NoSQL Database Python SDK Documentation

Oracle

Nov 16, 2022

Contents

This is the Python SDK for the Oracle NoSQL Database. Python 2.7+ and 3.5+ are supported.

For information about the Oracle NoSQL Database see https://www.oracle.com/database/technologies/related/nosql.html

CHAPTER 1

Installation

This topic describes how to install, configure, and use the Oracle NoSQL Database Python SDK. There are several supported environments:

1. Oracle NoSQL Database Cloud Service
2. Oracle NoSQL Database Cloud Simulator
3. Oracle NoSQL Database on-premise

1.1 Prerequisites

The Python SDK requires:

- Python version 2.7.5 or 3.5 or later, running on Mac, Windows, or Linux.
- For the Oracle NoSQL Cloud Service:
  - An Oracle Cloud Infrastructure account
  - A user created in that account, in a group with a policy that grants the desired permissions.
- For the Oracle NoSQL Database Cloud Simulator:
  - See Download the Oracle NoSQL Cloud Simulator to download and start the Cloud Simulator.
- For the on-premise Oracle NoSQL Database:
  - An instance of the database (See Oracle NoSQL Database Downloads)
  - A running proxy server, see Information about the proxy

1.2 Downloading and Installing the SDK

You can install the Python SDK through the Python Package Index (PyPI), or alternatively through GitHub.

1.2.1 PyPi

To install from PyPI use the following command:

```
pip install borneo
```

1.2.2 GitHub

To install from GitHub:

1. Download the SDK from GitHub. The download is a zip containing a whl file and documentation.
2. Extract the files from the zip.
3. Use the following command to install the SDK:

```
pip install borneo-*-py2.py3-none-any.whl
```

Note: If you're unable to install the whl file, make sure pip is up to date. Use `pip install -U pip` and then try to install the whl file again.

1.3 Configuring the SDK

This section describes configuring the SDK for the 3 environments supported. Skip to the section or sections of interest. The areas where the environments and use differ are

1. Authentication and authorization. This is encapsulated in the AuthorizationProvider interface. The Cloud Service is secure and requires a Cloud Service identity as well as authorization for desired operations. The Cloud Simulator is not secure at all and requires no identity. The on-premise configuration can be either secure or not and it also requires an instance of the proxy service to access the database.
2. API differences. Some classes and methods are specific to an environment. For example, the on-premise configuration includes methods to create namespaces and users and these concepts don't exist in the cloud service. Similarly, the cloud service includes interfaces to specify and acquire throughput information on tables that is not relevant on-premise.

1.3.1 Configure for the Cloud Service

The SDK requires an Oracle Cloud account and a subscription to the Oracle NoSQL Database Cloud Service. If you do not already have an Oracle Cloud account you can start here. Credentials used for connecting an application are associated with a specific user. If needed, create a user for the person or system using the api. See Adding Users.

Using the SDK with the Oracle NoSQL Database Cloud Service also requires installation of the Oracle Cloud Infrastructure (OCI) Python SDK:

```
pip install oci
```

Acquire Credentials for the Oracle NoSQL Database Cloud Service

These steps only need to be performed one time for a user. If they have already been done they can be skipped. You need to obtain the following credentials:

- Tenancy ID
- User ID
- API signing key (private key in PEM format)
- Private key pass phrase, only needed if the private key is encrypted
- Fingerprint for the public key uploaded to the user's account

See Required Keys and OCIDs for detailed descriptions of the above credentials and the steps you need to perform to obtain them. Specifically:

- How to Generate an API Signing Key
- How to Get the Key's Fingerprint
- How to Upload the Public Key
- Where to Get the Tenancy's OCID and User's OCID

Supplying Credentials to an Application

Credentials are used to establish the initial connection from your application to the service. There are 2 ways to supply credentials to the application:

1. Directly, via API
2. Using a configuration file

Both mechanisms use *borneo.iam.SignatureProvider* to handle credentials. If using a configuration file it's default location is *$HOME/.oci/config*, but the location can be changed using the api.

The format of the configuration file is that of a properties file with the format of *key=value*, with one property per line. The contents and format are:

```
[DEFAULT]
tenancy=<your-tenancy-id>
user=<your-user-id>
fingerprint=<fingerprint-of-your-public-key>
key_file=<path-to-your-private-key-file>
pass_phrase=<optional-pass-phrase-for-key-file>
```

The Tenancy ID, User ID and fingerprint should be acquired using the instructions above. The path to your private key file is the absolute path of the RSA private key. The order of the properties does not matter. The *[DEFAULT]* portion is the *profile*. A configuration file may contain multiple profiles with the target profile specified in the *borneo.iam.SignatureProvider* parameters.

Provide credentials without a configuration file:

```
from borneo.iam import SignatureProvider

#
# Use SignatureProvider directly via API. Note that the
# private_key argument can either point to a key file or be the
# string content of the private key itself.
#
at_provider = SignatureProvider(
    tenant_id='ocid1.tenancy.oc1..tenancy',
    user_id='ocid1.user.oc1..user',
    private_key=key_file_or_key,
```

(continues on next page)

(continued from previous page)

```
    fingerprint='fingerprint',
    pass_phrase='mypassphrase')
```

Provide credentials using a configuration file in the default location, using the default profile:

```
from borneo.iam import SignatureProvider

#
# Use SignatureProvider with a default credentials file and
# profile $HOME/.oci/config
#
at_provider = SignatureProvider()
```

Provide credentials using a configuration file in a non-default location and non-default profile:

```
from borneo.iam import SignatureProvider

#
# Use SignatureProvider with a non-default credentials file and
# profile
#
at_provider = SignatureProvider(config_file='myconfigfile',
    profile_name='myprofile')
```

Connecting an Application

The first step in any Oracle NoSQL Database Cloud Service application is to create a *handle* used to send requests to the service. The handle is configured using your credentials and other authentication information as well as the endpoint to which the application will connect. An example endpoint is to use the region **Regions.US_ASHBURN_1**. Information on regions can be found in *borneo.Regions*.

```
from borneo import NoSQLHandle, NoSQLHandleConfig, Regions
from borneo.iam import SignatureProvider

#
# Required information:
#

# the region to which the application will connect
region = Regions.US_ASHBURN_1

# if using a specified credentials file
credentials_file = <path-to-your-credentials-file>

#
# Create an AuthorizationProvider
#
at_provider = SignatureProvider(config_file=credentials_file)

#
# create a configuration object
#
config = NoSQLHandleConfig(region, at_provider)

#
```

(continues on next page)

(continued from previous page)

```
# create a handle from the configuration object
#
handle = NoSQLHandle(config)
```

See examples and test code for specific details. Both of these use config*.py files for configuration of required information.

1.3.2 Configure for the Cloud Simulator

The Oracle NoSQL Cloud Simulator is a useful way to use this SDK to connect to a local server that supports the same protocol. The Cloud Simulator requires Java 8 or higher.

See Download the Oracle NoSQL Cloud Simulator to download and start the Cloud Simulator.

1. Download and start the Cloud Simulator
2. Follow instructions in the examples/config.py file for connecting examples to the Cloud Simulator. By default that file is configured to communicate with the Cloud Simulator, using default configuration.

The Cloud Simulator does not require the credentials and authentication information required by the Oracle NoSQL Database Cloud Service. The Cloud Simulator should not be used for deploying applications or important data.

Before using the Cloud Service it is recommended that users start with the Cloud Simulator to become familiar with the interfaces supported by the SDK.

1.3.3 Configure for the On-Premise Oracle NoSQL Database

The on-premise configuration requires a running instance of the Oracle NoSQL database. In addition a running proxy service is required. See Oracle NoSQL Database Downloads for downloads, and see Information about the proxy for proxy configuration information.

If running a secure store, a certificate path should be specified through the REQUESTS_CA_BUNDLE environment variable:

$ export REQUESTS_CA_BUNDLE=<path-to-certificate>/certificate.pem:$REQUESTS_CA_BUNDLE

Or *borneo.NoSQLHandleConfig.set_ssl_ca_certs()*.

In addition, a user identity must be created in the store (separately) that has permission to perform the required operations of the application, such as manipulated tables and data. The identity is used in the *borneo.kv.StoreAccessTokenProvider*.

If the store is not secure, an empty instance of *borneo.kv.StoreAccessTokenProvider* is used. For example:

```
from borneo import NoSQLHandle, NoSQLHandleConfig
from borneo.kv import StoreAccessTokenProvider

#
# Assume the proxy is running on localhost:8080
#
endpoint = 'http://localhost:8080'

#
# Assume the proxy is secure and running on localhost:443
#
endpoint = 'https://localhost:443'
```

(continues on next page)

(continued from previous page)

```
#
# Create the AuthorizationProvider for a secure store:
#
ap = StoreAccessTokenProvider(user_name, password)

#
# Create the AuthorizationProvider for a not secure store:
#
ap = StoreAccessTokenProvider()

#
# create a configuration object
#
config = NoSQLHandleConfig(endpoint).set_authorization_provider(ap)

#
# set the certificate path if running a secure store
#
config.set_ssl_ca_certs(<ca_certs>)

#
# create a handle from the configuration object
#
handle = NoSQLHandle(config)
```

CHAPTER 2

Working With Tables

Applications using the Oracle NoSQL Database work with tables. Tables are created and data is added, modified and removed. Indexes can be added on tables. These topics are covered. Not all options and functions are described here. Detailed descriptions of interfaces can be found in *API Reference*.

2.1 Obtain a NoSQL Handle

borneo.NoSQLHandle represents a connection to the service. Once created it must be closed using the method *borneo.NoSQLHandle.close()* in order to clean up resources. Handles are thread-safe and intended to be shared. A handle is created by first creating a *borneo.NoSQLHandleConfig* instance to configure the communication endpoint, authorization information, as well as default values for handle configuration.

Configuration requires an *borneo.AuthorizationProvider* to provide identity and authorization information to the handle. There are different instances of this class for the different environments:

1. Oracle NoSQL Cloud Service
2. Oracle NoSQL Cloud Simulator
3. Oracle NoSQL Database on-premise

2.1.1 About Compartments

In the Oracle NoSQL Cloud Service environment tables are always created in an Oracle Cloud Infrastructure *compartment* (see Managing Compartments). It is recommended that compartments be created for tables to better organize them and control security, which is a feature of compartments. When authorized as a specific user the default compartment for tables is the root compartment of the user's tenancy. A method exists to allow specification of a default compartment for requests in `borneo.NoSQLHandleConfig.set_compartment()`. This overrides the user's default. In addition it is possible to specify a compartment is each `Request` instance.

The *set_compartment* methods take either an id (OCID) or a compartment name or path. If a compartment name is used it may be the name of a top-level compartment. If a compartment path is used to reference a nested com-

partment, the path is a dot-separate path that excludes the top-level compartment of the path, for example *compartmentA.compartmentB*.

Instead of setting a compartment in the request it is possible to use a compartment name to prefix a table name in a request, query, or DDL statement. This usage overrides any other setting of the compartment. E.g.

```
...
request = PutRequest().set_table_name('mycompartment:mytable')
...
create_statement = 'create table mycompartment:mytable(...)'
...
request = GetRequest().set_table_name('compartmentA.compartmentB')
```

If the application is authorized using an instance principal (see *borneo.iam.SignatureProvider.create_with_instance_principal()*) a compartment must be specified either using a default or in each request, and it **must** be specified as an id, as there is no default root compartment in this path.

An example of acquiring a NoSQL Handle for the Oracle NoSQL Cloud Service:

```
from borneo import NoSQLHandle, NoSQLHandleConfig, Regions
from borneo.iam import SignatureProvider

# create AuthorizationProvider
provider = SignatureProvider()

# create handle config using the correct desired region as endpoint, add a
# default compartment.
config = NoSQLHandleConfig(Regions.US_ASHBURN_1).set_authorization_provider(
    provider).set_default_compartment('mycompartment')

# create the handle
handle = NoSQLHandle(config)
```

An example using the on-premise Oracle NoSQL Database in a secure configuration, a certificate path should be specified through the REQUESTS_CA_BUNDLE environment variable:

$ export REQUESTS_CA_BUNDLE=<path-to-certificate>/certificate.pem:$REQUESTS_CA_BUNDLE

Or *borneo.NoSQLHandleConfig.set_ssl_ca_certs()*, for example:

```
from borneo import NoSQLHandle, NoSQLHandleConfig
from borneo.kv import StoreAccessTokenProvider

# create AuthorizationProvider
provider = StoreAccessTokenProvider(<user_name>, <password>)

# create handle config using the correct endpoint for the running proxy
config = NoSQLHandleConfig(
    'https://localhost:443').set_authorization_provider(
    provider).set_ssl_ca_certs(<ca_certs>)

# create the handle
handle = NoSQLHandle(config)
```

To reduce resource usage and overhead of handle creation it is best to avoid excessive creation and closing of *borneo.NoSQLHandle* instances.

2.2 Create Tables and Indexes

Learn how to create tables and indexes in Oracle NoSQL Database.

Creating a table is the first step of developing your application. You use the *borneo.TableRequest* class and its methods to execute Data Definition Language (DDL) statements, such as, creating, modifying, and dropping tables. If using the Oracle NoSQL Cloud Service or Cloud Simulator you must also set table limits using *borneo.TableRequest.set_table_limits()* method. Limits are ignored on-premise, if provided.

Before creating a table, learn about:

The supported data types for Oracle NoSQL Database. See Supported Data Types. Also see *Data Types* for a description of how database types map to Python.

For the Oracle NoSQL Database Cloud Service limits. See Oracle NoSQL Database Cloud Limits. These limits are not relevant on-premise.

Examples of DDL statements are:

```
# Create a new table called users
CREATE IF NOT EXISTS users (id INTEGER, name STRING, PRIMARY KEY (id))

# Create a new table called users and set the TTl value to 4 days
CREATE IF NOT EXISTS users (id INTEGER, name STRING, PRIMARY KEY (id))
USING TTL 4 days

# Create a new index called nameIdx on the name field in the users table
CREATE INDEX IF NOT EXISTS nameIdx ON users(name)
```

DDL statements are executing using the *borneo.TableRequest* class. All calls to *borneo.NoSQLHandle.table_request()* are asynchronous so it is necessary to check the result and call *borneo.TableResult.wait_for_completion()* to wait for operation to complete. The convenience method, *borneo.NoSQLHandle.do_table_request()*, exists to combine execution of the operation with waiting for completion.

```
from borneo import TableLimits, TableRequest

statement = 'create table if not exists users(id integer, name string, ' +
            'primary key(id)'

# In the Cloud Service TableLimits is a required object for table creation.
# It specifies the throughput and capacity for the table in ReadUnits,
# WriteUnits, GB
request = TableRequest().set_statement(statement).set_table_limits(
    TableLimits(20, 10, 5))

# assume that a handle has been created, as handle, make the request wait
# for 40 seconds, polling every 3 seconds
result = handle.do_table_request(request, 40000, 3000)

# the above call to do_table_request is equivalent to
result = handle.table_request(request)
result.wait_for_completion(handle, 40000, 3000)
```

2.3 Add Data

Add rows to your table.

When you store data in table rows, your application can easily retrieve, add to, or delete information from the table.

The *borneo.PutRequest* class represents input to the *borneo.NoSQLHandle.put()* method used to insert single rows. This method can be used for unconditional and conditional puts to:

- Overwrite any existing row. This is the default.
- Succeed only if the row does not exist. Use *borneo.PutOption.IF_ABSENT* for this case.
- Succeed only if the row exists. Use *borneo.PutOption.IF_PRESENT* for this case.
- Succeed only if the row exists and its *borneo.Version* matches a specific *borneo.Version*. Use *borneo.PutOption.IF_VERSION* for this case and *borneo.PutRequest.set_match_version()* to specify the version to match.

Options can be set using *borneo.PutRequest.set_option()*.

To add rows to your table:

```
from borneo import PutRequest

# PutRequest requires a table name
 request = PutRequest().set_table_name('users')

# set the value
request.set_value({'id': i, 'name': 'myname'})
result = handle.put(request)

# a successful put returns a non-empty version
if result.get_version() is not None:
   # success
```

When adding data the values supplied must accurately correspond to the schema for the table. If they do not, IllegalArgumentException is raised. Columns with default or nullable values can be left out without error, but it is recommended that values be provided for all columns to avoid unexpected defaults. By default, unexpected columns are ignored silently, and the value is put using the expected columns.

If you have multiple rows that share the same shard key they can be put in a single request using *borneo.WriteMultipleRequest* which can be created using a number of PutRequest or DeleteRequest objects.

You can also add JSON data to your table. In the case of a fixed-schema table the JSON is converted to the target schema. JSON data can be directly inserted into a column of type *JSON*. The use of the JSON data type allows you to create table data without a fixed schema, allowing more flexible use of the data.

2.3.1 Add JSON Data

The data value provided for a row or key is a Python *dict*. It can be supplied to the relevant requests (GetRequest, PutRequest, DeleteRequest) in multiple ways:

- as a Python dict directly:

```
request.set_value({'id': 1})
request.set_key({'id': 1 })
```

- as a JSON string:

```
request.set_value_from_json('{"id": 1, "name": "myname"}')
request.set_key_from_json('{"id": 1}')
```

In both cases the keys and values provided must accurately correspond to the schema of the table. If not an *borneo.IllegalArgumentException* exception is raised. If the data is provided as JSON and the JSON cannot be parsed a `ValueError` is raised.

2.4 Read Data

Learn how to read data from your table.

You can read single rows using the *borneo.NoSQLHandle.get()* method. This method allows you to retrieve a record based on its primary key value. In order to read multiple rows in a single request see *Use Queries*, below.

The *borneo.GetRequest* class is used for simple get operations. It contains the primary key value for the target row and returns an instance of *borneo.GetResult*.

```
from borneo import GetRequest

# GetRequest requires a table name
request = GetRequest().set_table_name('users')

# set the primary key to use
request.set_key({'id': 1})
result = handle.get(request)

# on success the value is not empty
if result.get_value() is not None:
   # success
```

By default all read operations are eventually consistent, using *borneo.Consistency.EVENTUAL*. This type of read is less costly than those using absolute consistency, *borneo.Consistency.ABSOLUTE*. This default can be changed in *borneo.NoSQLHandle* using *borneo.NoSQLHandleConfig.set_consistency()* before creating the handle. It can be changed for a single request using *borneo.GetRequest.set_consistency()*.

2.5 Use Queries

Learn about using queries in your application.

Oracle NoSQL Database provides a rich query language to read and update data. See the SQL For NoSQL Specification for a full description of the query language.

There are two ways to get the results of a query: using an iterator or loop through partial results.

2.5.1 Iterator

Use *borneo.NoSQLHandle.query_iterable()* to get an iterable that contains all the results of a query. Usage example:

```
from borneo import QueryRequest

handle = ...
```

(continues on next page)

(continued from previous page)

```
statement = 'select * from users where name = "Taylor"'
request = QueryRequest().set_statement(statement)
qiresult = handle.query_iterable(request)
for row in qiresult:
    # do something with the result row
    print(row)
```

2.5.2 Partial results

Another way is to loop through partial results by using the *borneo.NoSQLHandle.query()* method. For example, to execute a *SELECT* query to read data from your table, a *borneo.QueryResult* contains a list of results. And if the *borneo.QueryRequest.is_done()* returns False, there may be more results, so queries should generally be run in a loop. It is possible for single request to return no results but the query still not done, indicating that the query loop should continue. For example:

```
from borneo import QueryRequest

# Query at table named 'users" using the field 'name' where name may match 0
# or more rows in the table. The table name is inferred from the query
# statement
statement = 'select * from users where name = "Taylor"'
request = QueryRequest().set_statement(statement)
# loop until request is done, handling results as they arrive
while True:
    result = handle.query(request)
    # handle results
    handle_results(result) # do something with results
    if request.is_done():
        break
```

When using queries it is important to be aware of the following considerations:

- Oracle NoSQL Database provides the ability to prepare queries for execution and reuse. It is recommended that you use prepared queries when you run the same query for multiple times. When you use prepared queries, the execution is much more efficient than starting with a query string every time. The query language and API support query variables to assist with query reuse. See *borneo.NoSQLHandle.prepare()* and *borneo.PrepareRequest* for more information.
- The *borneo.QueryRequest* allows you to set the read consistency for a query as well as modifying the maximum amount of resource (read and write) to be used by a single request. This can be important to prevent a query from getting throttled because it uses too much resource too quickly.

Here is an example of using a prepared query with a single variable:

```
from borneo import PrepareRequest, QueryRequest

# Use a similar query to above but make the name a variable
statement = 'declare $name string; select * from users where name = $name'
prequest = PrepareRequest().set_statement(statement)
presult = handle.prepare(prequest)

# use the prepared statement, set the variable
pstatement = presult.get_prepared_statement()
pstatement.set_variable('$name', 'Taylor')
qrequest = QueryRequest().set_prepared_statement(pstatement)
```

(continues on next page)

(continued from previous page)

```
qiresult = handle.query_iterable(qrequest)
# loop on all the results
for row in qiresult:
    # do something with the result row
    print(row)


# use a different variable value with the same prepared query
pstatement.set_variable('$name', 'another_name')
qrequest = QueryRequest().set_prepared_statement(pstatement)
# loop until qrequest is done, handling results as they arrive
while True:
    # use the prepared query in the query request
    qresult = handle.query(qrequest)
    # handle results
    handle_results(qresult) # do something with results
    if qrequest.is_done():
        break
```

2.6 Delete Data

Learn how to delete rows from your table.

