARD` keyword within the primary key statement. For information on shard keys, see Shard Keys.

  For example:

  ```
  PRIMARY KEY (SHARD(id), lastName)
  ```

- `USING TTL`

  Optional. Defines a default time-to-live value for the table's rows. See USING TTL for information on this clause.

- `IN REGIONS`
  Optional. In case, the table being created is an MR Table, this parameter lists all the regions that the table should span. You must mention at least one remote region in this clause to create the table as an MR Table. For information on MR Tables, see Life Cycle of MR Tables in the *Concepts Guide*.

Field Definitions

When defining a table, field definitions take the form:

```
field-name type [constraints] [COMMENT "comment-string"]
```

where:

- *field-name* is the name of the field. For example: `id` or `familiarName`. Every field must have a name.
- *type* describes the field's data type. This can be a simple type such as INTEGER or STRING, or it can be a complex type such a RECORD. The list of allowable types is described in the next section.

Copyright © 2011, 2026, Oracle and/or its affiliates.

- *constraints* describes any limits placed on the data contained in the field. That is, allowable ranges or default values. Ann IDENTITY field, to be created by a sequence generator, is also permissable. This information is optional. See Field Constraints for more information.
- `COMMENT` is optional. You can use this to provide a brief description of the field. The comment will not be interpreted but it is included in the table's metadata.

Supported Data Types

The following data types are supported for table fields:

- `ARRAY`

  An array of data. All elements of the array must be of the same data type, and this type must be declared when you define the array field. For example, to define an array of strings:

  ```
  myArray ARRAY(STRING)
  ```

- `BINARY`

  Binary data.

- `BINARY(`*`length`*`)`

  Fixed-length binary field of size *length* (in bytes).

- `BOOLEAN`

  A boolean data type.

- `DOUBLE`

  A double.

- `ENUM`

  An enumerated list. The field definition must provide the list of allowable enumerated values. For example:

  ```
  fruitName ENUM(apple,pear,orange)
  ```

- `FLOAT`

  A float.

- `INTEGER`

  An integer.

- `JSON`

  A JSON-formatted string.

- `LONG`

  A long.

- `MAP`

  A data map. All map keys are strings, but when defining these fields you must define the data type of the data portion of the map. For example, if your keys map to integer values, then you define the field like this:

  ```
  myMap MAP(INTEGER)
  ```


Copyright © 2011, 2026, Oracle and/or its affiliates.

- Number

  A numeric type capable of handling any type of number of any value or precision.

- `RECORD`

  An embedded record. This field definition must define all the fields contained in the embedded record. All of the same syntax rules apply as are used for defining an ordinary table field. For example, a simple embedded record might be defined as:

  ```
  myEmbeddedRecord RECORD(firstField STRING, secondField INTEGER)
  ```

  Data constraints, default values, and so forth can also be used with the embedded record's field definitions.

- `STRING`

  A string.

- `TIMESTAMP(<precision>)`

  Represents a point in time as a date and, optionally, a time value.

  Timestamp values have a precision (0 - 9) which represents the fractional seconds to be held by the timestamp. A value of 0 means that no fractional seconds are stored, 3 means that the timestamp stores milliseconds, and 9 means a precision of nanoseconds. When declaring a timestamp field, the precision is required.

Field Constraints

Field constraints define information about the field, such as whether the field can be NULL, or what a row's default value should be. Not all data types support constraints, and individual data types do not support all possible constraints.

Integer Serialized Constraints

You can put a serialized size constraint on an INTEGER data type, provided the INTEGER is used for a primary key field. Doing this can reduce the size the keys in your store.

To do this, use `(n)` after the primary key field name, where *n* is the number of bytes allowed for the integer. The meaningful range for *n* is 1 - 4. For example:

```
create table myId (id integer, primary key(id(3)))
```

The number of bytes allowed defines how large the integer can be. The range is from negative to positive.

> **Note**
>
> Specifying an integer constraint value for number of bytes on an IDENTITY field is not permitted.

Number of Bytes	Allowed Integer Values
1	-63 to 63
2	-8191 to 8191

Copyright © 2011, 2026, Oracle and/or its affiliates.