Single rows are deleted using *borneo.DeleteRequest* using a primary key value:

```
from borneo import DeleteRequest

# DeleteRequest requires table name and primary key
request = DeleteRequest().set_table_name('users')
request.set_key({'id': 1})

# perform the operation
result = handle.delete(request)
if result.get_success():
   # success -- the row was deleted

# if the row didn't exist or was not deleted for any other reason, False is
# returned
```

Delete operations can be conditional based on a *borneo.Version* returned from a get operation. See *borneo.DeleteRequest*.

You can perform multiple deletes in a single operation using a value range using *borneo.MultiDeleteRequest* and *borneo.NoSQLHandle.multi_delete()*.

2.7 Modify Tables

Learn how to modify tables. You modify a table to:

- Add or remove fields to an existing table
- Change the default TimeToLive (TTL) value for the table
- Modify table limits

Examples of DDL statements to modify a table are:

```
# Add a new field to the table
ALTER TABLE users (ADD age INTEGER)

# Drop an existing field from the table
ALTER TABLE users (DROP age)

# Modify the default TTl value
ALTER TABLE users USING TTL 4 days
```

If using the Oracle NoSQL Database Cloud Service table limits can be modified using *borneo.TableRequest.set_table_limits()*, for example:

```
from borneo import TableLimits, TableRequest

# in this path the table name is required, as there is no DDL statement
request = TableRequest().set_table_name('users')
request.set_table_limits(TableLimits(40, 10, 5))
result = handle.table_request(request)

# table_request is asynchronous, so wait for the operation to complete, wait
# for 40 seconds, polling every 3 seconds
result.wait_for_completion(handle, 40000, 3000)
```

2.8 Delete Tables and Indexes

Learn how to delete a table or index.

To drop a table or index, use the *drop table* or *drop index* DDL statement, for example:

```
# drop the table named users (implicitly drops any indexes on that table)
DROP TABLE users

# drop the index called nameIndex on the table users. Don't fail if the index
# doesn't exist
DROP INDEX IF EXISTS nameIndex ON users
```

```
from borneo import TableRequest

# the drop statement
statement = 'drop table users'
request = TableRequest().set_statement(statement)

# perform the operation, wait for 40 seconds, polling every 3 seconds
result = handle.do_table_request(request, 40000, 3000)
```

2.9 Handle Errors

Python errors are raised as exceptions defined as part of the API. They are all instances of Python's `RuntimeError`. Most exceptions are instances of *borneo.NoSQLException* which is a base class for exceptions raised by the Python driver.

Exceptions are split into 2 broad categories:

- Exceptions that may be retried with the expectation that they may succeed on retry. These are all instances of *borneo.RetryableException*. Examples of these are the instances of *borneo.ThrottlingException* which is raised when resource consumption limits are exceeded.
- Exceptions that should not be retried, as they will fail again. Examples of these include *borneo.IllegalArgumentException*, *borneo.TableNotFoundException*, etc.

borneo.ThrottlingException instances will never be thrown in an on-premise configuration as there are no relevant limits.

2.10 Handle Resource Limits

This section is relevant only to the Cloud Service and Simulator.

Programming in a resource-limited environment can be unfamiliar and can lead to unexpected errors. Tables have user-specified throughput limits and if an application exceeds those limits it may be throttled, which means requests will raise instances of *borneo.ThrottlingException*.

There is some support for built-in retries and users can create their own *borneo.RetryHandler* instances to be set using *borneo.NoSQLHandleConfig.set_retry_handler()* allowing more direct control over retries as well as tracing of throttling events. An application should not rely on retries to handle throttling exceptions as that will result in poor performance and an inability to use all of the throughput available for the table. This happens because the default retry handler will do exponential backoff, starting with a one-second delay.

While handling *borneo.ThrottlingException* is necessary it is best to avoid throttling entirely by rate-limiting your application. In this context *rate-limiting* means keeping request rates under the limits for the table. This is most common using queries, which can read a lot of data, using up capacity very quickly. It can also happen for get and put operations that run in a tight loop. Some tools to control your request rate include:

- use the methods available in all Result objects that indicate how much read and write throughput was used by that request. For example, see *borneo.GetResult.get_read_units()* or *borneo.PutResult.get_write_units()*.
- reduce the default amount of data read for a single query request by using *borneo.QueryRequest.set_max_read_kb()*. Remember to perform query operations in a loop, looking at the continuation key. Be aware that a single query request can return 0 results but still have a continuation key that means you need to keep looping.
- add rate-limiting code in your request loop. This may be as simple as a delay between requests or intelligent code that considers how much data has been read (see *borneo.QueryResult.get_read_units()*) as well as the capacity of the table to either delay a request or reduce the amount of data to be read.

CHAPTER 3

Data Types

This topic describes the mapping between types in the Oracle NoSQL Database and Python data types. The database types are referred to as *database types* while the Python equivalents are *Python types*.

3.1 Oracle NoSQL Database Types

See Supported Data Types for a description of the data types supported by the service. An application uses these types to create tables and indexes. For example, a table may be created using this Data Definition Language (DDL) statement, which defines types in terms of the database types:

```
create table mytable(id integer, name string, created timestamp,
    address record(street string, city string, zip integer), primary key(id))
```

In order to insert rows into such a table your application must create a Python dict that corresponds to that schema, for example:

```
{'id': 1, 'name': 'myname', 'created': datetime.now(),
 'address' : {'street' : '14 Elm Street', 'city' : "hometown',
 'zip' : 00000}}
```

Similarly, when operating on rows retrieved from the database it is important to understand the mappings to Python types.

3.2 Mapping Between Database and Python types

These mappings apply on both input (get/query) and output (put). In general the system is permissive in terms of valid conversions among types and that any lossless conversion is allowed. For example an integer will be accepted for a float or double database type. The *Timestamp* type is also flexible and will accept any valid IS0 8601 formatted string. Timestamps are always stored and managed in UTC.

Database Type	Python Type
Integer	int
Long	int (Python 3), long (Python2)
Float	float
Double	float
Number	decimal.Decimal
Boolean	bool
String	str
Timestamp	datetime.datetime
Enum	str
Binary	bytearray
FixedBinary	bytearray
Array	list
Map	dict
Record	dict
JSON	any valid JSON

3.3 Timestamp in Borneo

As mentioned above *Timestamp* fields are managed internally as UTC time. If a timezone is supplied when setting a *Timestamp*, either as a string or as a Python datetime object, it will be honored. The value will be converted to UTC internally and will be in UTC when returned in a row. Although they are represented in UTC returned datetime objects will be "naive" as described by Python documentation. On input, if no timezone is supplied, python datetime instances and time strings are treated as UTC.

CHAPTER 4

API Reference

4.1 borneo Package

4.1.1 Classes

AuthorizationProvider	AuthorizationProvider is a callback interface used by the driver to obtain an authorization string for a request.
BatchOperationNumberLimitException(message)	Cloud service only.
Consistency	Set the consistency for read requests.
Durability(master_sync, replica_sync, ...)	Durability defines the durability characteristics associated with a standalone write (put or update) operation.
DefaultRetryHandler([retries, delay_s])	Default retry handler.
DeleteRequest()	Represents the input to a *NoSQLHandle.delete()* operation.
DeleteResult()	Represents the result of a *NoSQLHandle.delete()* operation.
FieldRange(field_path)	FieldRange defines a range of values to be used in a *NoSQLHandle.multi_delete()* operation, as specified in *MultiDeleteRequest.set_range()*.
GetIndexesRequest()	Represents the argument of a *NoSQLHandle.get_indexes()* operation which returns the information of a specific index or all indexes of the specified table, as returned in *GetIndexesResult*.
GetIndexesResult()	Represents the result of a *NoSQLHandle.get_indexes()* operation.
GetRequest()	Represents the input to a *NoSQLHandle.get()* operation which returns a single row based on the specified key.

Continued on next page

Table 1 – continued from previous page

GetResult()	Represents the result of a *NoSQLHandle.get()* operation.
GetTableRequest()	Represents the argument of a *NoSQLHandle.get_table()* operation which returns static information associated with a table, as returned in *TableResult*.
IllegalArgumentException([message, cause])	Exception class that is used when an invalid argument was passed, this could mean that the type is not the expected or the value is not valid for the specific case.
IllegalStateException([message, cause])	Exception that is thrown when a method has been invoked at an illegal or inappropriate time.
IndexExistsException(message)	The operation attempted to create an index for a table but the named index already exists.
IndexInfo(index_name, field_names)	IndexInfo represents the information about a single index including its name and field names.
IndexNotFoundException(message)	The operation attempted to access a index that does not exist or is not in a visible state.
InvalidAuthorizationException(message)	The exception is thrown if the application presents an invalid authorization string in a request.
ListTablesRequest()	Represents the argument of a *NoSQLHandle.list_tables()* operation which lists all available tables associated with the identity associated with the handle used for the operation.
ListTablesResult()	Represents the result of a *NoSQLHandle.list_tables()* operation.
MultiDeleteRequest()	Represents the input to a *NoSQLHandle.multi_delete()* operation which can be used to delete a range of values that match the primary key and range provided.
MultiDeleteResult()	Represents the result of a *NoSQLHandle.multi_delete()* operation.
NoSQLException(message[, cause])	A base class for most exceptions thrown by the NoSQL driver.
NoSQLHandle(config)	NoSQLHandle is a handle that can be used to access Oracle NoSQL tables.
NoSQLHandleConfig([endpoint, provider])	An instance of this class is required by *NoSQLHandle*.
OperationNotSupportedException(message)	The operation attempted is not supported.
OperationResult()	A single Result associated with the execution of an individual operation in a *NoSQLHandle.write_multiple()* request.
OperationThrottlingException(message)	Cloud service only.
PreparedStatement(sql_text, query_plan, ...)	A class encapsulating a prepared query statement.
PrepareRequest()	A request that encapsulates a query prepare call.
PrepareResult()	The result of a prepare operation.
PutOption	Set the put option for put requests.
PutRequest()	Represents the input to a *NoSQLHandle.put()* operation.
PutResult()	Represents the result of a *NoSQLHandle.put()* operation.
QueryRequest()	A request that represents a query.

Continued on next page

Table 1 – continued from previous page

QueryResult(request[, computed])	QueryResult comprises a list of dict instances representing the query results.
QueryIterableResult(request, handle)	QueryIterableResult comprises an iterable list of dict instances representing all the query results.
ReadThrottlingException(message)	Cloud service only.
Region(region_id)	Cloud service only.
Regions	Cloud service only.
Request()	A request is a class used as a base for all requests types.
RequestSizeLimitException(message)	Cloud service only.
RequestTimeoutException(message[, ...])	Thrown when a request cannot be processed because the configured timeout interval is exceeded.
ResourceExistsException(message)	The operation attempted to create a resource but it already exists.
ResourcePrincipalClaimKeys	Claim keys in the resource principal session token(RPST).
ResourceNotFoundException(message)	The operation attempted to access a resource that does not exist or is not in a visible state.
Result()	Result is a base class for result classes for all supported operations.
RetryHandler	RetryHandler is called by the request handling system when a *RetryableException* is thrown.
RetryableException(message)	A base class for all exceptions that may be retried with a reasonable expectation that they may succeed on retry.
SecurityInfoNotReadyException(message)	Cloud service only.
State	Represents the table state.
StatsControl(config, logger, ...)	StatsControl allows user to control the collection of driver statistics at
StatsProfile	The following semantics are attached to the StatsProfile values:
SystemException(message)	An exception that is thrown when there is an internal system problem.
SystemRequest()	On-premise only.
SystemResult()	On-premise only.
SystemState	On-premise only.
SystemStatusRequest()	On-premise only.
TableExistsException(message)	The operation attempted to create a table but the named table already exists.
TableLimits(read_units, write_units, storage_gb)	Cloud service only.
TableNotFoundException(message)	The operation attempted to access a table that does not exist or is not in a visible state.
TableRequest()	TableRequest is used to create, modify, and drop tables.
TableResult()	TableResult is returned from *NoSQLHandle.get_table()* and *NoSQLHandle.table_request()* operations.
TableUsageRequest()	Cloud service only.
TableUsageResult()	Cloud service only.
ThrottlingException(message)	Cloud service only.
TimeToLive(value, timeunit)	TimeToLive is a utility class that represents a period of time, similar to java.time.Duration in Java, but specialized to the needs of this driver.
TimeUnit	The time unit to use.

Continued on next page

Table 1 – continued from previous page

UserInfo(user_id, user_name)	On-premise only.
Version(version)	Version is an opaque class that represents the version of a row in the database.
WriteMultipleRequest()	Represents the input to a *NoSQLHandle.write_multiple()* operation.
WriteMultipleResult()	Represents the result of a *NoSQLHandle.write_multiple()* operation.
WriteThrottlingException(message)	Cloud service only.

AuthorizationProvider

class borneo.**AuthorizationProvider**

Bases: object

AuthorizationProvider is a callback interface used by the driver to obtain an authorization string for a request. It is called when an authorization string is required. In general applications need not implement this interface, instead using the default mechanisms.

Instances of this interface must be reentrant and thread-safe.

Methods Summary

close()	Closes the authorization provider and releases any resources it may be using.
get_authorization_string([request])	Returns an authorization string for the specified request.
get_logger()	Returns the logger of this provider if set, None if not.
set_logger(logger)	Sets a logger instance for this provider.

Methods Documentation

close()

Closes the authorization provider and releases any resources it may be using.

get_authorization_string(*request=None*)

Returns an authorization string for the specified request. The string is sent to the server in the request and is used for authorization. Authorization information can be request-dependent.

Parameters **request** (*Request*) – the request to be issued. This is an instance of *Request()*.

Returns a string indicating that the application is authorized to perform the request.

Return type str

get_logger()

Returns the logger of this provider if set, None if not.

Returns the logger.

Return type Logger or None

set_logger(*logger*)

Sets a logger instance for this provider. If not set, the logger associated with the driver is used.

Parameters **logger** (*Logger*) – the logger to use.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if logger is not an instance of Logger.

BatchOperationNumberLimitException

exception borneo.**BatchOperationNumberLimitException**(*message*)

Cloud service only.

Thrown to indicate that the number of operations included in *NoSQLHandle.write_multiple()* operation exceeds the system defined limit.

Consistency

class borneo.**Consistency**

Bases: object

Set the consistency for read requests.

Attributes Summary

ABSOLUTE	Set Consistency.ABSOLUTE to use absolute consistency for read requests.
EVENTUAL	Set Consistency.EVENTUAL to use eventual consistency for read requests.

Attributes Documentation

ABSOLUTE = 0

Set Consistency.ABSOLUTE to use absolute consistency for read requests.

EVENTUAL = 1

Set Consistency.EVENTUAL to use eventual consistency for read requests. This is the default value for operations.

Durability

class borneo.**Durability**(*master_sync*, *replica_sync*, *replica_ack*)

Bases: object

Durability defines the durability characteristics associated with a standalone write (put or update) operation.

This is currently only supported in On-Prem installations. It is ignored in the cloud service.

The overall durability is a function of the SYNC_POLICY and ACK_POLICY in effect for the Master, and the SYNC_POLICY in effect for each Replica.

SYNC_POLICY represents policies to be used when committing a transaction. High levels of synchronization offer a greater guarantee that the transaction is persistent to disk, but trade that off for lower performance. The possible SYNC_POLICY values are:

- SYNC writes and synchronously flushes the log on transaction commit. Transactions exhibit all the ACID (atomicity, consistency, isolation, and durability) properties.

- NO_SYNC does not write or synchronously flush the log on transaction commit. Transactions exhibit the ACI (atomicity, consistency, and isolation) properties, but not D (durability); that is, database integrity will be maintained, but if the application or system fails, it is possible some number of the most recently committed transactions may be undone during recovery. The number of transactions at risk is governed by how many log updates can fit into the log buffer, how often the operating system flushes dirty buffers to disk, and how often log checkpoints occur.
- WRITE_NO_SYNC writes but does not synchronously flush the log on transaction commit. Transactions exhibit the ACI (atomicity, consistency, and isolation) properties, but not D (durability); that is, database integrity will be maintained, but if the operating system fails, it is possible some number of the most recently committed transactions may be undone during recovery. The number of transactions at risk is governed by how often the operating system flushes dirty buffers to disk, and how often log checkpoints occur.

REPLICA_ACK_POLICY defines the policy for how replicated commits are handled. A replicated environment makes it possible to increase an application's transaction commit guarantees by committing changes to its replicas on the network.

Possible REPLICA_ACK_POLICY values include:

- ALL defines that all replicas must acknowledge that they have committed the transaction. This policy should be selected only if your replication group has a small number of replicas, and those replicas are on extremely reliable networks and servers.
- NONE defines that no transaction commit acknowledgments are required and the master will never wait for replica acknowledgments. In this case, transaction durability is determined entirely by the type of commit that is being performed on the master.
- SIMPLE_MAJORITY defines that a simple majority of replicas must acknowledge that they have committed the transaction. This acknowledgment policy, in conjunction with an election policy which requires at least a simple majority, ensures that the changes made by the transaction remains durable if a new election is held.

The default Durability is configured in the proxy server with which this SDK communicates. It is an optional startup parameter.

Methods Documentation

__init__ (*master_sync*, *replica_sync*, *replica_ack*)
Create a Durability object

Parameters

- **master_sync** (*SYNC_POLICY*) – the master sync policy
- **replica_sync** (*SYNC_POLICY*) – the replica sync policy
- **replica_ack** (*REPLICA_ACK_POLICY*) – the replica ack policy

Attributes Documentation

REPLICA_ACK_POLICY
REPLICA_ACK_POLICY

alias of `Enum`

SYNC_POLICY
SYNC_POLICY

alias of `Enum`

DefaultRetryHandler

class borneo.**DefaultRetryHandler**(*retries=10*, *delay_s=0*)
Bases: borneo.config.RetryHandler

Default retry handler. It's a default instance of *RetryHandler* This may be extended by clients for specific use cases.

The default retry handler decides when and for how long retries will be attempted. See *RetryHandler* for more information on retry handlers.

Methods Summary

delay(request, num_retried, re)	Delay (sleep) during retry cycle.
do_retry(request, num_retried, re)	Decide whether to retry or not.
get_num_retries()	Returns the number of retries that this handler instance will allow before the exception is thrown to the application.

Methods Documentation

delay(*request*, *num_retried*, *re*)
Delay (sleep) during retry cycle. If delay_ms is non-zero, use it. Otherwise, use an incremental backoff algorithm to compute the time of delay.

do_retry(*request*, *num_retried*, *re*)
Decide whether to retry or not. Default behavior is to *not* retry OperationThrottlingException because the retry time is likely much longer than normal because they are DDL operations. In addition, *not* retry any requests that should not be retried: TableRequest, ListTablesRequest, GetTableRequest, TableUsageRequest, GetIndexesRequest.

get_num_retries()
Returns the number of retries that this handler instance will allow before the exception is thrown to the application.

Returns the max number of retries.

Return type int

DeleteRequest

class borneo.**DeleteRequest**
Bases: borneo.operations.WriteRequest

Represents the input to a *NoSQLHandle.delete()* operation.

This request can be used to perform unconditional and conditional deletes:

- Delete any existing row. This is the default.
- Succeed only if the row exists and and its *Version* matches a specific *Version*. Use *set_match_version()* for this case. Using this option in conjunction with using *set_return_row()* allows information about the existing row to be returned if the operation fails because of a version mismatch. On success no information is returned.

Using *set_return_row()* may incur additional cost and affect operation latency.

The table name and key are required parameters. On a successful operation *DeleteResult.get_success()* returns True. Additional information, such as previous row information, may be available in *DeleteResult*.

Methods Summary

get_compartment()	Cloud service only.
get_durability()	On-premise only.
get_key()	Returns the key of the row to be deleted.
get_match_version()	Returns the *Version* used for a match on a conditional delete.
get_return_row()	Returns whether information about the existing row should be returned on failure because of a version mismatch.
get_table_name()	Returns the table name to use for the operation.
get_timeout()	Returns the timeout to use for the operation, in milliseconds.
set_compartment(compartment)	Cloud service only.
set_durability(durability)	On-premise only.
set_key(key)	Sets the key to use for the delete operation.
set_key_from_json(json_key)	Sets the key to use for the delete operation based on a JSON string.
set_match_version(version)	Sets the *Version* to use for a conditional delete operation.
set_return_row(return_row)	Sets whether information about the existing row should be returned on failure because of a version mismatch.
set_table_name(table_name)	Sets the table name to use for the operation.
set_timeout(timeout_ms)	Sets the optional request timeout value, in milliseconds.

Methods Documentation

get_compartment()
Cloud service only.

Get the compartment id or name if set for the request.

Returns compartment id or name if set for the request, otherwise None if not set.

Return type str

get_durability()
On-premise only. Gets the durability to use for the operation or None if not set :returns: the Durability :versionadded: 5.3.0

get_key()
Returns the key of the row to be deleted.

Returns the key value, or None if not set.

Return type dict

get_match_version()
Returns the *Version* used for a match on a conditional delete.

Returns the Version or None if not set.

Return type *Version*

get_return_row()
Returns whether information about the existing row should be returned on failure because of a version mismatch.