Number of Bytes	Allowed Integer Values
3	-1048575 to 1048575
4	-134217727 to 134217727
5	Any integer value

COMMENT

All data types can accept a COMMENT as part of their constraint. COMMENT strings are not parsed, but do become part of the table's metadata. For example:

```
myRec RECORD(a STRING, b INTEGER) COMMENT "Comment string"
```

DEFAULT

All fields can accept a DEFAULT constraint, except for ARRAY, BINARY, MAP, and RECORD. The value specified by DEFAULT is used in the event that the field data is not specified when the table is written to the store.

For example:

```
id INTEGER DEFAULT -1,
description STRING DEFAULT "NONE",
size ENUM(small,medium,large) DEFAULT medium,
inStock BOOLEAN DEFAULT FALSE
```

IDENTITY

You can define one IDENTITY field per table. All IDENTITY fields must have a numeric type: INTEGER. LONG, or NUMBER. An IDENTITY field can optionally be a primary key.

There are two ways to define an IDENTITY field. You can optionally specify one or more Sequence Generator attributes for the Sequence Generator (SG) associated with the IDENTITY. These are the options:

- GENERATED ALWAYS AS IDENTITY
- GENERATED BY DEFAULT AS IDENTITY

These are the Sequence Generator attributes you can define:

Attribute	Type	Default Value and Description
`StartWith`	`Number`	**Default:** 1 The first value in the sequence. Zero (0) is not permitted.
`IncrementBy`	`Long`	**Default:** 1 The value to increment the current value, which can be a positive or a negative number. Specifying a negative number for `IncrementBy` decrements values from the `StartWith` value.
`MinValue`	`Number`	**Default:** The minimum value of the field data type. The lower bound of the IDENTITY values that the SG supplies.
`MaxValue`	`Number`	**Default:** The maximum value of the field data type. The upper bound of the IDENTITY values that the SG supplies. If you do not specify this attribute, SG uses the maximum value of the field data type.

Copyright © 2011, 2026, Oracle and/or its affiliates.

Attribute	Type	Default Value and Description
`Cache`	`Long`	**Default:** `1000` The number of values stored in cache on the client to use for the next IDENTITY value. When the set of values is exhausted, the SG requests another set to store in the local cache (unless you specify the `Cycle` attribute).
`Cycle | NoCycle`	`Boolean`	**Default:** `NoCycle` Determines whether to reuse the set of stored values in cache. For example, if the cache stores 1024 integers for the IDENTITY column, and you specify `Cycle`, when the IDENTITY value reaches `1023`, the next row value is `0001`. If you do not specify `Cycle`, Oracle NoSQL Database guarantees that each IDENTITY value in the column is unique, but not necessarily sequential.

For example:

```
CREATE Table T (id INTEGER GENERATED ALWAYS AS IDENTITY
(START WITH 2 INCREMENT BY 2 MAXVALUE 200),
name STRING,
PRIMARY KEY (id));

CREATE Table T_DEFAULT (id LONG GENERATED BY DEFAULT AS IDENTITY
(START WITH 1 INCREMENT BY 1 CYCLE CACHE 200),
account_id INTEGER,
name STRING,
PRIMARY KEY (account_id));
```

UUID

You can define one UUID field per table. UUID is a subtype of the STRING data type. The UUID column can be defined as GENERATED BY DEFAULT. The system then automatically generates a value for the UUID column if you do not supply a value for it.

For example :

```
CREATE TABLE myTable (id STRING AS UUID,name STRING, PRIMARY KEY (id));
```

In the above example, the id column has no "GENERATED BY DEFAULT" defined, therefore, whenever you insert a new row, you need to explicitly specify a value for the id column.

MR_COUNTER

In a multi-region table, you can create an MR_COUNTER datatype. MR_COUNTER datatype ensures that though data modifications can happen simultaneously on different regions, the data can always be merged into a consistent state. This merge is performed automatically by MR_COUNTER datatype, without requiring any special conflict resolution code or user intervention. You can also use the MR_COUNTER data type in a schema-less JSON field.

Example 1:

Copyright © 2011, 2026, Oracle and/or its affiliates.

In the below example, you create a PN counter data type in two regions DEN and LON.

```
CREATE Table myTable( name STRING,
                      count INTEGER AS MR_COUNTER,
                      PRIMARY KEY(name)) IN REGIONS DEN,LON;
```

In the above example, while inserting data into the table, the system initially inserts the default value (0) when you either give the "DEFAULT" keyword or skip the column name in the INSERT clause.

Example 2:

Create a JSON MR_COUNTER data type in a multi-region table.

```
CREATE TABLE demoJSONMR(name STRING,
jsonWithCounter JSON(counter as INTEGER MR_COUNTER,
                      person.count as LONG MR_COUNTER),
PRIMARY KEY(name)) IN REGIONS FRA,LON;
```

NOT NULL

`NOT NULL` indicates that the field cannot be NULL. This constraint requires that you also specify a DEFAULT value. Order is unimportant for these constraints. For example:

```
id INTEGER NOT NULL DEFAULT -1,
description STRING DEFAULT "NONE" NOT NULL
```

USING TTL

`USING TTL` is an optional statement that defines a default time-to-live value for a table's rows. See Using Time to Live for information on TTL.

If specified, this statement must provide a *ttl* value, which is an integer greater than or equal to 0, followed by a space, followed by time unit designation which is either *hours* or *days*. For example:

```
USING TTL 5 days
```

If `0` is specified, then either `days` or `hours` can be used. A value of `0` causes table rows to have no expiration time. Note that `0` is the default if a default TTL has never been applied to a table schema. However, if you previously applied a default TTL to a table schema, and then want to turn it off, use `0 days` or `0 hours`.

```
USING TTL 0 days
```

Be aware that if you altering an existing table, you can not both add/drop a field *and* alter the default TTL value for the field using the same `ALTER TABLE` statement. These two operations must be performed using separate statements.

Copyright © 2011, 2026, Oracle and/or its affiliates.

Table Creation Examples

The following are provided to illustrate the concepts described above.

```
CREATE TABLE users
COMMENT "This comment applies to the table itself" (
  id INTEGER,
  firstName STRING,
  lastName STRING,
  age INTEGER,
  PRIMARY KEY (id),
)


CREATE TABLE temporary
COMMENT "These rows expire after 3 days" (
  sku STRING,
  id STRING,
  price FLOAT,
  count INTEGER,
  PRIMARY KEY (sku),
) USING TTL 3 days


CREATE TABLE Users
COMMENT "This is an MR table"(
  id INTEGER,
  firstName STRING,
  lastName STRING,
  age INTEGER,
  primary key (id)
) IN REGIONS us_east, us_west;


CREATE TABLE usersNoId (
  firstName STRING,
  lastName STRING COMMENT "This comment applies to this field only",
  age INTEGER,
  ssn STRING NOT NULL DEFAULT "xxx-yy-zzzz",
  PRIMARY KEY (SHARD(lastName), firstName)
)


CREATE TABLE users.address (
  streetNumber INTEGER,
  streetName STRING,  // this comment is ignored by the DDL parser
  city STRING,
  /* this comment is ignored */
  zip INTEGER,
  addrType ENUM (home, work, other),
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

```
  PRIMARY KEY (addrType)
)


CREATE TABLE complex
COMMENT "this comment goes into the table metadata" (
  id INTEGER,
  PRIMARY KEY (id), # this comment is just syntax
  nestedMap MAP(RECORD( m MAP(FLOAT), a ARRAY(RECORD(age INTEGER)))),
  address RECORD (street INTEGER, streetName STRING, city STRING, \
                  zip INTEGER COMMENT "zip comment"),
  friends MAP (STRING),
  floatArray ARRAY (FLOAT),
  aFixedBinary BINARY(5),
  days ENUM(mon, tue, wed, thur, fri, sat, sun) NOT NULL DEFAULT tue
)