Returns True if information should be returned.

Return type bool

get_table_name()
Returns the table name to use for the operation.

Returns the table name, or None if not set.

Returns str

get_timeout()
Returns the timeout to use for the operation, in milliseconds. A value of 0 indicates that the timeout has not been set.

Returns the timeout value.

Return type int

set_compartment(*compartment*)
Cloud service only.

Sets the name or id of a compartment to be used for this operation.

The compartment may be specified as either a name (or path for nested compartments) or as an id (OCID). A name (vs id) can only be used when authenticated using a specific user identity. It is *not* available if authenticated as an Instance Principal which can be done when calling the service from a compute instance in the Oracle Cloud Infrastructure. See *borneo.iam.SignatureProvider.create_with_instance_principal()*.

Parameters **compartment** (*str*) – the compartment name or id. If using a nested compartment, specify the full compartment path compartmentA.compartmentB, but exclude the name of the root compartment (tenant).

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if compartment is not a str.

set_durability(*durability*)
On-premise only. Sets the durability to use for the operation.

Parameters **durability** (Durability) – the Durability to use

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if Durability is not valid

Versionadded 5.3.0

set_key(*key*)
Sets the key to use for the delete operation. This is a required field.

Parameters **key** (*dict*) – the key value.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if key is not a dictionary.

set_key_from_json(*json_key*)
Sets the key to use for the delete operation based on a JSON string. The string is parsed for validity and stored internally as a dict.

Parameters **json_key** (*str*) – the key as a JSON string.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if json_key is not a string.

set_match_version(*version*)
Sets the *Version* to use for a conditional delete operation. The Version is usually obtained from *GetResult.get_version()* or other method that returns a Version. When set, the delete operation will succeed only if the row exists and its Version matches the one specified. Using this option will incur additional cost.

Parameters **version** (Version) – the *Version* to match.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if version is not an instance of Version.

set_return_row(*return_row*)
Sets whether information about the existing row should be returned on failure because of a version mismatch. If a match version has not been set via *set_match_version()* this parameter is ignored and there will be no return information. This parameter is optional and defaults to False. It's use may incur additional cost.

Parameters **return_row** (*bool*) – set to True if information should be returned.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if return_row is not True or False.

set_table_name(*table_name*)
Sets the table name to use for the operation. This is a required parameter.

Parameters **table_name** (*str*) – the table name.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if table_name is not a string.

set_timeout(*timeout_ms*)
Sets the optional request timeout value, in milliseconds. This overrides any default value set in *NoSQLHandleConfig*. The value must be positive.

Parameters **timeout_ms** (*int*) – the timeout value, in milliseconds.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if the timeout value is less than or equal to 0.

DeleteResult

class borneo.**DeleteResult**
Bases: borneo.operations.WriteResult

Represents the result of a *NoSQLHandle.delete()* operation.

If the delete succeeded *get_success()* returns True. Information about the existing row on failure may be available using *get_existing_value()* and *get_existing_version()*, depending on the use of *DeleteRequest.set_return_row()*.

Methods Summary

get_existing_modification_time()	Returns the existing row modification time if available.
get_existing_value()	Returns the existing row value if available.
get_existing_version()	Returns the existing row *Version* if available.
get_read_kb()	Returns the read throughput consumed by this operation, in KBytes.
get_read_units()	Returns the read throughput consumed by this operation, in read units.
get_success()	Returns True if the delete operation succeeded.
get_write_kb()	Returns the write throughput consumed by this operation, in KBytes.
get_write_units()	Returns the write throughput consumed by this operation, in write units.

Methods Documentation

get_existing_modification_time()
Returns the existing row modification time if available. It will be available if the target row exists and the operation failed because of a *Version* mismatch and the corresponding *DeleteRequest* the method *DeleteRequest.set_return_row()* was called with a True value.

Returns the modification time in milliseconds since January 1, 1970

Return type int

Versionadded 5.3.0

get_existing_value()
Returns the existing row value if available. It will be available if the target row exists and the operation failed because of a *Version* mismatch and the corresponding *DeleteRequest* the method *DeleteRequest.set_return_row()* was called with a True value.

Returns the value.

Return type dict

get_existing_version()
Returns the existing row *Version* if available. It will be available if the target row exists and the operation failed because of a *Version* mismatch and the corresponding *DeleteRequest* the method *DeleteRequest.set_return_row()* was called with a True value.

Returns the version.

Return type *Version*

get_read_kb()
Returns the read throughput consumed by this operation, in KBytes. This is the actual amount of data read by the operation. The number of read units consumed is returned by *get_read_units()* which may be a larger number because this was an update operation.

Returns the read KBytes consumed.

Return type int

get_read_units()
Returns the read throughput consumed by this operation, in read units. This number may be larger than that returned by *get_read_kb()* because it was an update operation.

Returns the read units consumed.

Return type int

get_success()
Returns True if the delete operation succeeded.

Returns True if the operation succeeded.

Return type bool

get_write_kb()
Returns the write throughput consumed by this operation, in KBytes.

Returns the write KBytes consumed.

Return type int

get_write_units()
Returns the write throughput consumed by this operation, in write units.

Returns the write units consumed.

Return type int

FieldRange

class borneo.**FieldRange**(*field_path*)
Bases: object

FieldRange defines a range of values to be used in a *NoSQLHandle.multi_delete()* operation, as specified in *MultiDeleteRequest.set_range()*. FieldRange is only relevant if a primary key has multiple components because all values in the range must share the same shard key.

FieldRange is used as the least significant component in a partially specified key value in order to create a value range for an operation that returns multiple rows or keys. The data types supported by FieldRange are limited to the atomic types which are valid for primary keys.

The least significant component of a key is the first component of the key that is not fully specified. For example, if the primary key for a table is defined as the tuple (a, b, c), a FieldRange can be specified for "a" if the primary key supplied is empty. A FieldRange can be specified for "b" if the primary key supplied to the operation has a concrete value for "a" but not for "b" or "c".

This object is used to scope a *NoSQLHandle.multi_delete()* operation. The field_path specified must name a field in a table's primary key. The values used must be of the same type and that type must match the type of the field specified.

Validation of this object is performed when is it used in an operation. Validation includes verifying that the field is in the required key and, in the case of a composite key, that the field is in the proper order relative to the key used in the operation.

Parameters **field_path** (*str*) – the path to the field used in the range.

Raises ***IllegalArgumentException*** – raises the exception if field_path is not a string.

Methods Summary

get_end()	Returns the field value that defines upper bound of the range, or None if no upper bound is enforced.
get_end_inclusive()	Returns whether end is included in the range, i.e., end is greater than or equal to the last field value in the range.
get_field_path()	Returns the name for the field used in the range.
get_start()	Returns the field value that defines lower bound of the range, or None if no lower bound is enforced.
get_start_inclusive()	Returns whether start is included in the range, i.e., start is less than or equal to the first field value in the range.
set_end(value, is_inclusive)	Sets the end value of the range to the specified value.
set_start(value, is_inclusive)	Sets the start value of the range to the specified value.

Methods Documentation

get_end()
Returns the field value that defines upper bound of the range, or None if no upper bound is enforced.

Returns the end field value.

get_end_inclusive()
Returns whether end is included in the range, i.e., end is greater than or equal to the last field value in the range. This value is valid only if the end value is not None.

Returns True if the end value is inclusive.

Return type bool

get_field_path()
Returns the name for the field used in the range.

Returns the name of the field.

Return type str

get_start()
Returns the field value that defines lower bound of the range, or None if no lower bound is enforced.

Returns the start field value.

get_start_inclusive()
Returns whether start is included in the range, i.e., start is less than or equal to the first field value in the range. This value is valid only if the start value is not None.

Returns True if the start value is inclusive.

Return type bool

set_end(*value*, *is_inclusive*)
Sets the end value of the range to the specified value.

Parameters

- **value** (*any*) – the value to set.
- **is_inclusive** (*bool*) – set to True if the range is inclusive of the value, False if it is exclusive.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if parameters are not expected type.

set_start(*value*, *is_inclusive*)
Sets the start value of the range to the specified value.

Parameters

- **value** (*any*) – the value to set.
- **is_inclusive** (*bool*) – set to True if the range is inclusive of the value, False if it is exclusive.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if parameters are not expected type.

GetIndexesRequest

class borneo.**GetIndexesRequest**
Bases: borneo.operations.Request

Represents the argument of a *NoSQLHandle.get_indexes()* operation which returns the information of a specific index or all indexes of the specified table, as returned in *GetIndexesResult*.

The table name is a required parameter.

Methods Summary

get_compartment()	Cloud service only.
get_index_name()	Gets the index name to use for the request.
get_table_name()	Returns the table name to use for the operation.
get_timeout()	Returns the timeout to use for the operation, in milliseconds.
set_compartment(compartment)	Cloud service only.
set_index_name(index_name)	Sets the index name to use for the request.
set_table_name(table_name)	Sets the table name to use for the request.
set_timeout(timeout_ms)	Sets the request timeout value, in milliseconds.

Methods Documentation

get_compartment()
Cloud service only.

Get the compartment id or name if set for the request.

Returns compartment id or name if set for the request, otherwise None if not set.

Return type str

get_index_name()
Gets the index name to use for the request.

Returns the index name.

Return type str

get_table_name()
Returns the table name to use for the operation.

Returns the table name, or None if not set.

Returns str

get_timeout()
Returns the timeout to use for the operation, in milliseconds. A value of 0 indicates that the timeout has not been set.

Returns the timeout value.

Return type int

set_compartment(*compartment*)
Cloud service only.

Sets the name or id of a compartment to be used for this operation.

The compartment may be specified as either a name (or path for nested compartments) or as an id (OCID). A name (vs id) can only be used when authenticated using a specific user identity. It is *not* available if authenticated as an Instance Principal which can be done when calling the service from a compute instance in the Oracle Cloud Infrastructure. See *borneo.iam.SignatureProvider.create_with_instance_principal()*.

Parameters **compartment** (*str*) – the compartment name or id. If using a nested compartment, specify the full compartment path compartmentA.compartmentB, but exclude the name of the root compartment (tenant).

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if compartment is not a str.

set_index_name(*index_name*)
Sets the index name to use for the request. If not set, this request will return all indexes of the table.

Parameters **index_name** (*str*) – the index name.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if index_name is not a string.

set_table_name(*table_name*)
Sets the table name to use for the request.

Parameters **table_name** (*str*) – the table name. This is a required parameter.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if table_name is not a string.

set_timeout(*timeout_ms*)
Sets the request timeout value, in milliseconds. This overrides any default value set in *NoSQLHandleConfig*. The value must be positive.

Parameters **timeout_ms** (*int*) – the timeout value, in milliseconds.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if the timeout value is less than or equal to 0.

GetIndexesResult

class borneo.**GetIndexesResult**
Bases: borneo.operations.Result

Represents the result of a *NoSQLHandle.get_indexes()* operation.

On a successful operation the index information is returned in a list of IndexInfo.

Methods Summary

get_indexes()	Returns the list of index information returned by the operation.

Methods Documentation

get_indexes()
Returns the list of index information returned by the operation.

Returns the indexes information.

Return type list(*IndexInfo*)

GetRequest

class borneo.**GetRequest**
Bases: borneo.operations.ReadRequest

Represents the input to a *NoSQLHandle.get()* operation which returns a single row based on the specified key.

The table name and key are required parameters.

Methods Summary

get_compartment()	Cloud service only.
get_key()	Returns the primary key used for the operation.
get_timeout()	Returns the timeout to use for the operation, in milliseconds.
set_consistency(consistency)	Sets the consistency to use for the operation.
set_compartment(compartment)	Cloud service only.
set_key(key)	Sets the primary key used for the get operation.
set_key_from_json(json_key)	Sets the key to use for the get operation based on a JSON string.
set_table_name(table_name)	Sets the table name to use for the operation.
set_timeout(timeout_ms)	Sets the request timeout value, in milliseconds.

Methods Documentation

get_compartment()
Cloud service only.

Get the compartment id or name if set for the request.

Returns compartment id or name if set for the request, otherwise None if not set.

Return type str

get_key()
Returns the primary key used for the operation. This is a required parameter.

Returns the key.

Return type dict

get_timeout()
Returns the timeout to use for the operation, in milliseconds. A value of 0 indicates that the timeout has not been set.

Returns the timeout value.

Return type int

set_consistency(*consistency*)
Sets the consistency to use for the operation. This parameter is optional and if not set the default consistency configured for the *NoSQLHandle* is used.

Parameters **consistency** (*Consistency*) – the consistency.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if consistency is not Consistency.ABSOLUTE or Consistency.EVENTUAL.

set_compartment(*compartment*)
Cloud service only.

Sets the name or id of a compartment to be used for this operation.

The compartment may be specified as either a name (or path for nested compartments) or as an id (OCID). A name (vs id) can only be used when authenticated using a specific user identity. It is *not* available if authenticated as an Instance Principal which can be done when calling the service from a compute instance in the Oracle Cloud Infrastructure. See *borneo.iam.SignatureProvider.create_with_instance_principal()*.

Parameters **compartment** (*str*) – the compartment name or id. If using a nested compartment, specify the full compartment path compartmentA.compartmentB, but exclude the name of the root compartment (tenant).

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if compartment is not a str.

set_key(*key*)
Sets the primary key used for the get operation. This is a required parameter.

Parameters **key** (*dict*) – the primary key.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if key is not a dictionary.

set_key_from_json(*json_key*)
Sets the key to use for the get operation based on a JSON string.

Parameters **json_key** (*str*) – the key as a JSON string.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if json_key is not a string.

set_table_name(*table_name*)
Sets the table name to use for the operation. This is a required parameter.

Parameters **table_name** (*str*) – the table name.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if table_name is not a string.

set_timeout(*timeout_ms*)
Sets the request timeout value, in milliseconds. This overrides any default value set in *NoSQLHandleConfig*. The value must be positive.

Parameters **timeout_ms** (*int*) – the timeout value, in milliseconds.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if the timeout value is less than or equal to 0.

GetResult

class borneo.**GetResult**
Bases: borneo.operations.Result

Represents the result of a *NoSQLHandle.get()* operation.

On a successful operation the value of the row is available using *get_value()* and the other state available in this class is valid. On failure that value is None and other state, other than consumed capacity, is undefined.

Methods Summary

get_expiration_time()	Returns the expiration time of the row.
get_read_kb()	Returns the read throughput consumed by this operation, in KBytes.
get_read_units()	Returns the read throughput consumed by this operation, in read units.
get_value()	Returns the value of the returned row, or None if the row does not exist.
get_version()	Returns the *Version* of the row if the operation was successful, or None if the row does not exist.
get_write_kb()	Returns the write throughput consumed by this operation, in KBytes.
get_write_units()	Returns the write throughput consumed by this operation, in write units.

Methods Documentation

get_expiration_time()
Returns the expiration time of the row. A zero value indicates that the row does not expire. This value is valid only if the operation successfully returned a row (*get_value()* returns non-none).

Returns the expiration time in milliseconds since January 1, 1970, or zero if the row never expires or the row does not exist.

Return type int

get_read_kb()
Returns the read throughput consumed by this operation, in KBytes. This is the actual amount of data read by the operation. The number of read units consumed is returned by *get_read_units()* which may be a larger number if the operation used Consistency.ABSOLUTE.

Returns the read KBytes consumed.

Return type int

get_read_units()
Returns the read throughput consumed by this operation, in read units. This number may be larger than that returned by *get_read_kb()* if the operation used Consistency.ABSOLUTE.

Returns the read units consumed.

Return type int

get_value()
Returns the value of the returned row, or None if the row does not exist.

Returns the value of the row, or None if it does not exist.

Return type dict

get_version()
Returns the *Version* of the row if the operation was successful, or None if the row does not exist.

Returns the version of the row, or None if the row does not exist.

Return type *Version*

get_write_kb()
Returns the write throughput consumed by this operation, in KBytes.

Returns the write KBytes consumed.

Return type int

get_write_units()
Returns the write throughput consumed by this operation, in write units.

Returns the write units consumed.

Return type int

GetTableRequest

class borneo.**GetTableRequest**
Bases: borneo.operations.Request

Represents the argument of a *NoSQLHandle.get_table()* operation which returns static information associated with a table, as returned in *TableResult*. This information only changes in response to a change in table schema or a change in provisioned throughput or capacity for the table.

The table name is a required parameter.

Methods Summary

get_compartment()	Cloud service only.

Continued on next page

Table 12 – continued from previous page

get_operation_id()	Returns the operation id to use for the request, None if not set.
get_table_name()	Returns the table name to use for the operation.
get_timeout()	Returns the timeout to use for the operation, in milliseconds.
set_compartment(compartment)	Cloud service only.
set_operation_id(operation_id)	Sets the operation id to use for the request.
set_table_name(table_name)	Sets the table name to use for the request.
set_timeout(timeout_ms)	Sets the request timeout value, in milliseconds.

Methods Documentation

get_compartment()
Cloud service only.

Get the compartment id or name if set for the request.

Returns compartment id or name if set for the request, otherwise None if not set.

Return type str

get_operation_id()
Returns the operation id to use for the request, None if not set.

Returns the operation id.

Return type str

get_table_name()
Returns the table name to use for the operation.

Returns the table name, or None if not set.

Returns str

get_timeout()
Returns the timeout to use for the operation, in milliseconds. A value of 0 indicates that the timeout has not been set.

Returns the timeout value.

Return type int

set_compartment(*compartment*)
Cloud service only.

Sets the name or id of a compartment to be used for this operation.

The compartment may be specified as either a name (or path for nested compartments) or as an id (OCID). A name (vs id) can only be used when authenticated using a specific user identity. It is *not* available if authenticated as an Instance Principal which can be done when calling the service from a compute instance in the Oracle Cloud Infrastructure. See *borneo.iam.SignatureProvider.create_with_instance_principal()*.

Parameters **compartment** (*str*) – the compartment name or id. If using a nested compartment, specify the full compartment path compartmentA.compartmentB, but exclude the name of the root compartment (tenant).

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if compartment is not a str.

set_operation_id(*operation_id*)
Sets the operation id to use for the request. The operation id can be obtained via *TableResult.get_operation_id()*. This parameter is optional. If non-none, it represents an asynchronous table operation that may be in progress. It is used to examine the result of the operation and if the operation has failed an exception will be thrown in response to a *NoSQLHandle.get_table()* operation. If the operation is in progress or has completed successfully, the state of the table is returned.

Parameters **operation_id** (*str*) – the operation id. This is optional.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if operation_id is a negative number.

set_table_name(*table_name*)
Sets the table name to use for the request.

Parameters **table_name** (*str*) – the table name. This is a required parameter.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if table_name is not a string.

set_timeout(*timeout_ms*)
Sets the request timeout value, in milliseconds. This overrides any default value set in *NoSQLHandleConfig*. The value must be positive.

Parameters **timeout_ms** (*int*) – the timeout value, in milliseconds.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if the timeout value is less than or equal to 0.

IllegalArgumentException

exception borneo.**IllegalArgumentException**(*message=None*, *cause=None*)
Exception class that is used when an invalid argument was passed, this could mean that the type is not the expected or the value is not valid for the specific case.

IllegalStateException

exception borneo.**IllegalStateException**(*message=None*, *cause=None*)
Exception that is thrown when a method has been invoked at an illegal or inappropriate time.

IndexExistsException

exception borneo.**IndexExistsException**(*message*)
The operation attempted to create an index for a table but the named index already exists.

IndexInfo

class borneo.**IndexInfo**(*index_name*, *field_names*)
Bases: object

IndexInfo represents the information about a single index including its name and field names. Instances of this class are returned in *GetIndexesResult*.

Methods Summary

get_field_names()	Returns the list of field names that define the index.
get_index_name()	Returns the name of the index.

Methods Documentation

get_field_names()
Returns the list of field names that define the index.

Returns the field names.

Return type list(str)

get_index_name()
Returns the name of the index.

Returns the index name.

Return type str

IndexNotFoundException

exception borneo.**IndexNotFoundException**(*message*)
The operation attempted to access a index that does not exist or is not in a visible state.

InvalidAuthorizationException

exception borneo.**InvalidAuthorizationException**(*message*)
The exception is thrown if the application presents an invalid authorization string in a request.

ListTablesRequest

class borneo.**ListTablesRequest**
Bases: borneo.operations.Request

Represents the argument of a *NoSQLHandle.list_tables()* operation which lists all available tables associated with the identity associated with the handle used for the operation. If the list is large it can be paged by using the start_index and limit parameters. The list is returned in a simple array in *ListTablesResult*. Names are returned sorted in alphabetical order in order to facilitate paging.

Methods Summary

get_compartment()	Cloud service only.
get_limit()	Returns the maximum number of table names to return in the operation.
get_namespace()	On-premise only.
get_start_index()	Returns the index to use to start returning table names.
get_timeout()	Returns the timeout to use for the operation, in milliseconds.

Continued on next page

Table 14 – continued from previous page

set_compartment(compartment)	Cloud service only.
set_limit(limit)	Sets the maximum number of table names to return in the operation.
set_namespace(namespace)	On-premise only.
set_start_index(start_index)	Sets the index to use to start returning table names.
set_timeout(timeout_ms)	Sets the request timeout value, in milliseconds.

Methods Documentation

get_compartment()
Cloud service only.

Get the compartment id or name if set for the request.

Returns compartment id or name if set for the request, otherwise None if not set.

Return type str

get_limit()
Returns the maximum number of table names to return in the operation. If not set (0) there is no application-imposed limit.

Returns the maximum number of tables to return in a single request.

Return type int

get_namespace()
On-premise only.

Returns the namespace to use for the list or None if not set.

Returns the namespace.

Return type str

get_start_index()
Returns the index to use to start returning table names. This is related to the *ListTablesResult.get_last_returned_index()* from a previous request and can be used to page table names. If not set, the list starts at index 0.

Returns the start index.

Return type int

get_timeout()
Returns the timeout to use for the operation, in milliseconds. A value of 0 indicates that the timeout has not been set.

Returns the timeout value.

Return type int

set_compartment(*compartment*)
Cloud service only.

Sets the name or id of a compartment to be used for this operation.

The compartment may be specified as either a name (or path for nested compartments) or as an id (OCID). A name (vs id) can only be used when authenticated using a specific user identity. It is *not* available if authenticated as an Instance Principal which can be done when calling the service from a compute instance in the Oracle Cloud Infrastructure. See *borneo.iam.SignatureProvider.create_with_instance_principal()*.

Parameters **compartment** (*str*) – the compartment name or id. If using a nested compartment, specify the full compartment path compartmentA.compartmentB, but exclude the name of the root compartment (tenant).

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if compartment is not a str.

set_limit(*limit*)

Sets the maximum number of table names to return in the operation. If not set (0) there is no limit.

Parameters **limit** (*int*) – the maximum number of tables.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if limit is a negative number.

set_namespace(*namespace*)

On-premise only.

Sets the namespace to use for the list. If not set, all tables accessible to the user will be returned. If set, only tables in the namespace provided are returned.

Parameters **namespace** (*str*) – the namespace to use.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if namespace is not a string.

set_start_index(*start_index*)

Sets the index to use to start returning table names. This is related to the *ListTablesResult.get_last_returned_index()* from a previous request and can be used to page table names. If not set, the list starts at index 0.

Parameters **start_index** (*int*) – the start index.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if start_index is a negative number.

set_timeout(*timeout_ms*)

Sets the request timeout value, in milliseconds. This overrides any default value set in *NoSQLHandleConfig*. The value must be positive.

Parameters **timeout_ms** (*int*) – the timeout value, in milliseconds.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if the timeout value is less than or equal to 0.

ListTablesResult

class borneo.**ListTablesResult**

Bases: borneo.operations.Result

Represents the result of a *NoSQLHandle.list_tables()* operation.

On a successful operation the table names are available as well as the index of the last returned table. Tables are returned in a list, sorted alphabetically.

Methods Summary

get_last_returned_index()	Returns the index of the last table name returned.
get_tables()	Returns the array of table names returned by the operation.

Methods Documentation

get_last_returned_index()
Returns the index of the last table name returned. This can be provided to *ListTablesRequest* to be used as a starting point for listing tables.

Returns the index.

Return type int

get_tables()
Returns the array of table names returned by the operation.

Returns the table names.

Return type list(str)

MultiDeleteRequest

class borneo.**MultiDeleteRequest**
Bases: borneo.operations.Request

Represents the input to a *NoSQLHandle.multi_delete()* operation which can be used to delete a range of values that match the primary key and range provided.

A range is specified using a partial key plus a range based on the portion of the key that is not provided. For example if a table's primary key is <id, timestamp>; and the its shard key is the id, it is possible to delete a range of timestamp values for a specific id by providing an id but no timestamp in the value used for *set_key()* and providing a range of timestamp values in the *FieldRange* used in *set_range()*.

Because this operation can exceed the maximum amount of data modified in a single operation a continuation key can be used to continue the operation. The continuation key is obtained from *MultiDeleteResult.get_continuation_key()* and set in a new request using *set_continuation_key()*. Operations with a continuation key still require the primary key.

The table name and key are required parameters.

Methods Summary

get_compartment()	Cloud service only.
get_continuation_key()	Returns the continuation key if set.
get_durability()	On-premise only.
get_key()	Returns the key to be used for the operation.
get_max_write_kb()	Returns the limit on the total KB write during this operation.
get_range()	Returns the *FieldRange* to be used for the operation if set.
get_table_name()	Returns the table name to use for the operation.

Continued on next page

Table 16 – continued from previous page

get_timeout()	Returns the timeout to use for the operation, in milliseconds.
set_compartment(compartment)	Cloud service only.
set_continuation_key(continuation_key)	Sets the continuation key.
set_durability(durability)	On-premise only.
set_key(key)	Sets the key to be used for the operation.
set_max_write_kb(max_write_kb)	Sets the limit on the total KB write during this operation, 0 means no application-defined limit.
set_range(field_range)	Sets the *FieldRange* to be used for the operation.
set_table_name(table_name)	Sets the table name to use for the operation.
set_timeout(timeout_ms)	Sets the request timeout value, in milliseconds.

Methods Documentation

get_compartment()
Cloud service only.

Get the compartment id or name if set for the request.

Returns compartment id or name if set for the request, otherwise None if not set.

Return type str

get_continuation_key()
Returns the continuation key if set.

Returns the continuation key.

Return type bytearray

get_durability()
On-premise only. Gets the durability to use for the operation or None if not set :returns: the Durability :versionadded: 5.3.0

get_key()
Returns the key to be used for the operation.

Returns the key.

Return type dict

get_max_write_kb()
Returns the limit on the total KB write during this operation. If not set by the application this value will be 0 which means the default system limit is used.

Returns the limit, or 0 if not set.

Return type int

get_range()
Returns the *FieldRange* to be used for the operation if set.

Returns the range, None if no range is to be used.

Return type *FieldRange*

get_table_name()
Returns the table name to use for the operation.

Returns the table name, or None if not set.

Returns str

get_timeout()
Returns the timeout to use for the operation, in milliseconds. A value of 0 indicates that the timeout has not been set.

Returns the timeout value.

Return type int

set_compartment(*compartment*)
Cloud service only.

Sets the name or id of a compartment to be used for this operation.

The compartment may be specified as either a name (or path for nested compartments) or as an id (OCID). A name (vs id) can only be used when authenticated using a specific user identity. It is *not* available if authenticated as an Instance Principal which can be done when calling the service from a compute instance in the Oracle Cloud Infrastructure. See *borneo.iam.SignatureProvider.create_with_instance_principal()*.

Parameters **compartment** (*str*) – the compartment name or id. If using a nested compartment, specify the full compartment path compartmentA.compartmentB, but exclude the name of the root compartment (tenant).

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if compartment is not a str.

set_continuation_key(*continuation_key*)
Sets the continuation key.

Parameters **continuation_key** (*bytearray*) – the key which should have been obtained from *MultiDeleteResult.get_continuation_key()*.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if continuation_key is not a bytearray.

set_durability(*durability*)
On-premise only. Sets the durability to use for the operation.

Parameters **durability** (Durability) – the Durability to use

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if Durability is not valid

Versionadded 5.3.0

set_key(*key*)
Sets the key to be used for the operation. This is a required parameter and must completely specify the target table's shard key.

Parameters **key** (*dict*) – the key.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if key is not a dictionary.

set_max_write_kb(*max_write_kb*)
Sets the limit on the total KB write during this operation, 0 means no application-defined limit. This value can only reduce the system defined limit.

Parameters **max_write_kb** (*int*) – the limit in terms of number of KB write during this operation.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if the max_write_kb value is less than 0.

set_range(*field_range*)
Sets the *FieldRange* to be used for the operation. This parameter is optional, but required to delete a specific range of rows.

Parameters **field_range** (FieldRange) – the field range.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if field_range is not an instance of FieldRange.

set_table_name(*table_name*)
Sets the table name to use for the operation. This is a required parameter.

Parameters **table_name** (*str*) – the table name.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if table_name is not a string.

set_timeout(*timeout_ms*)
Sets the request timeout value, in milliseconds. This overrides any default value set in *NoSQLHandleConfig*. The value must be positive.

Parameters **timeout_ms** (*int*) – the timeout value, in milliseconds.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if the timeout value is less than or equal to 0.

MultiDeleteResult

class borneo.**MultiDeleteResult**
Bases: borneo.operations.Result

Represents the result of a *NoSQLHandle.multi_delete()* operation.

On a successful operation the number of rows deleted is available using *get_num_deletions()*. There is a limit to the amount of data consumed by a single call. If there are still more rows to delete, the continuation key can be get using *get_continuation_key()*.

Methods Summary

get_continuation_key()	Returns the continuation key where the next MultiDelete request resume from.
get_num_deletions()	Returns the number of rows deleted from the table.
get_read_kb()	Returns the read throughput consumed by this operation, in KBytes.
get_read_units()	Returns the read throughput consumed by this operation, in read units.
get_write_kb()	Returns the write throughput consumed by this operation, in KBytes.

Continued on next page

Table 17 – continued from previous page

get_write_units()	Returns the write throughput consumed by this operation, in write units.

Methods Documentation

get_continuation_key()
Returns the continuation key where the next MultiDelete request resume from.

Returns the continuation key, or None if there are no more rows to delete.

Return type bytearray

get_num_deletions()
Returns the number of rows deleted from the table.

Returns the number of rows deleted.

Return type int

get_read_kb()
Returns the read throughput consumed by this operation, in KBytes. This is the actual amount of data read by the operation. The number of read units consumed is returned by *get_read_units()* which may be a larger number because this was an update operation.

Returns the read KBytes consumed.

Return type int

get_read_units()
Returns the read throughput consumed by this operation, in read units. This number may be larger than that returned by *get_read_kb()* because it was an update operation.

Returns the read units consumed.

Return type int

get_write_kb()
Returns the write throughput consumed by this operation, in KBytes.

Returns the write KBytes consumed.

Return type int

get_write_units()
Returns the write throughput consumed by this operation, in write units.

Returns the write units consumed.

Return type int

NoSQLException

exception borneo.**NoSQLException**(*message*, *cause=None*)
A base class for most exceptions thrown by the NoSQL driver.

NoSQLHandle

class borneo.**NoSQLHandle**(*config*)
Bases: object

NoSQLHandle is a handle that can be used to access Oracle NoSQL tables. To create a connection represented by NoSQLHandle, request an instance using *NoSQLHandleConfig*, which allows an application to specify default values and other configuration information to be used by the handle.

The same interface is available to both users of the Oracle NoSQL Database Cloud Service and the on-premise Oracle NoSQL Database; however, some methods and/or parameters are specific to each environment. The documentation has notes about whether a class, method, or parameter is environment-specific. Unless otherwise noted they are applicable to both environments.

A handle has memory and network resources associated with it. Consequently, the *close()* method must be invoked to free up the resources when the application is done using the handle. To minimize network activity as well as resource allocation and deallocation overheads, it's best to avoid repeated creation and closing of handles. For example, creating and closing a handle around each operation, would incur large resource allocation overheads resulting in poor application performance.

A handle permits concurrent operations, so a single handle is sufficient to access tables in a multi-threaded application. The creation of multiple handles incurs additional resource overheads without providing any performance benefit.

With the exception of *close()* the operations on this interface follow a similar pattern. They accept a Request object containing parameters, both required and optional. They return a Result object containing results. Operation failures throw exceptions. Unique subclasses of Request and Result exist for most operations, containing information specific to the operation. All of these operations result in remote calls across a network.

All Request instances support specification of parameters for the operation as well as the ability to override default parameters which may have been specified in *NoSQLHandleConfig*, such as request timeouts, etc.

Objects returned by methods of this interface can only be used safely by one thread at a time unless synchronized externally. Request objects are not copied and must not be modified by the application while a method on this interface is using them.

For Error and Exception Handling, on success all methods in this interface return Result objects. Errors are thrown as exceptions. Exceptions that may be retried may succeed on retry. These are instances of *RetryableException*. Exceptions that may not be retried and if retried, will fail again. Exceptions that may be retried return True for `RetryableException.ok_to_retry()` while those that may not will return False. Examples of retryable exceptions are those which indicate resource consumption violations such as *OperationThrottlingException*. Examples of exceptions that should not be retried are *IllegalArgumentException*, *TableNotFoundException*, and any other exception indicating a syntactic or semantic error.

Instances of NoSQLHandle are thread-safe and expected to be shared among threads.

Parameters **config** (NoSQLHandleConfig) – an instance of NoSQLHandleConfig.

Raises ***IllegalArgumentException*** – raises the exception if config is not an instance of NoSQLHandleConfig.

Methods Summary

close()	Close the NoSQLHandle.
delete(request)	Deletes a row from a table.
do_system_request(statement[, timeout_ms, ...])	On-premise only.
do_table_request(request, timeout_ms, ...)	A convenience method that performs a TableRequest and waits for completion of the operation.
get(request)	Gets the row associated with a primary key.

Continued on next page

Table 18 – continued from previous page

get_client()	
get_indexes(request)	Returns information about and index, or indexes on a table.
get_stats_control()	
get_table(request)	Gets static information about the specified table including its state, provisioned throughput and capacity and schema.
get_table_usage(request)	Cloud service only.
list_namespaces()	On-premise only.
list_roles()	On-premise only.
list_tables(request)	Lists tables, returning table names.
list_users()	On-premise only.
multi_delete(request)	Deletes multiple rows from a table in an atomic operation.
prepare(request)	Prepares a query for execution and reuse.
put(request)	Puts a row into a table.
query(request)	Queries a table based on the query statement specified in the *QueryRequest*.
query_iterable(request)	Queries a table based on the query statement specified in the *QueryRequest*.
system_request(request)	On-premise only.
system_status(request)	On-premise only.
table_request(request)	Performs an operation on a table.
write_multiple(request)	Executes a sequence of operations associated with a table that share the same shard key portion of their primary keys, all the specified operations are executed within the scope of a single transaction.

Methods Documentation

close()
Close the NoSQLHandle.

delete(*request*)
Deletes a row from a table. The row is identified using a primary key value supplied in *DeleteRequest.set_key()*.

By default a delete operation is unconditional and will succeed if the specified row exists. Delete operations can be made conditional based on whether the *Version* of an existing row matches that supplied by *DeleteRequest.set_match_version()*.

It is also possible, on failure, to return information about the existing row. The row, including it's *Version* can be optionally returned if a delete operation fails because of a Version mismatch. The existing row information will only be returned if *DeleteRequest.set_return_row()* is True and the operation fails because *DeleteRequest.set_match_version()* is used and the operation fails because the row exists and its version does not match. Use of *DeleteRequest.set_return_row()* may result in additional consumed read capacity. If the operation is successful there will be no information returned about the previous row.

Parameters **request** (DeleteRequest) – the input parameters for the operation.

Returns the result of the operation.

Return type *DeleteResult*

Raises

- ***IllegalArgumentException*** – raises the exception if request is not an instance of *DeleteRequest*.
- ***NoSQLException*** – raises the exception if the operation cannot be performed for any other reason.

do_system_request(*statement*, *timeout_ms=30000*, *poll_interval_ms=1000*)

On-premise only.

A convenience method that performs a SystemRequest and waits for completion of the operation. This is the same as calling *system_request()* then calling *SystemResult.wait_for_completion()*. If the operation fails an exception is thrown.

System requests are those related to namespaces and security and are generally independent of specific tables. Examples of statements include

CREATE NAMESPACE mynamespace

CREATE USER some_user IDENTIFIED BY password

CREATE ROLE some_role

GRANT ROLE some_role TO USER some_user

Parameters

- **statement** (*str*) – the system statement for the operation.
- **timeout_ms** (*int*) – the amount of time to wait for completion, in milliseconds.
- **poll_interval_ms** (*int*) – the polling interval for the wait operation.

Returns the result of the system request.

Return type *SystemResult*

Raises

- ***IllegalArgumentException*** – raises the exception if any of the parameters are invalid or required parameters are missing.
- ***RequestTimeoutException*** – raises the exception if the operation times out.
- ***NoSQLException*** – raises the exception if the operation cannot be performed for any other reason.

do_table_request(*request*, *timeout_ms*, *poll_interval_ms*)

A convenience method that performs a TableRequest and waits for completion of the operation. This is the same as calling *table_request()* then calling *TableResult.wait_for_completion()*. If the operation fails an exception is thrown. All parameters are required.

Parameters

- **request** (TableRequest) – the *TableRequest* to perform.
- **timeout_ms** (*int*) – the amount of time to wait for completion, in milliseconds.
- **poll_interval_ms** (*int*) – the polling interval for the wait operation.

Returns the result of the table request.

Return type *TableResult*

Raises

- ***IllegalArgumentException*** – raises the exception if any of the parameters are invalid or required parameters are missing.
- ***RequestTimeoutException*** – raises the exception if the operation times out.
- ***NoSQLException*** – raises the exception if the operation cannot be performed for any other reason.

get(*request*)

Gets the row associated with a primary key. On success the value of the row is available using the *GetResult.get_value()* operation. If there are no matching rows that method will return None.

The default consistency used for the operation is Consistency.EVENTUAL unless an explicit value has been set using *NoSQLHandleConfig.set_consistency()* or *GetRequest.set_consistency()*. Use of Consistency.ABSOLUTE may affect latency of the operation and may result in additional cost for the operation.

Parameters **request** (GetRequest) – the input parameters for the operation.

Returns the result of the operation.

Return type *GetResult*

Raises

- ***IllegalArgumentException*** – raises the exception if request is not an instance of *GetRequest*.
- ***NoSQLException*** – raises the exception if the operation cannot be performed for any other reason.

get_client()

get_indexes(*request*)

Returns information about and index, or indexes on a table. If no index name is specified in the *GetIndexesRequest*, then information on all indexes is returned.

Parameters **request** (GetIndexesRequest) – the input parameters for the operation.

Returns the result of the operation.

Return type *GetIndexesResult*

Raises

- ***IllegalArgumentException*** – raises the exception if request is not an instance of *GetIndexesRequest*.
- ***NoSQLException*** – raises the exception if the operation cannot be performed for any other reason.

get_stats_control()

get_table(*request*)

Gets static information about the specified table including its state, provisioned throughput and capacity and schema. Dynamic information such as usage is obtained using *get_table_usage()*. Throughput, capacity and usage information is only available when using the Cloud Service and will be None or not defined on-premise.

Parameters **request** (GetTableRequest) – the input parameters for the operation.

Returns the result of the operation.

Return type *TableResult*

Raises

- ***IllegalArgumentException*** – raises the exception if request is not an instance of *GetTableRequest*.
- ***TableNotFoundException*** – raises the exception if the specified table does not exist.
- ***NoSQLException*** – raises the exception if the operation cannot be performed for any other reason.

get_table_usage(*request*)
Cloud service only.

Gets dynamic information about the specified table such as the current throughput usage. Usage information is collected in time slices and returned in individual usage records. It is possible to specify a time-based range of usage records using input parameters.

Parameters **request** (*TableUsageRequest*) – the input parameters for the operation.

Returns the result of the operation.

Return type *TableUsageResult*

Raises

- ***IllegalArgumentException*** – raises the exception if request is not an instance of *TableUsageRequest*.
- ***TableNotFoundException*** – raises the exception if the specified table does not exist.
- ***NoSQLException*** – raises the exception if the operation cannot be performed for any other reason.

list_namespaces()
On-premise only.

Returns the namespaces in a store as a list of string.

Returns the namespaces, or None if none are found.

Return type list(str)

list_roles()
On-premise only.

Returns the roles in a store as a list of string.

Returns the list of roles, or None if none are found.

Return type list(str)

list_tables(*request*)
Lists tables, returning table names. If further information about a specific table is desired the *get_table()* interface may be used. If a given identity has access to a large number of tables the list may be paged using input parameters.

Parameters **request** (*ListTablesRequest*) – the input parameters for the operation.

Returns the result of the operation.

Return type *ListTablesResult*

Raises

- ***IllegalArgumentException*** – raises the exception if request is not an instance of *ListTablesRequest*.
- ***NoSQLException*** – raises the exception if the operation cannot be performed for any other reason.

list_users()
On-premise only.

Returns the users in a store as a list of *UserInfo*.

Returns the list of users, or None if none are found.

Return type list(*UserInfo*)

multi_delete(*request*)
Deletes multiple rows from a table in an atomic operation. The key used may be partial but must contain all of the fields that are in the shard key. A range may be specified to delete a range of keys.

Parameters **request** (MultiDeleteRequest) – the input parameters for the operation.

Returns the result of the operation.

Return type *MultiDeleteResult*

Raises

- ***IllegalArgumentException*** – raises the exception if request is not an instance of *MultiDeleteRequest*.
- ***NoSQLException*** – raises the exception if the operation cannot be performed for any other reason.

prepare(*request*)
Prepares a query for execution and reuse. See *query()* for general information and restrictions. It is recommended that prepared queries are used when the same query will run multiple times as execution is much more efficient than starting with a query string every time. The query language and API support query variables to assist with re-use.

Parameters **request** (PrepareRequest) – the input parameters for the operation.

Returns the result of the operation.

Return type *PrepareResult*

Raises

- ***IllegalArgumentException*** – raises the exception if request is not an instance of *PrepareRequest*.
- ***NoSQLException*** – raises the exception if the operation cannot be performed for any other reason.

put(*request*)
Puts a row into a table. This method creates a new row or overwrites an existing row entirely. The value used for the put is in the *PutRequest* object and must contain a complete primary key and all required fields.