CREATE TABLE myJSON (
    recordID INTEGER,
    jsonData JSON,
    PRIMARY KEY (recordID)
)
```

Modify Table Definitions

Use `ALTER TABLE` statements to either add new fields to a table definition, or delete a currently existing field definition. You can also use an `ALTER TABLE` statement to change the default Time-to-Live (TTL) value for a table, and to add an IDENTITY column to a table.

ALTER TABLE ADD field

To add a field to an existing table, use the `ADD` statement:

```
ALTER TABLE table-name (ADD field-definition)
```

See Field Definitions for a description of what should appear in *field-definitions*, above. For example:

```
ALTER TABLE Users (ADD age INTEGER)
```

You can also add fields to nested records. For example, if you have the following table definition:

```
CREATE TABLE u (id INTEGER,
                info record(firstName String)),
                PRIMARY KEY(id))
```

then you can add a field to the nested record by using dot notation to identify the nested table, like this:

```
ALTER TABLE u(ADD info.lastName STRING)
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

ALTER TABLE DROP Option

To delete a field from an existing table, use the `DROP` option:

```
ALTER TABLE table-name (DROP field-name)
```

For example, to drop the `age` field from the `Users` table:

```
ALTER TABLE Users (DROP age)
```

> **Note**
>
> You cannot drop a field if it is the primary key, or if it participates in an index.

You can also us the `ALTER TABLE MODIFY FIELD` clause to add, drop, or modify an IDENTITY column in a table.

ALTER TABLE USING TTL

To change the default Time-to-Live (TTL) value for an existing table, use the `USING TTL` statement:

```
ALTER TABLE table-name USING TTL ttl
```

For example:

```
ALTER TABLE Users USING TTL 4 days
```

In case of MR Tables, you can not use the `USING TTL` clause along with the `IN REGIONS` clause. That is, you can not alter an MR table's TTL value and regions in a single statement.

For more information on the `USING TTL` statement, see USING TTL.

ALTER TABLE ADD REGIONS

The `add regions` clause lets you link an existing multi-region table (MR Table) with new regions in a multi-region Oracle NoSQL Database environment.

The `add regions` clause is used in expanding MR Tables to new regions. See Use Case 2: Expand a Multi-Region Table in the *Administrator's Guide*.

To add a region to an existing MR Table, use the `ADD REGIONS` option:

```
ALTER TABLE <table name> ADD REGIONS <comma separated list of regions>
```

See Add New Region to Existing Regions for the example code.

Copyright © 2011, 2026, Oracle and/or its affiliates.

ALTER TABLE DROP REGIONS

The `drop regions` clause lets you disconnect an existing multi-region table (MR Table) from a participating region in a multi-region Oracle NoSQL Database environment.

The `drop regions` clause is used in contracting MR Tables to fewer regions. See Use Case 3: Contract a Multi-Region Table in the *Administrator's Guide.*

To drop a region from an MR Table, use the `DROP REGIONS` option:

```
ALTER TABLE <table name> DROP REGIONS <comma separated list of regions>
```

See Alter the MR Table to Drop Regions for the example code.

DROP TABLE

To delete a table definition, use the `DROP TABLE` statement:

```
DROP TABLE [IF EXISTS] table-name
```

where:

- `IF EXISTS` is optional. If you use this option and the specified table does not exist, the system returns a message:

  ```
  sql-> drop table if exists foo;
  Statement did not require execution
  ```

  If you do not specify `IF EXISTS` , and the table does not currently exist, the DROP statement returns as follows:

  ```
  sql-> drop table foo;
  Error handling command drop table foo:
  Error: User error in query: DROP TABLE failed for table foo:
  Table does not exist: foo
  ```

  If you specify `IF EXISTS`, and the table exists, the DROP statement executes successfully:

  ```
  sql-> create table foo (first string, second integer, primary key
  (second));
  Statement completed successfully
  sql-> drop table if exists foo;
  Statement completed successfully
  ```

- *table-name* is the name of the table you want to drop.

As soon as you execute the DROP TABLE statement, users can no longer access the deleted table or its data. Deleting all of the table data occurs asynchronously in the background after you execute the DROP TABLE statement.

Copyright © 2011, 2026, Oracle and/or its affiliates.