It is not possible to put part of a row. Any fields that are not provided will be defaulted, overwriting any existing value. Fields that are not noneable or defaulted must be provided or an exception will be thrown.

By default a put operation is unconditional, but put operations can be conditional based on existence, or not, of a previous value as well as conditional on the *Version* of the existing value.

Use PutOption.IF_ABSENT to do a put only if there is no existing row that matches the primary key.

Use PutOption.IF_PRESENT to do a put only if there is an existing row that matches the primary key.

Use PutOption.IF_VERSION to do a put only if there is an existing row that matches the primary key and its *Version* matches that provided.

It is also possible, on failure, to return information about the existing row. The row, including it's *Version* can be optionally returned if a put operation fails because of a Version mismatch or if the operation fails because the row already exists. The existing row information will only be returned if *PutRequest.set_return_row()* is True and one of the following occurs:

The PutOption.IF_ABSENT is used and the operation fails because the row already exists.

The PutOption.IF_VERSION is used and the operation fails because the row exists and its version does not match.

Use of *PutRequest.set_return_row()* may result in additional consumed read capacity. If the operation is successful there will be no information returned about the previous row.

Parameters **request** (PutRequest) – the input parameters for the operation.

Returns the result of the operation.

Return type *PutResult*

Raises

- ***IllegalArgumentException*** – raises the exception if request is not an instance of *PutRequest*.
- ***NoSQLException*** – raises the exception if the operation cannot be performed for any other reason.

query(*request*)

Queries a table based on the query statement specified in the *QueryRequest*.

Queries that include a full shard key will execute much more efficiently than more distributed queries that must go to multiple shards.

Table and system-style queries such as "CREATE TABLE ..." or "DROP TABLE ..." are not supported by these interfaces. Those operations must be performed using *table_request()* or *system_request()* as appropriate.

The amount of data read by a single query request is limited by a system default and can be further limited using *QueryRequest.set_max_read_kb()*. This limits the amount of data *read* and not the amount of data *returned*, which means that a query can return zero results but still have more data to read. This situation is detected by checking if the *QueryRequest* is done using *QueryRequest.is_done()*. For this reason queries should always operate in a loop, acquiring more results, until *QueryRequest.is_done()* returns True, indicating that the query is done.

Parameters **request** (QueryRequest) – the input parameters for the operation.

Returns the result of the operation.

Return type *QueryResult*

Raises

- ***IllegalArgumentException*** – raises the exception if request is not an instance of *QueryRequest*.
- ***NoSQLException*** – raises the exception if the operation cannot be performed for any other reason.

query_iterable(*request*)

Queries a table based on the query statement specified in the *QueryRequest*.

Queries that include a full shard key will execute much more efficiently than more distributed queries that must go to multiple shards.

Table and system-style queries such as "CREATE TABLE ..." or "DROP TABLE ..." are not supported by these interfaces. Those operations must be performed using *table_request()* or *system_request()* as appropriate.

The amount of data read by a single query request is limited by a system default and can be further limited using *QueryRequest.set_max_read_kb()*. This limits the amount of data *read* and not the amount of data *returned*.

Parameters **request** (QueryRequest) – the input parameters for the operation.

Returns the result of the operation.

Return type *QueryResult*

Raises

- ***IllegalArgumentException*** – raises the exception if request is not an instance of *QueryRequest*.
- ***NoSQLException*** – raises the exception if the operation cannot be performed for any other reason.

Versionadded 5.3.6

system_request (*request*)

On-premise only.

Performs a system operation on the system, such as administrative operations that don't affect a specific table. For table-specific operations use *table_request()* or *do_table_request()*.

Examples of statements in the *SystemRequest* passed to this method include:

CREATE NAMESPACE mynamespace

CREATE USER some_user IDENTIFIED BY password

CREATE ROLE some_role

GRANT ROLE some_role TO USER some_user

This operation is implicitly asynchronous. The caller must poll using methods on *SystemResult* to determine when it has completed.

Parameters **request** (SystemRequest) – the input parameters for the operation.

Returns the result of the operation.

Return type *SystemResult*

Raises

- ***IllegalArgumentException*** – raises the exception if request is not an instance of *SystemRequest*.
- ***NoSQLException*** – raises the exception if the operation cannot be performed for any other reason.

system_status (*request*)

On-premise only.

Checks the status of an operation previously performed using *system_request()*.

Parameters **request** (SystemStatusRequest) – the input parameters for the operation.

Returns the result of the operation.

Return type *SystemResult*

Raises

- ***IllegalArgumentException*** – raises the exception if request is not an instance of *SystemStatusRequest*.
- ***NoSQLException*** – raises the exception if the operation cannot be performed for any other reason.

table_request(*request*)

Performs an operation on a table. This method is used for creating and dropping tables and indexes as well as altering tables. Only one operation is allowed on a table at any one time.

This operation is implicitly asynchronous. The caller must poll using methods on *TableResult* to determine when it has completed.

Parameters **request** (TableRequest) – the input parameters for the operation.

Returns the result of the operation.

Return type *TableResult*

Raises

- ***IllegalArgumentException*** – raises the exception if request is not an instance of *TableRequest*.
- ***NoSQLException*** – raises the exception if the operation cannot be performed for any other reason.

write_multiple(*request*)

Executes a sequence of operations associated with a table that share the same shard key portion of their primary keys, all the specified operations are executed within the scope of a single transaction.

There are some size-based limitations on this operation:

The max number of individual operations (put, delete) in a single WriteMultipleRequest is 50.

The total request size is limited to 25MB.

Parameters **request** (WriteMultipleRequest) – the input parameters for the operation.

Returns the result of the operation.

Return type *WriteMultipleResult*

Raises

- ***IllegalArgumentException*** – raises the exception if request is not an instance of *WriteMultipleRequest*.
- **RowSizeLimitException** – raises the exception if data size in an operation exceeds the limit.
- ***BatchOperationNumberLimitException*** – raises the exception if the number of operations exceeds this limit.
- ***NoSQLException*** – raises the exception if the operation cannot be performed for any other reason.

NoSQLHandleConfig

class borneo.**NoSQLHandleConfig**(*endpoint=None*, *provider=None*)
Bases: object

An instance of this class is required by *NoSQLHandle*.

NoSQLHandleConfig groups parameters used to configure a *NoSQLHandle*. It also provides a way to default common parameters for use by *NoSQLHandle* methods. When creating a *NoSQLHandle*, the NoSQLHandleConfig instance is copied so modification operations on the instance have no effect on existing handles which are immutable. NoSQLHandle state with default values can be overridden in individual operations.

The service endpoint is used to connect to the Oracle NoSQL Database Cloud Service or, if on-premise, the Oracle NoSQL Database proxy server. It should be a string or a *Region*.

If a string is provided to endpoint argument, there is flexibility in how endpoints are specified. A fully specified endpoint is of the format:

- http[s]://host:port

It also accepts portions of a fully specified endpoint, including a region id (see *Region*) string if using the Cloud service. A valid endpoint is one of these:

- region id string (cloud service only)
- a string with the syntax [http[s]://]host[:port]

For example, these are valid endpoint arguments:

- us-ashburn-1 (equivalent to using Region Regions.US_ASHBURN_1 as the endpoint argument)
- nosql.us-ashburn-1.oci.oraclecloud.com (equivalent to using Region Regions.US_ASHBURN_1 as the endpoint argument)
- https://nosql.us-ashburn-1.oci.oraclecloud.com:443
- localhost:8080 - used for connecting to a Cloud Simulator instance running locally on port 8080
- https://machine-hosting-proxy:443

When using the endpoint (vs region id) syntax, if the port is omitted, the endpoint uses 8080 if protocol is http, and 443 in all other cases. If the protocol is omitted, the endpoint uses https if the port is 443, and http in all other cases.

When using the Oracle NoSQL Database Cloud Service, it is recommended that a *Region* object is provided rather than a Region's id string.

If a *Region* object is provided to endpoint argument, See *Regions* for information on available regions. For example:

- Regions.US_ASHBURN_1

For cloud service, one or both of endpoint and provider must be set. For other scenarios, endpoint is required while provider is optional.

Parameters

- **endpoint** (*str or Region*) – identifies a server, region id or *Region* for use by the NoSQLHandle.
- **provider** (AuthorizationProvider) – *AuthorizationProvider* to use for the handle.

Raises ***IllegalArgumentException*** – raises the exception if the endpoint is None or malformed.

Methods Summary

clone()	All the configurations will be copied.
configure_default_retry_handler(num_retries, ...)	Sets the *RetryHandler* using a default retry handler configured with the specified number of retries and a static delay.
get_authorization_provider()	Returns the *AuthorizationProvider* configured for the handle, or None.
get_consistency()	Returns the configured default *Consistency*, None if it has not been configured.
get_default_compartment()	Cloud service only.
get_default_consistency()	Returns the default consistency value that will be used by the system.
get_default_table_request_timeout()	Returns the default value for a table request timeout.
get_default_timeout()	Returns the default value for request timeout in milliseconds.
get_logger()	Returns the logger, or None if not configured by user.
get_max_content_length()	Returns the maximum size, in bytes, of a request operation payload.
get_pool_connections()	Returns the number of connection pools to cache.
get_pool_maxsize()	Returns the maximum number of individual connections to use to connect to the service.
get_region()	Cloud service only.
get_retry_handler()	Returns the *RetryHandler* configured for the handle, or None if None is set.
get_service_url()	Returns the url to use for the *NoSQLHandle* connection.
get_ssl_ca_certs()	Returns the SSL CA certificates.
get_ssl_cipher_suites()	Returns the SSL cipher suites to enable.
get_ssl_protocol()	Returns the SSL protocols to enable.
get_table_request_timeout()	Returns the configured table request timeout value, in milliseconds.
get_timeout()	Returns the configured request timeout value, in milliseconds, 0 if it has not been set.
set_authorization_provider(provider)	Sets the *AuthorizationProvider* to use for the handle.
set_consistency(consistency)	Sets the default request *Consistency*.
set_default_compartment(compartment)	Cloud service only.
set_default_rate_limiting_percentage(percent)	Cloud service only.
set_logger(logger)	Sets the logger used for the driver.
set_max_content_length(max_content_length)	Sets the maximum size in bytes of request payloads.
set_pool_connections(pool_connections)	Sets the number of connection pools to cache.
set_pool_maxsize(pool_maxsize)	Sets the maximum number of individual connections to use to connect to to the service.
set_rate_limiting_enabled(enable)	Cloud service only.
set_retry_handler(retry_handler)	Sets the *RetryHandler* to use for the handle.
set_ssl_ca_certs(ssl_ca_certs)	On-premise only.
set_ssl_cipher_suites(ssl_ciphers)	Set SSL cipher suites to enable.
set_ssl_protocol(ssl_protocol)	Set SSL protocol to enable.
set_table_request_timeout(table_request_timeout)	Sets the default table request timeout.

Continued on next page

Table 19 – continued from previous page

set_timeout(timeout)	Sets the default request timeout in milliseconds, the default timeout is 5 seconds.

Methods Documentation

clone()
All the configurations will be copied.

Returns the copy of the instance.

Return type *NoSQLHandleConfig*

configure_default_retry_handler(*num_retries*, *delay_s*)
Sets the *RetryHandler* using a default retry handler configured with the specified number of retries and a static delay. A delay of 0 means "use the default delay algorithm" which is an incremental backoff algorithm. A non-zero delay will work but is not recommended for production systems as it is not flexible.

The default retry handler will not retry exceptions of type *OperationThrottlingException*. The reason is that these operations are long-running, and while technically they can be retried, an immediate retry is unlikely to succeed because of the low rates allowed for these operations.

Parameters

- **num_retries** (*int*) – the number of retries to perform automatically. This parameter may be 0 for no retries.
- **delay_s** (*int*) – the delay, in seconds. Pass 0 to use the default delay algorithm.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if num_retries or delay_s is a negative number.

get_authorization_provider()
Returns the *AuthorizationProvider* configured for the handle, or None.

Returns the AuthorizationProvider.

Return type *AuthorizationProvider*

get_consistency()
Returns the configured default *Consistency*, None if it has not been configured.

Returns the consistency, or None if it has not been configured.

Return type *Consistency*

get_default_compartment()
Cloud service only.

Returns the default compartment to use for requests or None if not set. The value may be a compartment name or id, as set by *set_default_compartment()*.

Returns the compartment, or None.

Return type str or None

get_default_consistency()
Returns the default consistency value that will be used by the system. If consistency has been set using *set_consistency()*, that will be returned. If not a default value of Consistency.EVENTUAL is returned.

Returns the default consistency.

Return type *Consistency*

get_default_table_request_timeout()
Returns the default value for a table request timeout. If there is no configured timeout or it is configured as 0, a "default" default value of 10000 milliseconds is used.

Returns the default timeout, in milliseconds.

Return type int

get_default_timeout()
Returns the default value for request timeout in milliseconds. If there is no configured timeout or it is configured as 0, a "default" value of 5000 milliseconds is used.

Returns the default timeout, in milliseconds.

Return type int

get_logger()
Returns the logger, or None if not configured by user.

Returns the logger.

Return type Logger

get_max_content_length()
Returns the maximum size, in bytes, of a request operation payload. On-premise only. This value is ignored for cloud operations.

Returns the size.

Return type int

get_pool_connections()
Returns the number of connection pools to cache.

Returns the number of connection pools.

Return type int

get_pool_maxsize()
Returns the maximum number of individual connections to use to connect to the service. Each request/response pair uses a connection. The pool exists to allow concurrent requests and will bound the number of concurrent requests. Additional requests will wait for a connection to become available.

Returns the pool size.

Return type int

get_region()
Cloud service only.

Returns the region will be accessed by the NoSQLHandle.

Returns the region.

Return type *Region*

get_retry_handler()
Returns the *RetryHandler* configured for the handle, or None if None is set.

Returns the handler.

Return type *RetryHandler*

get_service_url()
Returns the url to use for the *NoSQLHandle* connection.

Returns the url.

Return type ParseResult

get_ssl_ca_certs()
Returns the SSL CA certificates.

Returns ssl ca certificates.

Return type str

get_ssl_cipher_suites()
Returns the SSL cipher suites to enable.

Returns ssl ciphers in a string in the OpenSSL cipher list format.

Return type str

get_ssl_protocol()
Returns the SSL protocols to enable.

Returns ssl protocols.

Return type int

get_table_request_timeout()
Returns the configured table request timeout value, in milliseconds. The table request timeout default can be specified independently to allow it to be larger than a typical data request. If it is not specified the default table request timeout of 10000 is used.

Returns the timeout, in milliseconds, or 0 if it has not been set.

Return type int

get_timeout()
Returns the configured request timeout value, in milliseconds, 0 if it has not been set.

Returns the timeout, in milliseconds, or 0 if it has not been set.

Return type int

set_authorization_provider(*provider*)
Sets the *AuthorizationProvider* to use for the handle. The provider must be safely usable by multiple threads.

Parameters **provider** (AuthorizationProvider) – the AuthorizationProvider.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if provider is not an instance of *AuthorizationProvider*.

set_consistency(*consistency*)
Sets the default request *Consistency*. If not set in this object or by a specific request, the default consistency used is Consistency.EVENTUAL.

Parameters **consistency** (Consistency) – the consistency.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if consistency is not Consistency.ABSOLUTE or Consistency.EVENTUAL.

set_default_compartment(*compartment*)
Cloud service only.

Sets the default compartment to use for requests sent using the handle. Setting the default is optional and if set it is overridden by any compartment specified in a request or table name. If no compartment is set for a request, either using this default or by specification in a request, the behavior varies with how the application is authenticated:

- If authenticated with a user identity the default is the root compartment of the tenancy
- If authenticated as an instance principal (see *borneo.iam.SignatureProvider.create_with_instance_principal()*) the compartment id (OCID) must be specified by either using this method or in each Request object. If not an exception is thrown.

Parameters **compartment** (*str*) – may be either the name of a compartment or the id (OCID) of a compartment.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if compartment is not a string.

set_default_rate_limiting_percentage(*percent*)
Cloud service only.

Sets a default percentage of table limits to use. This may be useful for cases where a client should only use a portion of full table limits. This only applies if rate limiting is enabled using *set_rate_limiting_enabled()*.

The default for this value is 100.0 (full table limits).

Parameters **percent** (*int or float or Decimal*) – the percentage of table limits to use. This value must be positive.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if percent is not a positive digital number.

set_logger(*logger*)
Sets the logger used for the driver.

Parameters **logger** (*Logger*) – the logger or None, None means disable logging.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if logger is not an instance of Logger.

set_max_content_length(*max_content_length*)
Sets the maximum size in bytes of request payloads. On-premise only. This setting is ignored for cloud operations. If not set, or set to zero, the default value of 32MB is used.

Parameters **max_content_length** (*int*) – the maximum bytes allowed in requests. Pass zero to use the default.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if max_content_length is a negative number.

set_pool_connections(*pool_connections*)
Sets the number of connection pools to cache.

Parameters **pool_connections** (*int*) – the number of connection pools.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if pool_connections is not a positive number.

set_pool_maxsize(*pool_maxsize*)
Sets the maximum number of individual connections to use to connect to to the service. Each request/response pair uses a connection. The pool exists to allow concurrent requests and will bound the number of concurrent requests. Additional requests will wait for a connection to become available.

Parameters **pool_maxsize** (*int*) – the pool size.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if pool_maxsize is not a positive number.

set_rate_limiting_enabled(*enable*)
Cloud service only.

Enables internal rate limiting.

Parameters **enable** (*bool*) – If True, enable internal rate limiting, otherwise disable internal rate limiting.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if enable is not a boolean.

set_retry_handler(*retry_handler*)
Sets the *RetryHandler* to use for the handle. If no handler is configured a default is used. The handler must be safely usable by multiple threads.

Parameters **retry_handler** (RetryHandler) – the handler.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if retry_handler is not an instance of *RetryHandler*.

set_ssl_ca_certs(*ssl_ca_certs*)
On-premise only.

When running against on-premise Oracle NoSQL Database with security enabled, certificates should be specified using this method. Otherwise environment variable REQUESTS_CA_BUNDLE should be configured. See the installation guide for the configuration of REQUESTS_CA_BUNDLE.

Parameters **ssl_ca_certs** (*str*) – ssl ca certificates.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if ssl_ca_certs is not a string.

set_ssl_cipher_suites(*ssl_ciphers*)
Set SSL cipher suites to enable.

Parameters **ssl_ciphers** (*str*) – ssl ciphers in a string in the OpenSSL cipher list format.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if ssl_ciphers is not a string.

set_ssl_protocol(*ssl_protocol*)
Set SSL protocol to enable.

Parameters **ssl_protocol** (*int*) – ssl protocol version.

Returns self.

Raises *IllegalArgumentException* – raises the exception if ssl_protocol is a negative integer.

set_table_request_timeout(*table_request_timeout*)
Sets the default table request timeout. The default timeout is 5 seconds. The table request timeout can be specified independently of that specified by `set_request_timeout()` because table requests can take longer and justify longer timeouts. The default timeout is 10 seconds (10000 milliseconds).

Parameters **table_request_timeout** (*int*) – the timeout value, in milliseconds.

Returns self.

Raises *IllegalArgumentException* – raises the exception if table_request_timeout is a negative number.

set_timeout(*timeout*)
Sets the default request timeout in milliseconds, the default timeout is 5 seconds.

Parameters **timeout** (*int*) – the timeout value, in milliseconds.

Returns self.

Raises *IllegalArgumentException* – raises the exception if timeout is a negative number.

OperationNotSupportedException

exception borneo.**OperationNotSupportedException**(*message*)
The operation attempted is not supported. This may be related to on-premise vs cloud service configurations.

OperationResult

class borneo.**OperationResult**
Bases: `borneo.operations.WriteResult`

A single Result associated with the execution of an individual operation in a *NoSQLHandle.write_multiple()* request. A list of OperationResult is contained in *WriteMultipleResult* and obtained using *WriteMultipleResult.get_results()*.

Methods Summary

get_existing_modification_time()	Returns the existing row modification time if available.
get_existing_value()	Returns the previous row value associated with the key if available.
get_existing_version()	Returns the existing row version associated with the key if available.
get_generated_value()	Returns the value generated if the operation created a new value.
get_success()	Returns the flag indicates whether the operation succeeded.
get_version()	Returns the version of the new row for put operation, or None if put operations did not succeed or the operation is delete operation.

Methods Documentation

get_existing_modification_time()
Returns the existing row modification time if available.

Returns the modification time in milliseconds since January 1, 1970

Return type int

Versionadded 5.3.0

get_existing_value()
Returns the previous row value associated with the key if available.

Returns the previous row value

Return type dict

get_existing_version()
Returns the existing row version associated with the key if available.

Returns the existing row version

Return type *Version*

get_generated_value()
Returns the value generated if the operation created a new value. This can happen if the table contains an identity column or string column declared as a generated UUID. If the table has no such columns this value is None. If a value was generated for the operation, it is non-None.

This value is only valid for a put operation on a table with an identity column or string as uuid column.

Returns the generated value.

get_success()
Returns the flag indicates whether the operation succeeded. A put or delete operation may be unsuccessful if the condition is not matched.

Returns True if the operation succeeded.

Return type bool

get_version()
Returns the version of the new row for put operation, or None if put operations did not succeed or the operation is delete operation.

Returns the version.

Return type *Version*

OperationThrottlingException

exception borneo.**OperationThrottlingException**(*message*)
Cloud service only.

An exception that is thrown when a non-data operation is throttled. This can happen if an application attempts too many control operations such as table creation, deletion, or similar methods. Such operations do not use throughput or capacity provisioned for a given table but they consume system resources and their use is limited.

Operations resulting in this exception can be retried but it is recommended that callers use a relatively large delay before retrying in order to minimize the chance that a retry will also be throttled.

PreparedStatement

class borneo.**PreparedStatement**(*sql_text*, *query_plan*, *topology_info*, *proxy_statement*, *driver_plan*, *num_iterators*, *num_registers*, *external_vars*, *namespace*, *table_name*, *operation*)

Bases: `object`

A class encapsulating a prepared query statement. It includes state that can be sent to a server and executed without re-parsing the query. It includes bind variables which may be set for each successive use of the query. The prepared query itself is read-only but this object contains a dictionary of bind variables and is not thread-safe if variables are used.

PreparedStatement instances are returned inside *PrepareResult* objects returned by *NoSQLHandle.prepare()*

A single instance of PreparedStatement is thread-safe if bind variables are not used. If bind variables are to be used and the statement shared among threads additional instances of PreparedStatement can be constructed using *copy_statement()*.

Methods Summary

clear_variables()	Clears all bind variables from the statement.
copy_statement()	Returns a new instance that shares this object's prepared query, which is immutable, but does not share its variables.
get_query_plan()	Returns a string representation of the query execution plan, if it was requested in the *PrepareRequest*; None otherwise.
get_sql_text()	Returns the SQL text of this PreparedStatement.
get_variables()	Returns the dictionary of variables to use for a prepared query with variables.
set_variable(variable, value)	Binds an external variable to a given value.

Methods Documentation

clear_variables()
Clears all bind variables from the statement.

copy_statement()
Returns a new instance that shares this object's prepared query, which is immutable, but does not share its variables.

Returns a new PreparedStatement using this instance's prepared query. Bind variables are uninitialized.

Return type *PreparedStatement*

get_query_plan()
Returns a string representation of the query execution plan, if it was requested in the *PrepareRequest*; None otherwise.

Returns the string representation of the query execution plan.

Return type bool

get_sql_text()
Returns the SQL text of this PreparedStatement.

Returns the SQL text of this PreparedStatement.

Return type str

get_variables()
Returns the dictionary of variables to use for a prepared query with variables.

Returns the dictionary.

Return type dict

set_variable(*variable*, *value*)
Binds an external variable to a given value. The variable is identified by its name or its position within the query string. The variable that appears first in the query text has position 1, the variable that appears second has position 2 and so on.

Parameters

- **variable** (*str or int*) – the name or the position of the variable.
- **value** (*a value matching the type of the field*) – the value.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if variable is not a string or positive integer.

PrepareRequest

class borneo.**PrepareRequest**
Bases: borneo.operations.Request

A request that encapsulates a query prepare call. Query preparation allows queries to be compiled (prepared) and reused, saving time and resources. Use of prepared queries vs direct execution of query strings is highly recommended.

Prepared queries are implemented as *PreparedStatement* which supports bind variables in queries which can be used to more easily reuse a query by parameterization.

The statement is required parameter.

Methods Summary

get_compartment()	Cloud service only.
get_query_plan()	Returns whether a printout of the query execution plan should be include in the *PrepareResult*.
get_statement()	Returns the query statement.
get_timeout()	Returns the timeout to use for the operation, in milliseconds.
set_compartment(compartment)	Cloud service only.
set_get_query_plan(get_query_plan)	Sets whether a printout of the query execution plan should be included in the *PrepareResult*.
set_statement(statement)	Sets the query statement.
set_timeout(timeout_ms)	Sets the request timeout value, in milliseconds.

Methods Documentation

get_compartment()
Cloud service only.

Get the compartment id or name if set for the request.

Returns compartment id or name if set for the request, otherwise None if not set.

Return type str

get_query_plan()
Returns whether a printout of the query execution plan should be include in the *PrepareResult*.

Returns whether a printout of the query execution plan should be include in the *PrepareResult*.

Return type bool

get_statement()
Returns the query statement.

Returns the statement, or None if it has not been set.

Return type str

get_timeout()
Returns the timeout to use for the operation, in milliseconds. A value of 0 indicates that the timeout has not been set.

Returns the value.

Return type int

set_compartment(*compartment*)
Cloud service only.

Sets the name or id of a compartment to be used for this operation.

The compartment may be specified as either a name (or path for nested compartments) or as an id (OCID). A name (vs id) can only be used when authenticated using a specific user identity. It is *not* available if authenticated as an Instance Principal which can be done when calling the service from a compute instance in the Oracle Cloud Infrastructure. See *borneo.iam.SignatureProvider.create_with_instance_principal()*.

Parameters **compartment** (*str*) – the compartment name or id. If using a nested compartment, specify the full compartment path compartmentA.compartmentB, but exclude the name of the root compartment (tenant).

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if compartment is not a str.

set_get_query_plan(*get_query_plan*)
Sets whether a printout of the query execution plan should be included in the *PrepareResult*.

Parameters **get_query_plan** (*bool*) – True if a printout of the query execution plan should be included in the *PrepareResult*. False otherwise.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if get_query_plan is not a boolean.

set_statement(*statement*)
Sets the query statement.

Parameters **statement** (*str*) – the query statement.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if statement is not a string.

set_timeout(*timeout_ms*)
Sets the request timeout value, in milliseconds. This overrides any default value set in *NoSQLHandleConfig*. The value must be positive.

Parameters **timeout_ms** (*int*) – the timeout value, in milliseconds.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if the timeout value is less than or equal to 0.

PrepareResult

class borneo.**PrepareResult**
Bases: borneo.operations.Result

The result of a prepare operation. The returned *PreparedStatement* can be re-used for query execution using *QueryRequest.set_prepared_statement()*

Methods Summary

get_prepared_statement()	Returns the value of the prepared statement.
get_read_kb()	Returns the read throughput consumed by this operation, in KBytes.
get_read_units()	Returns the read throughput consumed by this operation, in read units.
get_write_kb()	Returns the write throughput consumed by this operation, in KBytes.
get_write_units()	Returns the write throughput consumed by this operation, in write units.

Methods Documentation

get_prepared_statement()
Returns the value of the prepared statement.

Returns the value of the prepared statement.

Return type *PreparedStatement*

get_read_kb()
Returns the read throughput consumed by this operation, in KBytes. This is the actual amount of data read by the operation. The number of read units consumed is returned by *get_read_units()* which may be a larger number if the operation used Consistency.ABSOLUTE.

Returns the read KBytes consumed.

Return type int

get_read_units()
Returns the read throughput consumed by this operation, in read units. This number may be larger than that returned by *get_read_kb()* if the operation used Consistency.ABSOLUTE.

Returns the read units consumed.

Return type int

get_write_kb()
Returns the write throughput consumed by this operation, in KBytes.

Returns the write KBytes consumed.

Return type int

get_write_units()
Returns the write throughput consumed by this operation, in write units.

Returns the write units consumed.

Return type int

PutOption

class borneo.**PutOption**
Bases: object

Set the put option for put requests.

Attributes Summary

IF_ABSENT	Set PutOption.IF_ABSENT to perform put if absent operation.
IF_PRESENT	Set PutOption.IF_PRESENT to perform put if present operation.
IF_VERSION	Set PutOption.IF_VERSION to perform put if version operation.

Attributes Documentation

IF_ABSENT = 0
Set PutOption.IF_ABSENT to perform put if absent operation.

IF_PRESENT = 1
Set PutOption.IF_PRESENT to perform put if present operation.

IF_VERSION = 2
Set PutOption.IF_VERSION to perform put if version operation.

PutRequest

class borneo.**PutRequest**
Bases: borneo.operations.WriteRequest

Represents the input to a *NoSQLHandle.put()* operation.

This request can be used to perform unconditional and conditional puts:

Overwrite any existing row. This is the default.

Succeed only if the row does not exist. Use PutOption.IF_ABSENT for this case.

Succeed only if the row exists. Use PutOption.IF_PRESENT for this case.

Succeed only if the row exists and its *Version* matches a specific *Version*. Use PutOption.IF_VERSION for this case and *set_match_version()* to specify the version to match.

Information about the existing row can be returned on failure of a put operation using PutOption.IF_VERSION or PutOption.IF_ABSENT by using *set_return_row()*. Requesting this information incurs additional cost and may affect operation latency.

On a successful operation the *Version* returned by *PutResult.get_version()* is non-none. Additional information, such as previous row information, may be available in *PutResult*.

The table name and value are required parameters.

Methods Summary

get_compartment()	Cloud service only.
get_durability()	On-premise only.
get_exact_match()	Returns whether the value must be an exact match to the table schema or not.
get_identity_cache_size()	Gets the number of generated identity values that are requested from the server during a put if set in this request.
get_match_version()	Returns the *Version* used for a match on a conditional put.
get_option()	Returns the option specified for the put.
get_return_row()	Returns whether information about the exist row should be returned on failure because of a version mismatch or failure of an "if absent" operation.
get_table_name()	Returns the table name to use for the operation.
get_timeout()	Returns the timeout to use for the operation, in milliseconds.
get_ttl()	Returns the *TimeToLive* value, if set.
get_update_ttl()	Returns True if the operation should update the ttl.
get_use_table_default_ttl()	Returns whether or not to update the row's time to live (TTL) based on a table default value if the row exists.
get_value()	Returns the value of the row to be used.
set_compartment(compartment)	Cloud service only.
set_durability(durability)	On-premise only.
set_exact_match(exact_match)	If True the value must be an exact match for the table schema or the operation will fail.
set_identity_cache_size(identity_cache_size)	Sets the number of generated identity values that are requested from the server during a put.
set_match_version(version)	Sets the *Version* to use for a conditional put operation.
set_option(option)	Sets the option for the put.
set_return_row(return_row)	Sets whether information about the exist row should be returned on failure because of a version mismatch or failure of an "if absent" operation.

Continued on next page

Table 25 – continued from previous page

set_table_name(table_name)	Sets the table name to use for the operation.
set_timeout(timeout_ms)	Sets the request timeout value, in milliseconds.
set_ttl(ttl)	Sets the *TimeToLive* value, causing the time to live on the row to be set to the specified value on put.
set_use_table_default_ttl(update_ttl)	If value is True, and there is an existing row, causes the operation to update the time to live (TTL) value of the row based on the Table's default TTL if set.
set_value(value)	Sets the value to use for the put operation.
set_value_from_json(json_value)	Sets the value to use for the put operation based on a JSON string.

Methods Documentation

get_compartment()
Cloud service only.

Get the compartment id or name if set for the request.

Returns compartment id or name if set for the request, otherwise None if not set.

Return type str

get_durability()
On-premise only. Gets the durability to use for the operation or None if not set :returns: the Durability :versionadded: 5.3.0

get_match_version()
Returns the *Version* used for a match on a conditional put.

Returns the Version or None if not set.

Return type *Version*

get_option()
Returns the option specified for the put.

Returns the option specified.

Return type *PutOption*

get_return_row()
Returns whether information about the exist row should be returned on failure because of a version mismatch or failure of an "if absent" operation. If no option is set via *set_option()* or the option is PutOption.IF_PRESENT the value of this parameter is ignored and there will not be any return information.

Returns True if information should be returned.

Return type bool

get_table_name()
Returns the table name to use for the operation.

Returns the table name, or None if not set.

Returns str

get_timeout()
Returns the timeout to use for the operation, in milliseconds. A value of 0 indicates that the timeout has not been set.

Returns the timeout value.

Return type int

get_ttl()
Returns the *TimeToLive* value, if set.

Returns the *TimeToLive* if set, None otherwise.

Return type *TimeToLive*

get_update_ttl()
Returns True if the operation should update the ttl.

Returns True if the operation should update the ttl.

Return type bool

get_use_table_default_ttl()
Returns whether or not to update the row's time to live (TTL) based on a table default value if the row exists. By default updates of existing rows do not affect that row's TTL.

Returns whether or not to update the row's TTL based on a table default value if the row exists.

Return type bool

get_value()
Returns the value of the row to be used.

Returns the value, or None if not set.

Return type dict

set_compartment(*compartment*)
Cloud service only.

Sets the name or id of a compartment to be used for this operation.

The compartment may be specified as either a name (or path for nested compartments) or as an id (OCID). A name (vs id) can only be used when authenticated using a specific user identity. It is *not* available if authenticated as an Instance Principal which can be done when calling the service from a compute instance in the Oracle Cloud Infrastructure. See *borneo.iam.SignatureProvider.create_with_instance_principal()*.

Parameters **compartment** (*str*) – the compartment name or id. If using a nested compartment, specify the full compartment path compartmentA.compartmentB, but exclude the name of the root compartment (tenant).

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if compartment is not a str.

set_durability(*durability*)
On-premise only. Sets the durability to use for the operation.

Parameters **durability** (Durability) – the Durability to use

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if Durability is not valid

Versionadded 5.3.0

set_match_version(*version*)
Sets the *Version* to use for a conditional put operation. The Version is usually obtained from *GetResult.get_version()* or other method that returns a Version. When set, the put operation

will succeed only if the row exists and its Version matches the one specified. This condition exists to allow an application to ensure that it is updating a row in an atomic read-modify-write cycle. Using this mechanism incurs additional cost.

Parameters **version** (Version) – the Version to match.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if version is not an instance of Version.

set_option(*option*)
Sets the option for the put.

Parameters **option** (PutOption) – the option to set.

Returns self.

set_return_row(*return_row*)
Sets whether information about the exist row should be returned on failure because of a version mismatch or failure of an "if absent" operation.

Parameters **return_row** (*bool*) – set to True if information should be returned.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if return_row is not True or False.

set_table_name(*table_name*)
Sets the table name to use for the operation.

Parameters **table_name** (*str*) – the table name.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if table_name is not a string.

set_timeout(*timeout_ms*)
Sets the request timeout value, in milliseconds. This overrides any default value set in *NoSQLHandleConfig*. The value must be positive.

Parameters **timeout_ms** (*int*) – the timeout value, in milliseconds.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if the timeout value is less than or equal to 0.

set_ttl(*ttl*)
Sets the *TimeToLive* value, causing the time to live on the row to be set to the specified value on put. This value overrides any default time to live setting on the table.

Parameters **ttl** (TimeToLive) – the time to live.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if ttl is not an instance of TimeToLive.

set_use_table_default_ttl(*update_ttl*)
If value is True, and there is an existing row, causes the operation to update the time to live (TTL) value of the row based on the Table's default TTL if set. If the table has no default TTL this state has no effect. By default updating an existing row has no effect on its TTL.

Parameters **update_ttl** (*bool*) – True or False.

Returns self.

Raises *IllegalArgumentException* – raises the exception if update_ttl is not True or False.

set_value(*value*)
Sets the value to use for the put operation. This is a required parameter and must be set using this method or *set_value_from_json()*

Parameters **value** (*dict*) – the row value.

Returns self.

Raises *IllegalArgumentException* – raises the exception if value is not a dictionary.

set_value_from_json(*json_value*)
Sets the value to use for the put operation based on a JSON string. The string is parsed for validity and stored internally as a dict. This is a required parameter and must be set using this method or *set_value()*

Parameters **json_value** (*str*) – the row value as a JSON string.

Returns self.

Raises *IllegalArgumentException* – raises the exception if json_value is not a string.

PutResult

class borneo.**PutResult**
Bases: borneo.operations.WriteResult

Represents the result of a *NoSQLHandle.put()* operation.

On a successful operation the value returned by *get_version()* is non-none. On failure that value is None. Information about the existing row on failure may be available using *get_existing_value()* and *get_existing_version()*, depending on the use of *PutRequest.set_return_row()* and whether the put had an option set using *PutRequest.set_option()*.

Methods Summary

get_existing_modification_time()	Returns the existing row modification time if available.
get_existing_value()	Returns the existing row value if available.
get_existing_version()	Returns the existing row *Version* if available.
get_generated_value()	Returns the value generated if the operation created a new value.
get_read_kb()	Returns the read throughput consumed by this operation, in KBytes.
get_read_units()	Returns the read throughput consumed by this operation, in read units.
get_version()	Returns the *Version* of the new row if the operation was successful.
get_write_kb()	Returns the write throughput consumed by this operation, in KBytes.
get_write_units()	Returns the write throughput consumed by this operation, in write units.

Methods Documentation

get_existing_modification_time()
Returns the existing row modification time if available. It will be available if the conditional put operation failed and the request specified that return information be returned using *PutRequest.set_return_row()*. A value of -1 indicates this feature is not available at the connected server.

Returns the modification time in milliseconds since January 1, 1970

Return type int

Versionadded 5.3.0

get_existing_value()
Returns the existing row value if available. This value will only be available if the conditional put operation failed and the request specified that return information be returned using *PutRequest.set_return_row()*.

Returns the value.

Return type dict

get_existing_version()
Returns the existing row *Version* if available. This value will only be available if the conditional put operation failed and the request specified that return information be returned using *PutRequest.set_return_row()*.

Returns the *Version*.

Return type *Version*

get_generated_value()
Returns the value generated if the operation created a new value. This can happen if the table contains an identity column or string column declared as a generated UUID. If the table has no such columns this value is None. If a value was generated for the operation, it is non-None.

Returns the generated value.

get_read_kb()
Returns the read throughput consumed by this operation, in KBytes. This is the actual amount of data read by the operation. The number of read units consumed is returned by *get_read_units()* which may be a larger number because this was an update operation.

Returns the read KBytes consumed.

Return type int

get_read_units()
Returns the read throughput consumed by this operation, in read units. This number may be larger than that returned by *get_read_kb()* because it was an update operation.

Returns the read units consumed.

Return type int

get_version()
Returns the *Version* of the new row if the operation was successful. If the operation failed None is returned.

Returns the *Version* on success, None on failure.

Return type *Version*

get_write_kb()
Returns the write throughput consumed by this operation, in KBytes.

Returns the write KBytes consumed.

Return type int

get_write_units()
Returns the write throughput consumed by this operation, in write units.

Returns the write units consumed.

Return type int

QueryRequest

class borneo.**QueryRequest**
Bases: borneo.operations.Request

A request that represents a query. A query may be specified as either a textual SQL statement (a String) or a prepared query (an instance of *PreparedStatement*), which may include bind variables.

For performance reasons prepared queries are preferred for queries that may be reused. This is because prepared queries bypass query compilation. They also allow for parameterized queries using bind variables.

To compute and retrieve the full result set of a query, the same QueryRequest instance will, in general, have to be executed multiple times (via *NoSQLHandle.query()*). Each execution returns a *QueryResult*, which contains a subset of the result set. The following code snippet illustrates a typical query execution:

```
handle = ...
request = QueryRequest().set_statement('SELECT * FROM foo')
while True:
    result = handle.query(request)
    results = result.get_results()
    # do something with the results
    if request.is_done():
        break
```

Notice that a batch of results returned by a QueryRequest execution may be empty. This is because during each execution the query is allowed to read or write a maximum number of bytes. If this maximum is reached, execution stops. This can happen before any result was generated (for example, if none of the rows read satisfied the query conditions).

If an application wishes to terminate query execution before retrieving all of the query results, it should call *close()* in order to release any local resources held by the query. This also allows the application to reuse the QueryRequest instance to run the same query from the beginning or a different query.

QueryRequest instances are not thread-safe. That is, if two or more application threads need to run the same query concurrently, they must create and use their own QueryRequest instances.

The statement or prepared_statement is required parameter.

Methods Summary

close()	Terminates the query execution and releases any memory consumed by the query at the driver.
get_compartment()	Cloud service only.

Continued on next page

Table 27 – continued from previous page

get_consistency()	Returns the consistency set for this request, or None if not set.
get_limit()	Returns the limit on number of items returned by the operation.
get_math_context()	Returns the Context used for Decimal operations.
get_max_memory_consumption()	Returns the maximum number of memory bytes that may be consumed by the statement at the driver for operations such as duplicate elimination (which may be required due to the use of an index on a list or map) and sorting (sorting by distance when a query contains a geo_near() function).
get_max_read_kb()	Returns the limit on the total data read during this operation, in KB.
get_max_write_kb()	Returns the limit on the total data written during this operation, in KB.
get_prepared_statement()	Returns the prepared query statement.
get_statement()	Returns the query statement.
get_timeout()	Returns the timeout to use for the operation, in milliseconds.
is_done()	Returns True if the query execution is finished, i.e., there are no more query results to be generated.
set_compartment(compartment)	Cloud service only.
set_consistency(consistency)	Sets the consistency to use for the operation.
set_limit(limit)	Sets the limit on number of items returned by the operation.
set_math_context(math_context)	Sets the Context used for Decimal operations.
set_max_memory_consumption(memory_consumption)	Sets the maximum number of memory bytes that may be consumed by the statement at the driver for operations such as duplicate elimination (which may be required due to the use of an index on a list or map) and sorting.
set_max_read_kb(max_read_kb)	Sets the limit on the total data read during this operation, in KB.
set_max_write_kb(max_write_kb)	Sets the limit on the total data written during this operation, in KB.
set_prepared_statement(value)	Sets the prepared query statement.
set_statement(statement)	Sets the query statement.
set_timeout(timeout_ms)	Sets the request timeout value, in milliseconds.