If you choose to drop an MR Table in a particular region, it still continues to remain an MR Table in the other participating regions. In a case where you want to drop a particular MR Table from multiple regions, you must execute the `DROP TABLE` statement in each region separately.

> **Note**
>
> In a case where an MR Table is dropped in all remote regions but still exists in local region, it still continues to be an MR Table linked with a single region. Such an MR Table with a single region can be expanded to more regions in future. That is, you can add new regions to this table in future, as needed.

If the table to drop has child tables, you must drop those first. For example, if you have these tables:

- myTable
- myTable.childTable1
- myTable.childTable2

You must first drop `myTable.childTable1` and `myTable.childTable2` before you can drop the parent `myTable`. If you try to drop the parent, the statement returns an error.

CREATE INDEX

To add an index definition to the store, use a `CREATE INDEX` statement. It can be used to create simple indexes and multi-key indexes. It can also be used to create JSON indexes.

Indexable Field Types

Fields can be indexed only if they are declared to be one of the following types. For all complex types (arrays, maps, and records), the field can be indexed if the ultimate target of the index is a scalar datatype. So a complex type that contains a nested complex type (such as an array of records, for example) can be indexed if the index's target is a scalar datatype contained by the embedded record.

- Integer
- Long
- Number
- Float
- Double
- Json

  Note that there are some differences and restrictions on indexing Json data versus other data types. See JSON Indexes for more information.
- String
- Enum
- Array

  In the case of arrays, the field can be indexed only if the array contains values that are of one of the other indexable scalar types. For example, you can create an index on an array of Integers. You can also create an index on a specific record in an array of records. Only

Copyright © 2011, 2026, Oracle and/or its affiliates.

one array can participate in an index, otherwise the size of the index can grow exponentially because there is an index entry for each array entry.

- Maps

  As is the case with Arrays, you can index a map if the map contains scalar types, or if the map contains a record that contains scalar types.

- Records

  Like Arrays and Maps, you can index fields in an embedded record if the field contains scalar data.

Simple Indexes

An index is simple if it does not index any maps or arrays. To create a simple index:

```
CREATE INDEX [IF NOT EXISTS] index-name ON table-name (path_list)
```

where:

- `IF NOT EXISTS` is optional, and it causes the `CREATE INDEX` statement to be ignored if an index by that name currently exists. If this phrase is not specified, and an index using the specified name does currently exist, then the `CREATE INDEX` statement will fail with an error.
- *index-name* is the name of the index you want to create.
- *table-name* is the name of the table that you want to index.
- *path_list* is a comma-separated list of one or more *name_paths*. A name_path refers to an element of a table. Normally these are schema fields — that is, field names that appear in the CREATE TABLE expression used to create the associated table.

  However, if the table contains a record, then the name_path may be record keys that use dot-notation to identify a record field. For example:

  ```
  CREATE TABLE example (
          id INTEGER,
          myRecord RECORD(field_one STRING, field_two STRING),
          PRIMARY KEY (id)
      )
  ```

  An index can then be created on field_one by using the name_path of `myRecord.field_one`. See Indexing Embedded Records for a more detailed explanation of indexing records.

For example, if table `Users` has a field called `lastName`, then you can index that field with the following statement:

```
CREATE INDEX surnameIndex ON Users (lastName)
```

Note that depending on the amount of data in your store, creating indexes can take a long time. This is because index creation requires Oracle NoSQL Database to examine all the data in the store.

Copyright © 2011, 2026, Oracle and/or its affiliates.

Multi-Key Indexes

Multi-key indexes are used to index all the elements of an array. They are also used to index all of the elements and/or values of a map.

For each table row, a multi-key index contains as many entries as the number of elements/entries in the array/map that is being indexed (although duplicate entries are not represented in the index). To avoid an explosion in the number of index entries, only one array/map may be contained in a single multi-key index.