Methods Documentation

close()

Terminates the query execution and releases any memory consumed by the query at the driver. An application should use this method if it wishes to terminate query execution before retrieving all of the query results.

get_compartment()

Cloud service only.

Get the compartment id or name if set for the request.

Returns compartment id or name if set for the request, otherwise None if not set.

Return type str

get_consistency()
Returns the consistency set for this request, or None if not set.

Returns the consistency

Return type *Consistency*

get_limit()
Returns the limit on number of items returned by the operation. If not set by the application this value will be 0 which means no limit.

Returns the limit, or 0 if not set.

Return type int

get_math_context()
Returns the Context used for Decimal operations.

Returns the Context used for Decimal operations.

Return type Context

get_max_memory_consumption()
Returns the maximum number of memory bytes that may be consumed by the statement at the driver for operations such as duplicate elimination (which may be required due to the use of an index on a list or map) and sorting (sorting by distance when a query contains a geo_near() function). Such operations may consume a lot of memory as they need to cache the full result set at the client memory. The default value is 100MB.

Returns the maximum number of memory bytes.

Return type long

get_max_read_kb()
Returns the limit on the total data read during this operation, in KB. If not set by the application this value will be 0 which means no application-defined limit.

Returns the limit, or 0 if not set.

Return type int

get_max_write_kb()
Returns the limit on the total data written during this operation, in KB. If not set by the application this value will be 0 which means no application-defined limit.

Returns the limit, or 0 if not set.

Return type int

get_prepared_statement()
Returns the prepared query statement.

Returns the statement, or None if it has not been set.

Return type *PreparedStatement*

get_statement()
Returns the query statement.

Returns the statement, or None if it has not been set.

Return type str

get_timeout()
Returns the timeout to use for the operation, in milliseconds. A value of 0 indicates that the timeout has not been set.

Returns the timeout value.

Return type int

is_done()
Returns True if the query execution is finished, i.e., there are no more query results to be generated. Otherwise False.

Returns Whether the query execution is finished or not.

Return type bool

set_compartment(*compartment*)
Cloud service only.

Sets the name or id of a compartment to be used for this operation.

The compartment may be specified as either a name (or path for nested compartments) or as an id (OCID). A name (vs id) can only be used when authenticated using a specific user identity. It is *not* available if authenticated as an Instance Principal which can be done when calling the service from a compute instance in the Oracle Cloud Infrastructure. See *borneo.iam.SignatureProvider.create_with_instance_principal()*.

Parameters **compartment** (*str*) – the compartment name or id. If using a nested compartment, specify the full compartment path compartmentA.compartmentB, but exclude the name of the root compartment (tenant).

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if compartment is not a str.

set_consistency(*consistency*)
Sets the consistency to use for the operation.

Parameters **consistency** (Consistency) – the consistency.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if consistency is not Consistency.ABSOLUTE or Consistency.EVENTUAL.

set_limit(*limit*)
Sets the limit on number of items returned by the operation. This allows an operation to return less than the default amount of data.

Parameters **limit** (*int*) – the limit in terms of number of items returned.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if the limit is a negative number.

set_math_context(*math_context*)
Sets the Context used for Decimal operations.

Parameters **math_context** (*Context*) – the Context used for Decimal operations.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if math_context is not an instance of Context.

set_max_memory_consumption(*memory_consumption*)
Sets the maximum number of memory bytes that may be consumed by the statement at the driver for operations such as duplicate elimination (which may be required due to the use of an index on a list or

map) and sorting. Such operations may consume a lot of memory as they need to cache the full result set or a large subset of it at the client memory. The default value is 1GB.

Parameters **memory_consumption** (*long*) – the maximum number of memory bytes that may be consumed by the statement at the driver for blocking operations.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if memory_consumption is a negative number or 0.

set_max_read_kb(*max_read_kb*)

Sets the limit on the total data read during this operation, in KB. This value can only reduce the system defined limit. This limit is independent of read units consumed by the operation.

It is recommended that for tables with relatively low provisioned read throughput that this limit be reduced to less than or equal to one half of the provisioned throughput in order to avoid or reduce throttling exceptions.

Parameters **max_read_kb** (*int*) – the limit in terms of number of KB read during this operation.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if the max_read_kb value is less than 0.

set_max_write_kb(*max_write_kb*)

Sets the limit on the total data written during this operation, in KB. This limit is independent of write units consumed by the operation.

Parameters **max_write_kb** (*int*) – the limit in terms of number of KB written during this operation.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if the max_write_kb value is less than 0.

set_prepared_statement(*value*)

Sets the prepared query statement.

Parameters **value** (PreparedStatement) – the prepared query statement or the result of a prepare request.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if value is not an instance of PrepareResult or PreparedStatement.

set_statement(*statement*)

Sets the query statement.

Parameters **statement** (*str*) – the query statement.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if statement is not a string.

set_timeout(*timeout_ms*)

Sets the request timeout value, in milliseconds. This overrides any default value set in *NoSQLHandleConfig*. The value must be positive.

Parameters **timeout_ms** (*int*) – the timeout value, in milliseconds.

Returns self.

Raises *IllegalArgumentException* – raises the exception if the timeout value is less than or equal to 0.

QueryResult

class borneo.**QueryResult**(*request*, *computed=True*)

Bases: borneo.operations.Result

QueryResult comprises a list of dict instances representing the query results.

The shape of the values is based on the schema implied by the query. For example a query such as "SELECT * FROM ..." that returns an intact row will return values that conform to the schema of the table. Projections return instances that conform to the schema implied by the statement. UPDATE queries either return values based on a RETURNING clause or, by default, the number of rows affected by the statement.

A single QueryResult does not imply that all results for the query have been returned. If the value returned by *QueryRequest.is_done()* is False there are additional results available. This can happen even if there are no values in the returned QueryResult. The best way to use *QueryRequest* and *QueryResult* is to perform operations in a loop, for example:

```
handle = ...
request = QueryRequest().set_statement('SELECT * FROM foo')
while True:
    result = handle.query(request)
    results = result.get_results()
    # do something with the results
    if request.is_done():
        break
```

Modification queries either return values based on a RETURNING clause or, by default, return the number of rows affected by the statement in a dictionary. INSERT queries with no RETURNING clause return a dictionary indicating the number of rows inserted, for example {'NumRowsInserted': 5}. UPDATE queries with no RETURNING clause return a dictionary indicating the number of rows updated, for example {'NumRowsUpdated': 3}. DELETE queries with no RETURNING clause return a dictionary indicating the number of rows deleted, for example {'numRowsDeleted': 2}.

Methods Summary

get_continuation_key()	Returns the continuation key that can be used to obtain more results if non-none.
get_read_kb()	Returns the read throughput consumed by this operation, in KBytes.
get_read_units()	Returns the read throughput consumed by this operation, in read units.
get_results()	Returns a list of results for the query.
get_write_kb()	Returns the write throughput consumed by this operation, in KBytes.
get_write_units()	Returns the write throughput consumed by this operation, in write units.

Methods Documentation

get_continuation_key()
Returns the continuation key that can be used to obtain more results if non-none.

Returns the continuation key, or None if there are no further values to return.

Return type bytearray

get_read_kb()
Returns the read throughput consumed by this operation, in KBytes. This is the actual amount of data read by the operation. The number of read units consumed is returned by *get_read_units()* which may be a larger number if the operation used Consistency.ABSOLUTE.

Returns the read KBytes consumed.

Return type int

get_read_units()
Returns the read throughput consumed by this operation, in read units. This number may be larger than that returned by *get_read_kb()* if the operation used Consistency.ABSOLUTE.

Returns the read units consumed.

Return type int

get_results()
Returns a list of results for the query. It is possible to have an empty list and a non-none continuation key.

Returns a list of results for the query.

Return type list(dict)

get_write_kb()
Returns the write throughput consumed by this operation, in KBytes.

Returns the write KBytes consumed.

Return type int

get_write_units()
Returns the write throughput consumed by this operation, in write units.

Returns the write units consumed.

Return type int

QueryIterableResult

class borneo.**QueryIterableResult**(*request*, *handle*)
Bases: borneo.operations.Result

QueryIterableResult comprises an iterable list of dict instances representing all the query results.

The shape of the values is based on the schema implied by the query. For example a query such as "SELECT * FROM ..." that returns an intact row will return values that conform to the schema of the table. Projections return instances that conform to the schema implied by the statement. UPDATE queries either return values based on a RETURNING clause or, by default, the number of rows affected by the statement.

Each iterator from QueryIterableResult will iterate over all results of the query.

```
handle = ...
request = QueryRequest().set_statement('SELECT * FROM foo')
qiresult = handle.query-iterable(request)
for row in qiresult:
    # do something with the result row
    print(row)
```

Modification queries either return values based on a RETURNING clause or, by default, return the number of rows affected by the statement in a dictionary. INSERT queries with no RETURNING clause return a dictionary indicating the number of rows inserted, for example {'NumRowsInserted': 5}. UPDATE queries with no RETURNING clause return a dictionary indicating the number of rows updated, for example {'NumRowsUpdated': 3}. DELETE queries with no RETURNING clause return a dictionary indicating the number of rows deleted, for example {'numRowsDeleted': 2}.

Versionadded 5.3.6

Methods Summary

get_read_kb()	Returns the read throughput consumed by this operation, in KBytes.
get_read_units()	Returns the read throughput consumed by this operation, in read units.
get_write_kb()	Returns the write throughput consumed by this operation, in KBytes.

Methods Documentation

get_read_kb()
Returns the read throughput consumed by this operation, in KBytes. This is the cumulative actual amount of data read by the operation since the beginning of the iterable. The number of read units consumed is returned by *get_read_units()* which may be a larger number if the operation used Consistency.ABSOLUTE.

Returns the read KBytes consumed.

Return type int

get_read_units()
Returns the read throughput consumed by this operation, in read units. This is the cumulative amount since the beginning of the iterable. This number may be larger than that returned by *get_read_kb()* if the operation used Consistency.ABSOLUTE.

Returns the read units consumed.

Return type int

get_write_kb()
Returns the write throughput consumed by this operation, in KBytes.

Returns the write KBytes consumed.

Return type int

ReadThrottlingException

exception borneo.**ReadThrottlingException**(*message*)

Cloud service only.

This exception indicates that the provisioned read throughput has been exceeded.

Operations resulting in this exception can be retried but it is recommended that callers use a delay before retrying in order to minimize the chance that a retry will also be throttled. Applications should attempt to avoid throttling exceptions by rate limiting themselves to the degree possible.

Region

class borneo.**Region**(*region_id*)

Bases: object

Cloud service only.

The class represents a region of Oracle NoSQL Database Cloud.

Methods Summary

endpoint()	Returns the NoSQL Database Cloud Service endpoint string for this region.

Methods Documentation

endpoint()

Returns the NoSQL Database Cloud Service endpoint string for this region.

Returns NoSQL Database Cloud Service endpoint string.

Return type str

Raises ***IllegalArgumentException*** – raises the exception if region_id is unknown.

Regions

class borneo.**Regions**

Bases: object

Cloud service only.

The class contains the regions in the Oracle Cloud Infrastructure at the time of this release. The Oracle NoSQL Database Cloud Service is not available in all of these regions. For a definitive list of regions in which the Oracle NoSQL Database Cloud Service is available see Data Regions for Platform and Infrastructure Services.

A Region may be provided to *NoSQLHandleConfig* to configure a handle to communicate in a specific Region.

The string-based endpoints associated with regions for the Oracle NoSQL Database Cloud Service are of the format:

```
https://nosql.{region}.oci.{secondLevelDomain}
```

Examples of known second level domains include

- oraclecloud.com
- oraclegovcloud.com
- oraclegovcloud.uk

For example, this is a valid endpoint for the Oracle NoSQL Database Cloud Service in the U.S. East region:

```
https://nosql.us-ashburn-1.oci.oraclecloud.com
```

If the Oracle NoSQL Database Cloud Service becomes available in a region not listed here it is possible to connect to that region using the endpoint string rather than a Region.

For more information about Oracle Cloud Infrastructure regions see Regions and Availability Domains.

Attributes Summary

AP_MELBOURNE_1	Region Location: Melbourne, Australia
AP_MUMBAI_1	Region Location: Mumbai, India
AP_OSAKA_1	Region Location: Osaka, Japan
AP_SEOUL_1	Region Location: Seoul, South Korea
AP_SYDNEY_1	Region Location: Sydney, Australia
AP_TOKYO_1	Region Location: Tokyo, Japan
CA_MONTREAL_1	Region Location: Montreal, Canada
CA_TORONTO_1	Region Location: Toronto, Canada
EU_AMSTERDAM_1	Region Location: Amsterdam, Netherlands
EU_FRANKFURT_1	Region Location: Frankfurt, Germany
EU_ZURICH_1	Region Location: Zurich, Switzerland
GOV_REGIONS	A dict containing the government regions.
ME_JEDDAH_1	Region Location: Jeddah, Saudi Arabia
OC1_REGIONS	A dict containing the OC1 regions.
OC4_REGIONS	A dict containing the OC4 regions.
SA_SAOPAULO_1	Region Location: Sao Paulo, Brazil
UK_GOV_LONDON_1	Region Location: London, United Kingdom
UK_LONDON_1	Region Location: London, United Kingdom
US_ASHBURN_1	Region Location: Ashburn, VA
US_GOV_ASHBURN_1	Region Location: Ashburn, VA
US_GOV_CHICAGO_1	Region Location: Chicago, IL
US_GOV_PHOENIX_1	Region Location: Phoenix, AZ
US_LANGLEY_1	Region Location: Ashburn, VA
US_LUKE_1	Region Location: Phoenix, AZ
US_PHOENIX_1	Region Location: Phoenix, AZ

Methods Summary

from_region_id(region_id)	Returns the Region associated with the string value supplied, or None if the string does not represent a known region.

Attributes Documentation

AP_MELBOURNE_1 = <borneo.config.Region object>
Region Location: Melbourne, Australia

AP_MUMBAI_1 = <borneo.config.Region object>
Region Location: Mumbai, India

AP_OSAKA_1 = <borneo.config.Region object>
Region Location: Osaka, Japan

AP_SEOUL_1 = <borneo.config.Region object>
Region Location: Seoul, South Korea

AP_SYDNEY_1 = <borneo.config.Region object>
Region Location: Sydney, Australia

AP_TOKYO_1 = <borneo.config.Region object>
Region Location: Tokyo, Japan

CA_MONTREAL_1 = <borneo.config.Region object>
Region Location: Montreal, Canada

CA_TORONTO_1 = <borneo.config.Region object>
Region Location: Toronto, Canada

EU_AMSTERDAM_1 = <borneo.config.Region object>
Region Location: Amsterdam, Netherlands

EU_FRANKFURT_1 = <borneo.config.Region object>
Region Location: Frankfurt, Germany

EU_ZURICH_1 = <borneo.config.Region object>
Region Location: Zurich, Switzerland

GOV_REGIONS = {'us-gov-ashburn-1': <borneo.config.Region object>, 'us-gov-chicago-1':
A dict containing the government regions.

ME_JEDDAH_1 = <borneo.config.Region object>
Region Location: Jeddah, Saudi Arabia

OC1_REGIONS = {'ap-chuncheon-1': <borneo.config.Region object>, 'ap-hyderabad-1': <b
A dict containing the OC1 regions.

OC4_REGIONS = {'uk-gov-cardiff-1': <borneo.config.Region object>, 'uk-gov-london-1':
A dict containing the OC4 regions.

SA_SAOPAULO_1 = <borneo.config.Region object>
Region Location: Sao Paulo, Brazil

UK_GOV_LONDON_1 = <borneo.config.Region object>
Region Location: London, United Kingdom

UK_LONDON_1 = <borneo.config.Region object>
Region Location: London, United Kingdom

US_ASHBURN_1 = <borneo.config.Region object>
Region Location: Ashburn, VA

US_GOV_ASHBURN_1 = <borneo.config.Region object>
Region Location: Ashburn, VA

US_GOV_CHICAGO_1 = <borneo.config.Region object>
Region Location: Chicago, IL

US_GOV_PHOENIX_1 = <borneo.config.Region object>
Region Location: Phoenix, AZ

US_LANGLEY_1 = <borneo.config.Region object>
Region Location: Ashburn, VA

US_LUKE_1 = <borneo.config.Region object>
Region Location: Phoenix, AZ

US_PHOENIX_1 = <borneo.config.Region object>
Region Location: Phoenix, AZ

Methods Documentation

static from_region_id(*region_id*)
Returns the Region associated with the string value supplied, or None if the string does not represent a known region.

Parameters **region_id** (*str*) – the string value of the region.

Returns the Region or None if the string does not represent a Region.

Return type *Region*

Request

class borneo.**Request**
Bases: object

A request is a class used as a base for all requests types. Public state and methods are implemented by extending classes.

Methods Summary

—

Methods Documentation

RequestSizeLimitException

exception borneo.**RequestSizeLimitException**(*message*)
Cloud service only.

Thrown to indicate that the size of a Request exceeds the system defined limit.

RequestTimeoutException

exception borneo.**RequestTimeoutException**(*message*, *timeout_ms=0*, *cause=None*)
Thrown when a request cannot be processed because the configured timeout interval is exceeded. If a retry handler is configured it is possible that the request has been retried a number of times before the timeout occurs.

ResourceExistsException

exception borneo.**ResourceExistsException**(*message*)
The operation attempted to create a resource but it already exists.

ResourcePrincipalClaimKeys

class borneo.**ResourcePrincipalClaimKeys**
Bases: object

Claim keys in the resource principal session token(RPST).

They can be used to retrieve resource principal metadata such as its compartment and tenancy OCID.

Attributes Summary

COMPARTMENT_ID_CLAIM_KEY	The claim name that the RPST holds for the resource compartment.
TENANT_ID_CLAIM_KEY	The claim name that the RPST holds for the resource tenancy.

Attributes Documentation

COMPARTMENT_ID_CLAIM_KEY = 'res_compartment'
The claim name that the RPST holds for the resource compartment. This can be passed to *borneo.iam.SignatureProvider.get_resource_principal_claim()* to retrieve the resource's compartment OCID.

TENANT_ID_CLAIM_KEY = 'res_tenant'
The claim name that the RPST holds for the resource tenancy. This can be passed to *borneo.iam.SignatureProvider.get_resource_principal_claim()* to retrieve the resource's tenancy OCID.

ResourceNotFoundException

exception borneo.**ResourceNotFoundException**(*message*)
The operation attempted to access a resource that does not exist or is not in a visible state.

Result

class borneo.**Result**
Bases: object

Result is a base class for result classes for all supported operations. All state and methods are maintained by extending classes.

Methods Summary

—

Methods Documentation

RetryHandler

class borneo.**RetryHandler**

Bases: object

RetryHandler is called by the request handling system when a *RetryableException* is thrown. It controls the number of retries as well as frequency of retries using a delaying algorithm. A default RetryHandler is always configured on a *NoSQLHandle* instance and can be controlled or overridden using *NoSQLHandleConfig.set_retry_handler()* and *NoSQLHandleConfig.configure_default_retry_handler()*.

It is not recommended that applications rely on a RetryHandler for regulating provisioned throughput. It is best to add rate limiting to the application based on a table's capacity and access patterns to avoid throttling exceptions: see *NoSQLHandleConfig.set_rate_limiting_enabled()*.

Instances of this class must be immutable so they can be shared among threads.

Methods Summary

delay(request, num_retried, re)	This method is called when a *RetryableException* is thrown and it is determined that the request will be retried based on the return value of *do_retry()*.
do_retry(request, num_retried, re)	This method is called when a *RetryableException* is thrown and determines whether to perform a retry or not based on the parameters.
get_num_retries()	Returns the number of retries that this handler instance will allow before the exception is thrown to the application.

Methods Documentation

delay(*request*, *num_retried*, *re*)

This method is called when a *RetryableException* is thrown and it is determined that the request will be retried based on the return value of *do_retry()*. It provides a delay between retries. Most implementations will sleep for some period of time. The method should not return until the desired delay period has passed. Implementations should not busy-wait in a tight loop.

If delayMS is non-zero, use it. Otherwise, use a exponential backoff algorithm to compute the time of delay.

If retry-able exception is SecurityInfoNotReadyException, delay for SEC_RETRY_DELAY_MS when number of retries is smaller than 10. Otherwise, use the exponential backoff algorithm to compute the time of delay.

Parameters

- **request** (Request) – request to execute.
- **num_retried** (*int*) – the number of retries that have occurred for the operation.
- **re** (RetryableException) – the exception that was thrown.

Raises ***IllegalArgumentException*** – raises the exception if num_retried is not a positive number.

do_retry(*request*, *num_retried*, *re*)
This method is called when a *RetryableException* is thrown and determines whether to perform a retry or not based on the parameters.

Default behavior is to *not* retry OperationThrottlingException because the retry time is likely much longer than normal because they are DDL operations. In addition, *not* retry any requests that should not be retired: TableRequest, ListTablesRequest, GetTableRequest, TableUsageRequest, GetIndexesRequest.

Always retry SecurityInfoNotReadyException until exceed the request timeout. It's not restrained by the maximum retries configured for this handler, the driver with retry handler with 0 retry setting would still retry this exception.

Parameters

- **request** (Request) – the request that has triggered the exception.
- **num_retried** (*int*) – the number of retries that have occurred for the operation.
- **re** (RetryableException) – the exception that was thrown.