To create a multi-key index, use one of the following forms:

```
CREATE INDEX [IF NOT EXISTS] index-name ON table-name (name-path.keys())
```

or

```
CREATE INDEX [IF NOT EXISTS] index-name ON table-name (name-path.values())
```

or

```
CREATE INDEX [IF NOT EXISTS] index-name ON table-name \
(name-path.keys(),name-path.values())
```

or

```
CREATE INDEX [IF NOT EXISTS] index-name ON table-name (name-path[])
```

The syntax shown, above, is identical to that described in Simple Indexes, with the following additions:

- `.keys()`

  The index is created on the keys in a map. If used, *name-path* must be a map.

- `.values()`

  The index is created on the values in a map. If used, *name-path* must be a map.

- `[]`

  The index is created on an array. If used, *name-path* must be array.

For each of the previously identified forms, a comma-seperated list of name-paths may be provided. Some restrictions apply.

Multi-Key Index Restrictions

The following restrictions apply to multi-key indexes:

- There is at least one name-path that uses a multi-key step (`.keys()`, `.values()`, or `[]`). Any such path is called a *multi-key path*, and the associated index field a *multi-key field*. The index definition may contain more than one multi-key path, but all multi-key paths must use the same name-path before their multi-key step.
- Any non-multi-key paths must be simple paths.

Copyright © 2011, 2026, Oracle and/or its affiliates.

- The combined path specified for the index must contain at least one map and/or array. These must contain indexable atomic items, or record items, or map items. That is, an index of an array of arrays is not supported, nor is an index of maps containing arrays.

  For example, given the following table definition:

  ```
  create table Foo (
  id INTEGER,
  complex1 RECORD(mapField MAP(ARRAY(MAP(INTEGER)))),
  complex2 RECORD(matrix ARRAY(ARRAY(RECORD(a LONG, b LONG)))
  primary key(id)
  )
  ```

  The path expression `complex2.matrix[]` is not valid, because the result of this path expression is a sequence of arrays, not atomic items. Neither is `complex2.matrix[][].a` valid, because you cannot index arrays inside other arrays (in fact this path will raise a syntax error, because the syntax allows at most one `[]` per index path).

  On the other hand, the path `complex1.mapField.someKey[].someOtherKey` is valid. In this case, the path `complex1.mapField.someKey` specifies an array containing maps, which is valid. Notice that in this index path, `someKey` and `someOtherKey` are map-entry keys. So, although we are indexing arrays that are contained inside maps, and the arrays being indexed contain maps, the path is valid, because it is selecting specific entries from the map, rather than indexing all the map entries in addition to all the array entries.

- If the index is indexing an array-valued field:
  - If the array contains indexable atomic items:
    - There must be a single multi-key index path of the form `M[]` (without any name_path following after the []). Again, this implies that you cannot index more than one array in the same index.
    - For each table row (R), a number of index entries are created as follows:

      The simple index paths (if any) are computed on R.

      Then, M[] is computed (as if it were a query path expression), returning either NULL, or EMPTY, or all the elements of the array returned by M.

      Finally, for each value (V) returned by M[], an index entry is created whose field values are V and the values of the simple paths.
    - Any duplicate index entries (having equal field values and the same primary key) created by the above process are eliminated.
  - If the array contains records or maps:
    - All of the multi-key paths must be of the form `M[].name_path`. Each name_path appearing after M[] in the multi-key index path must return at most one indexable atomic item.
    - For each table row (R), a number of index entries are created as follows:

      The simple index paths (if any) are computed on R.

      Then, M[] is computed (as if it were a query path expression), returning either NULL, or EMPTY, or all the elements of the array returned by M.

      Next, for each value (V) returned by M[], one index entry is created as follows:

      The elements contained in each V are computed. This returns a single indexable atomic item (which may be the NULL or EMPTY item). An index entry is created

Copyright © 2011, 2026, Oracle and/or its affiliates.

for each of these, whose field values are the values of the simple index paths plus the values found for element contained in V.

* Any duplicate index entries (having equal field values and the same primary key) created by the above process are eliminated.

– If the index is indexing a map-valued field, the index may be indexing only map keys, or only map elements, or both keys and elements. In all cases, the definition of map indexes can be given in terms of array indexes, by viewing maps as arrays containing records with 2 fields: a field with name “key” and value a map key, and a field named “element” and value the corresponding map element (that is, MAP(T) is viewed as ARRAY(RECORD(key STRING, element T))). Then, the 2 valid kinds for map indexes are:

1. A single multi-key index path using a keys() step. Using the array view of maps, M.keys() is equivalent to M[].key.
2. One or more multi-key index paths, all using a .values() step. If Ri is an value contained in the map, then each of these has the form M.values().Ri. Using the array view of maps, each M.values().Ri path is equivalent to M[].element.Ri.

JSON Indexes

An index is a JSON index if it indexes at least one field that is contained inside JSON data.

Because JSON is schema-less, it is possible for JSON data to differ in type across table rows. However, when indexing JSON data, the data type must be consistent across table rows or the index creation will fail. Further, once or more JSON indexes have been created, any attempt to write data of an incorrect type will fail.

Indexing JSON data and working with JSON indexes is performed in much the same way as indexing non-JSON data. To create the index, specify a path to the JSON field using dot notation.

When creating JSON indexes, you must specify the data's type, using the `AS` keyword. The data type must be atomic, and cannot be a float. That is, only integer, long, double, number, string, and boolean are supported types for JSON indexes. Note that arrays and maps can be indexed so long as they contain these atomic values.

```
CREATE INDEX [IF NOT EXISTS] index-name ON table-name \
(JSONRow.JSONField AS data_type)
```

When creating a multi-key index on a JSON map, a type must not be given for the `.keys()` expression because the type will always be `String`. However, a type declaration is required for the `.values()` expression. Beyond that, all the constraints described in Multi-Key Index Restrictions also apply to a JSON multi-keyed index.

```
CREATE INDEX [IF NOT EXISTS] index-name ON table-name \
(JSONRow.JSONField.keys(),\
    JSONRow.JSONField.values() AS data_type)
```

For an example of using JSON indexes, see Indexing JSON Fields.

For additional examples of using JSON indexes, see Indexing JSON Data in the *SQL Beginner's Guide*.

Copyright © 2011, 2026, Oracle and/or its affiliates.

CREATE FULL TEXT INDEX

To create a text index on that table that indexes the category and txt columns, use `CREATE FULLTEXT INDEX` statement:

```
CREATE FULLTEXT INDEX [if not exists] <index-name> ON <table-name>
(<field-name> [ <mapping-spec> ], ...)
[ES_SHARDS = <n>] [ES_REPLICAS = <n>]
```

For example:

```
sql-> CREATE FULLTEXT INDEX JokeIndex ON Joke (category, txt);
Statement completed successfully
```

While creating index, `CREATE FULLTEXT INDEX` statement uses the `OVERRIDE` flag, which allows to delete any index existing in Elasticsearch by the same name as would be created by the command.

```
CREATE FULLTEXT INDEX [IF NOT EXISTS] index_name ON table_name
   (field_name [{mapping_spec}] [, field_name [{mapping_spec}]]...)
   [ES_SHARDS = value] [ES_REPLICAS = value]
   [OVERRIDE] [COMMENT comment]
```

For example:

```
CREATE INDEX JokeIndex on  Joke (category, txt) OVERRIDE
```

For more information, see Creating Full Text Index and Mapping a Full Text Index Field to an Elasticsearch Field in the *Integrations Guide*.

DROP INDEX

To delete an index definition from the store, use a `DROP INDEX` statement. Its form when deleting an index is:

```
DROP INDEX [IF EXISTS] index-name ON table-name
```

where:

- `IF EXISTS` is optional, and it causes the `DROP INDEX` statement to be ignored if an index by that name does not exist. If this phrase is not specified, and an index using the specified name does not exist, then the `DROP INDEX` statement will fail with an error.
- *index-name* is the name of the index you want to drop.
- *table-name* is the name of the table containing the index you want to delete.

For example, if table `Users` has an index called `surnameIndex`, then you can delete it using the following statement:

```
DROP INDEX IF EXISTS surnameIndex ON Users
```


Copyright © 2011, 2026, Oracle and/or its affiliates.

Use `DROP INDEX` on a text index to stop the population of the index from NoSQL Database shards, and removes the mapping and all related documents from Elasticsearch. See the following statement:

```
DROP INDEX [IF EXISTS] index_name ON table_name
```

For example:

```
sql-> DROP INDEX JokeIndex on Joke;
Statement completed successfully
```

While deleting index, you can use the `OVERRIDE` flag. The `DROP INDEX` statement uses the `OVERRIDE` flag to enable overriding of the default constraints:

```
DROP INDEX [IF EXISTS] index_name ON table_name [OVERRIDE]
```

For example:

```
DROP INDEX JokeIndex on Joke OVERRIDE
```

For more information, see Deleting FTI in the *Integrations Guide*.

DESCRIBE AS JSON TABLE

You can retrieve a JSON representation of a table by using the `DESCRIBE AS JSON TABLE` statement:

```
DESCRIBE AS JSON TABLE table_name [(field-name, field-name2, ...)]
```

or

```
DESC AS JSON TABLE table_name [(field-name, field-name2, ...)]
```

where:

- *table_name* is the name of the table you want to describe.
- *field-name* is 0 or more fields defined for the table that you want described. If specified, the output is limited to just the fields listed here.

  For Map and Array fields, use `[]` to restrict the JSON representation to just the map or array element.

DESCRIBE AS JSON INDEX

You can retrieve a JSON representation of an index by using the `DESCRIBE AS JSON INDEX` statement:

```
DESCRIBE AS JSON INDEX index_name ON table_name
```

where:

Copyright © 2011, 2026, Oracle and/or its affiliates.

- *index_name* is the name of the index you want to describe.
- *table_name* is the name of the table to which the index is applied.

SHOW TABLES

You can retrieve a list of all tables currently defined in the store using the `SHOW TABLES` statement:

```
SHOW [AS JSON] TABLES
```

where *AS JSON* is optional and causes the resulting output to be JSON-formatted.

SHOW INDEXES

You can retrieve a list of all indexes currently defined for a table using the `SHOW INDEXES` statement:

```
SHOW [AS JSON] INDEXES ON table_name
```

where:

- *AS JSON* is optional and causes the resulting output to be JSON-formatted.
- *table_name* is the name of the table for which you want to list all the indexes.

Copyright © 2011, 2026, Oracle and/or its affiliates.

C

Exceptions

Exceptions will be translated into Spring Data Framework exceptions as follows:

Table C-1 Exceptions

Oracle NoSQL Database	Spring Data Framework
• oracle.nosql.driver.RetryableException – oracle.nosql.driver.SecurityInfoNotReadyException – oracle.nosql.driver.SystemException – oracle.nosql.driver.TableBusyException – oracle.nosql.driver.ThrottlingException – oracle.nosql.driver.OperationThrottlingException – oracle.nosql.driver.ReadThrottlingException – oracle.nosql.driver.WriteThrottlingException	TransientDataAccessResourceException
• oracle.nosql.driver.InvalidAuthorizationException • oracle.nosql.driver.UnauthorizedException	PermissionDeniedDataAccessException
• oracle.nosql.driver.IndexExistsException • oracle.nosql.driver.IndexNotFoundException • oracle.nosql.driver.JsonParseException • oracle.nosql.driver.OperationNotSupportedException • oracle.nosql.driver.RequestTimeoutException • oracle.nosql.driver.ResourceExistsException • oracle.nosql.driver.ResourceNotFoundException • oracle.nosql.driver.TableExistsException • oracle.nosql.driver.TableNotFoundException	InvalidDataAccessApiUsageException
• oracle.nosql.driver.ResourceLimitException – oracle.nosql.driver.BatchOperationNumberLimitException – oracle.nosql.driver.DeploymentException – oracle.nosql.driver.EvolutionLimitException – oracle.nosql.driver.IndexLimitException – oracle.nosql.driver.KeySizeLimitException – oracle.nosql.driver.RequestSizeLimitException – oracle.nosql.driver.RowSizeLimitException – oracle.nosql.driver.TableLimitException • oracle.nosql.driver.TableSizeException	InvalidDataAccessResourceUsageException

Copyright © 2011, 2026, Oracle and/or its affiliates.