Returns True if the operation should be retried, False if not, causing the exception to be thrown to the application.

Return type bool

Raises ***IllegalArgumentException*** – raises the exception if num_retried is not a positive number.

get_num_retries()
Returns the number of retries that this handler instance will allow before the exception is thrown to the application.

Returns the max number of retries.

Return type int

RetryableException

exception borneo.**RetryableException**(*message*)
A base class for all exceptions that may be retried with a reasonable expectation that they may succeed on retry.

SecurityInfoNotReadyException

exception borneo.**SecurityInfoNotReadyException**(*message*)
Cloud service only.

An exception that is thrown when security information is not ready in the system. This exception will occur as the system acquires security information and must be retried in order for authorization to work properly.

State

class borneo.**State**
Bases: object

Represents the table state.

Attributes Summary

ACTIVE	Represents the table is active.
CREATING	Represents the table is creating.
DROPPED	Represents the table is dropped.
DROPPING	Represents the table is dropping.
UPDATING	Represents the table is updating.

Attributes Documentation

ACTIVE = 'ACTIVE'
Represents the table is active.

CREATING = 'CREATING'
Represents the table is creating.

DROPPED = 'DROPPED'
Represents the table is dropped.

DROPPING = 'DROPPING'
Represents the table is dropping.

UPDATING = 'UPDATING'
Represents the table is updating.

StatsControl

class borneo.**StatsControl**(*config*, *logger*, *is_rate_limiting_enabled*)
Bases: object

StatsControl allows user to control the collection of driver statistics at runtime.

The statistics data is collected for an interval of time. At the end of the interval, the stats data is logged in a specified JSON format that can be filtered and parsed. After the logging, the counters are cleared and collection of data resumes.

Collection intervals are aligned to the top of the hour. This means first interval logs may contain stats for a shorter interval.

Collection of stats are controlled by the following environment variables:

NOSQL_STATS_PROFILE=[none|regular|more|all]

Specifies the stats profile:

- none - disabled,
- regular - per request: counters, errors, latencies, delays, retries. This incurs minimum overhead.
- more - stats above with 95th and 99th percentile latencies. This may add 0.5% overhead compared to none stats profile.
- all - stats above with per query information. This may add 1% overhead compared to none stats profile.

NOSQL_STATS_INTERVAL=600 Interval in seconds to log the stats, by default is 10 minutes.

NOSQL_STATS_PRETTY_PRINT=true Option to enable pretty printing of the JSON data, default value is false.

Collection of stats can also be used by using the API:

`NoSQLHandleConfig.set_stats_profile()` or *StatsControl.set_profile()*. At runtime stats collection can be enabled selectively by using *StatsControl.start()* ond *StatsControl.stop()*. The following example shows how to use a stats handler and how to control the stas at runtime:

```
def stats_handler(stats):
    # type: (Dict) -> None
    print("Stats : " + str(stats))
...
config = NoSQLHandleConfig( endpoint )
config.set_stats_profile(StatsProfile.REGULAR)
config.set_stats_interval(600)
config.set_stats_pretty_print(False)
config.set_stats_handler(stats_handler)

handle = NoSQLHandle(config)

handle = get_handle(tenant_id)

stats_control = handle.get_stats_control()

#... application code without stats

# enable observations
stats_control.start();

#... application code with REGULAR stats

# For particular parts of code profile can be changed to collect more␣
↪stats.
stats_control.set_stats_profile(StatsProfile.ALL)
#... more sensitive code with ALL stats

stats_control.set_stats_profile(StatsProfile.REGULAR)
#... application code with REGULAR stats

# disable observations
stats_control.stop()

#... application code without stats
handle.close()
```

The following is an example of stats log entry using the ALL profile:

- A one time entry containing stats id and options:

```
INFO: Client stats|{    // INFO log entry
"sdkName" : "Oracle NoSQL SDK for Python",  // SDK name
"sdkVersion" : "5.2.4",                   // SDK version
"clientId" : "f595b333",                    // NoSQLHandle id
"profile" : "ALL",                          // stats profile
"intervalSec" : 600,                        // interval length in␣
↪seconds
"prettyPrint" : true,                       // JSON pretty print
```

(continues on next page)

(continued from previous page)

```
"rateLimitingEnabled" : false}            // if rate limiting is␣
↪enabled
```

- An entry at the end of each interval containing the stats values:

```
``INFO: Client stats|{
"clientId" : "b7bc7734",                // id of NoSQLHandle object
"startTime" : "2021-09-20T20:11:42Z", // UTC start interval time
"endTime" : "2021-09-20T20:11:47Z",    // UTC end interval time
"requests" : [{                         // array of types of requests
  "name" : "Get",                         // stats for GET request type
  "httpRequestCount" : 2,                 // count of http requests
  "errors" : 0,                           // number of errors in␣
↪interval
  "httpRequestLatencyMs" : {              // response time of http␣
↪requests
    "min" : 4,                              // minimum value in␣
↪interval
    "avg" : 4.5,                            // average value in␣
↪interval
    "max" : 5,                              // maximum value in␣
↪interval
    "95th" : 5,                             // 95th percentile value
    "99th" : 5                              // 99th percentile value
  },
  "requestSize" : {                       // http request size in bytes
    "min" : 42,                             // minimum value in␣
↪interval
    "avg" : 42.5,                           // average value in␣
↪interval
    "max" : 43                              // maximum value in␣
↪interval
  },
  "resultSize" : {                        // http result size in bytes
    "min" : 193,                            // minimum value in␣
↪interval
    "avg" : 206.5,                          // average value in␣
↪interval
    "max" : 220                             // maximum value in␣
↪interval
  },
  "rateLimitDelayMs" : 0,                 // delay in milliseconds␣
↪introduced by the rate limiter
  "retry" : {                             // retries
    "delayMs" : 0,                          // delay in milliseconds␣
↪introduced by retries
    "authCount" : 0,                        // no of auth retries
    "throttleCount" : 0,                    // no of throttle retries
    "count" : 0                             // total number of retries
  }
}, {
  "name" : "Query",                     // stats for all QUERY type␣
↪requests
  "httpRequestCount" : 14,
  "errors" : 0,
  "httpRequestLatencyMs" : {
    "min" : 3,
```

(continues on next page)

(continued from previous page)

```
    "avg" : 13.0,
    "max" : 32,
    "95th" : 32,
    "99th" : 32
  },
  "resultSize" : {
    "min" : 146,
    "avg" : 7379.71,
    "max" : 10989
  },
  "requestSize" : {
    "min" : 65,
    "avg" : 709.85,
    "max" : 799
  },
  "rateLimitDelayMs" : 0,
  "retry" : {
    "delayMs" : 0,
    "authCount" : 0,
    "throttleCount" : 0,
    "count" : 0
  }
}, {
  "name" : "Put",                      // stats for PUT type requests
  "httpRequestCount" : 1002,
  "errors" : 0,
  "httpRequestLatencyMs" : {
    "min" : 1,
    "avg" : 4.41,
    "max" : 80,
    "95th" : 8,
    "99th" : 20
  },
  "requestSize" : {
    "min" : 90,
    "avg" : 90.16,
    "max" : 187
  },
  "resultSize" : {
    "min" : 58,
    "avg" : 58.0,
    "max" : 58
  },
  "rateLimitDelayMs" : 0,
  "retry" : {
    "delayMs" : 0,
    "authCount" : 0,
    "throttleCount" : 0,
    "count" : 0
  }
}],
"queries" : [{            // query stats aggregated by query␣
↪statement
                          // query statement
  "query" : "SELECT * FROM audienceData ORDER BY cookie_id",
                          // query plan description
  "plan" : "SFW([6])
```

(continues on next page)

(continued from previous page)

```
      [
        FROM:
          RECV([3])
            [
              DistributionKind : ALL_PARTITIONS,
              Sort Fields : sort_gen,
            ] as $from-0
        SELECT:
         FIELD_STEP([6])
           [
            VAR_REF($from-0)([3]),
            audienceData
           ]
      ]",
   "doesWrites" : false,
   "httpRequestCount" : 12,  // number of http calls to the server
   "unprepared" : 1,         // number of query requests without␣
↪prepare
   "simple" : false,         // type of query
   "count" : 20,             // number of handle.query() API calls
   "errors" : 0,             // number of calls trowing exception
   "httpRequestLatencyMs" : {// response time of http requests in␣
↪milliseconds
     "min" : 8,                // minimum value in interval
     "avg" : 14.58,            // average value in interval
     "max" : 32,               // maximum value in interval
     "95th" : 32,              // 95th percentile value in interval
     "99th" : 32               // 99th percentile value in interval
   },
   "requestSize" : {         // http request size in bytes
     "min" : 65,               // minimum value in interval
     "avg" : 732.5,            // average value in interval
     "max" : 799               // maximum value in interval
   },
   "resultSize" : {          // http result size in bytes
     "min" : 914,              // minimum value in interval
     "avg" : 8585.33,          // average value in interval
     "max" : 10989             // maximum value in interval
   },
   "rateLimitDelayMs" : 0,   // total delay introduced by rate␣
↪limiter in milliseconds
   "retry" : {               // automatic retries
     "delayMs" : 0,            // delay introduced by retries
     "authCount" : 0,          // count of auth related retries
     "throttleCount" : 0,      // count of throttle related retries
     "count" : 0               // total count of retries
   }
 }]
}``
```

The log entries go to the logger configured in NoSQLHandlerConfig. By default, if no logger is configured the statistics entries, if enabled, will be logged to file **logs/driver.log** in the local directory.

Stats collection is not dependent of logging configuration, even if logging is disabled, collection of stats will still happen if stats profile other than *none* is used. In this case, the stats are available by using the stats handler.

Depending on the type of query, if client processing is required, for example in the case of ordered or aggregate queries, indicated by the false **simple** field of the **query** entry, the **count** and **httpRequestsCount** numbers

will differ. **count** represents the number of `handle.query()` API calls and **httpRequestCount** represents the number of internal http requests from server. For these type of queries, the driver executes several simpler queries, per shard or partition, and than combines the results locally.

Note: connection statistics are not available for NoSQL Python driver.

Attributes Summary

LOG_PREFIX

Methods Summary

get_id()	Returns a pseudo unique string to identify the NoSQLHandle object.
get_interval()	Returns the current collection interval.
get_logger()	Returns the current logger.
get_pretty_print()	Returns the current JSON pretty print flag.
get_profile()	Returns the stats collection profile.
get_stats_handler()	Returns the registered handler.
is_started()	Returns true if collection of stats is enabled, otherwise returns false.
observe(request, req_size, res_size, ...)	Internal method only.
observe_error(request)	Internal method only.
observe_query(query_request)	Internal method only.
set_pretty_print(pretty_print)	Enable JSON pretty print for easier human reading.
set_profile(profile)	Set the stats collection stats_profile.
set_stats_handler(stats_handler)	Registers a user defined stats handler.
shutdown()	Logs the stats collected and stops the timer.
start()	Collection of stats is enabled only between start and stop or from the beginning if environment property NOSQL_STATS_PROFILE is not "none".
stop()	Stops collection of stats.

Attributes Documentation

LOG_PREFIX = 'Client stats|'

Methods Documentation

get_id()
Returns a pseudo unique string to identify the NoSQLHandle object.

get_interval()
Returns the current collection interval. Default interval is 600 seconds, i.e. 10 min.

get_logger()
Returns the current logger.

get_pretty_print()
Returns the current JSON pretty print flag. Default is disabled.

get_profile()
Returns the stats collection profile. Default stats profile is NONE.

get_stats_handler()
Returns the registered handler.

is_started()
Returns true if collection of stats is enabled, otherwise returns false.

observe(*request*, *req_size*, *res_size*, *network_latency*)
Internal method only.

observe_error(*request*)
Internal method only.

observe_query(*query_request*)
Internal method only.

set_pretty_print(*pretty_print*)
Enable JSON pretty print for easier human reading. Default is disabled.

set_profile(*profile*)
Set the stats collection stats_profile. Default stats stats_profile is NONE.

set_stats_handler(*stats_handler*)
Registers a user defined stats handler. The handler is called at the end of the interval with a structure containing the logged stat values.

shutdown()
Logs the stats collected and stops the timer.

start()
Collection of stats is enabled only between start and stop or from the beginning if environment property NOSQL_STATS_PROFILE is not "none".

stop()
Stops collection of stats.

StatsProfile

class borneo.**StatsProfile**
Bases: enum.Enum

The following semantics are attached to the StatsProfile values:

- NONE: no stats are logged.
- REGULAR: per request: counters, errors, latencies, delays, retries
- MORE: stats above plus 95th and 99th percentile latencies.
- ALL: stats above plus per query information

Attributes Summary

ALL
MORE
NONE
REGULAR

Attributes Documentation

ALL = 4

MORE = 3

NONE = 1

REGULAR = 2

SystemException

exception borneo.**SystemException**(*message*)

An exception that is thrown when there is an internal system problem. Most system problems are temporary, so this is a retryable exception.

SystemRequest

class borneo.**SystemRequest**

Bases: borneo.operations.Request

On-premise only.

SystemRequest is an on-premise-only request used to perform any table-independent administrative operation such as create/drop of namespaces and security-relevant operations (create/drop users and roles). These operations are asynchronous and completion needs to be checked.

Examples of statements used in this object include:

CREATE NAMESPACE mynamespace

CREATE USER some_user IDENTIFIED BY password

CREATE ROLE some_role

GRANT ROLE some_role TO USER some_user

Execution of operations specified by this request is implicitly asynchronous. These are potentially long-running operations. *NoSQLHandle.system_request()* returns a *SystemResult* instance that can be used to poll until the operation succeeds or fails.

Methods Summary

get_statement()	Returns the statement, or None if not set.
get_timeout()	Returns the timeout to use for the operation, in milliseconds.
set_statement(statement)	Sets the statement to use for the operation.
set_timeout(timeout_ms)	Sets the request timeout value, in milliseconds.

Methods Documentation

get_statement()

Returns the statement, or None if not set.

Returns the statement.

Return type str

get_timeout()
Returns the timeout to use for the operation, in milliseconds. A value of 0 indicates that the timeout has not been set.

Returns the timeout value.

Return type int

set_statement(*statement*)
Sets the statement to use for the operation.

Parameters **statement** (*str*) – the statement. This is a required parameter.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if statement is not a string.

set_timeout(*timeout_ms*)
Sets the request timeout value, in milliseconds. This overrides any default value set in *NoSQLHandleConfig*. The value must be positive.

Parameters **timeout_ms** (*int*) – the timeout value, in milliseconds.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if the timeout value is less than or equal to 0.

SystemResult

class borneo.**SystemResult**
Bases: borneo.operations.Result

On-premise only.

SystemResult is returned from *NoSQLHandle.system_status()* and *NoSQLHandle.system_request()* operations. It encapsulates the state of the operation requested.

Some operations performed by *NoSQLHandle.system_request()* are asynchronous. When such an operation has been performed it is necessary to call *NoSQLHandle.system_status()* until the status of the operation is known. The method *wait_for_completion()* exists to perform this task and should be used whenever possible.

Asynchronous operations (e.g. create namespace) can be distinguished from synchronous system operations in this way:

Asynchronous operations may return a non-none operation id.

Asynchronous operations modify state, while synchronous operations are read-only.

Synchronous operations return a state of STATE.COMPLETE and have a non-none result string.

NoSQLHandle.system_status() is synchronous, returning the known state of the operation. It should only be called if the operation was asynchronous and returned a non-none operation id.

Methods Summary

get_operation_id()	Returns the operation id for the operation if it was asynchronous.
get_operation_state()	Returns the operation state.

Continued on next page

Table 42 – continued from previous page

get_result_string()	Returns the result string for the operation.
get_statement()	Returns the statement used for the operation.
wait_for_completion(handle, wait_millis, ...)	Waits for the operation to be complete.

Methods Documentation

get_operation_id()
Returns the operation id for the operation if it was asynchronous. This is None if the request did not generate a new operation and/or the operation state is SystemState.COMPLETE. The value can be used in *SystemStatusRequest.set_operation_id()* to get status and find potential errors resulting from the operation.

This method is only useful for the result of asynchronous operations.

Returns the operation id.

Return type str

get_operation_state()
Returns the operation state.

Returns the state.

Return type int

get_result_string()
Returns the result string for the operation. This is None if the request was asynchronous or did not return an actual result. For example the "show" operations return a non-none result string, but "create, drop, grant, etc." operations return a none result string.

Returns the result string.

Return type str

get_statement()
Returns the statement used for the operation.

Returns the statement.

Return type str

wait_for_completion(*handle*, *wait_millis*, *delay_millis*)
Waits for the operation to be complete. This is a blocking, polling style wait that delays for the specified number of milliseconds between each polling operation.

This instance is modified with any changes in state.

Parameters

- **handle** (NoSQLHandle) – the NoSQLHandle to use. This is required.
- **wait_millis** (*int*) – the total amount of time to wait, in milliseconds. This value must be non-zero and greater than delay_millis. This is required.
- **delay_millis** (*int*) – the amount of time to wait between polling attempts, in milliseconds. If 0 it will default to 500. This is required.

Raises ***IllegalArgumentException*** – raises the exception if the operation times out or the parameters are not valid.

SystemState

class borneo.**SystemState**
Bases: object

On-premise only.

The current state of the system request.

Attributes Summary

COMPLETE	The operation is complete and was successful.
WORKING	The operation is in progress.

Attributes Documentation

COMPLETE = 'COMPLETE'
The operation is complete and was successful. Failures are thrown as exceptions.

WORKING = 'WORKING'
The operation is in progress.

SystemStatusRequest

class borneo.**SystemStatusRequest**
Bases: borneo.operations.Request

On-premise only.

SystemStatusRequest is an on-premise-only request used to check the status of an operation started using a *SystemRequest*.

Methods Summary

get_operation_id()	Returns the operation id to use for the request, None if not set.
get_statement()	Returns the statement set by *set_statement()*, or None if not set.
get_timeout()	Returns the timeout to use for the operation, in milliseconds.
set_operation_id(operation_id)	Sets the operation id to use for the request.
set_statement(statement)	Sets the statement that was used for the operation.
set_timeout(timeout_ms)	Sets the request timeout value, in milliseconds.

Methods Documentation

get_operation_id()
Returns the operation id to use for the request, None if not set.

Returns the operation id.

Return type str

get_statement()
Returns the statement set by *set_statement()*, or None if not set.

Returns the statement.

Return type str

get_timeout()
Returns the timeout to use for the operation, in milliseconds. A value of 0 indicates that the timeout has not been set.

Returns the timeout value.

Return type int

set_operation_id(*operation_id*)
Sets the operation id to use for the request. The operation id can be obtained via *SystemResult.get_operation_id()*. This parameter is not optional and represents an asynchronous operation that may be in progress. It is used to examine the result of the operation and if the operation has failed an exception will be thrown in response to a *NoSQLHandle.system_status()* operation. If the operation is in progress or has completed successfully, the state of the operation is returned.

Parameters **operation_id** (*str*) – the operation id.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if operation_id is a negative number.

set_statement(*statement*)
Sets the statement that was used for the operation. This is optional and is not used in any significant way. It is returned, unmodified, in the *SystemResult* for convenience.

Parameters **statement** (*str*) – the statement. This is a optional parameter.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if statement is not a string.

set_timeout(*timeout_ms*)
Sets the request timeout value, in milliseconds. This overrides any default value set in *NoSQLHandleConfig*. The value must be positive.

Parameters **timeout_ms** (*int*) – the timeout value, in milliseconds.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if the timeout value is less than or equal to 0.

TableExistsException

exception borneo.**TableExistsException**(*message*)
The operation attempted to create a table but the named table already exists.

TableLimits

class borneo.**TableLimits**(*read_units*, *write_units*, *storage_gb*, *mode=1*)
Bases: object

Cloud service only.

A TableLimits instance is used during table creation to specify the throughput and capacity to be consumed by the table. It is also used in an operation to change the limits of an existing table. *NoSQLHandle.table_request()* and *TableRequest* are used to perform these operations. These values are enforced by the system and used for billing purposes.

Throughput limits are defined in terms of read units and write units. A read unit represents 1 eventually consistent read per second for data up to 1 KB in size. A read that is absolutely consistent is double that, consuming 2 read units for a read of up to 1 KB in size. This means that if an application is to use Consistency.ABSOLUTE it may need to specify additional read units when creating a table. A write unit represents 1 write per second of data up to 1 KB in size.

In addition to throughput table capacity must be specified to indicate the maximum amount of storage, in gigabytes, allowed for the table.

In provisioned mode, all 3 values must be used whenever using this object. There are no defaults and no mechanism to indicate "no change."

In on demand mode, only the storage_gb parameter must be set.

Parameters

- **read_units** (*int*) – the desired throughput of read operation in terms of read units. A read unit represents 1 eventually consistent read per second for data up to 1 KB in size. A read that is absolutely consistent is double that, consuming 2 read units for a read of up to 1 KB in size.
- **write_units** (*int*) – the desired throughput of write operation in terms of write units. A write unit represents 1 write per second of data up to 1 KB in size.
- **storage_gb** (*int*) – the maximum storage to be consumed by the table, in gigabytes.
- **mode** (*CAPACITY_MODE*) – the mode of the table: provisioned (the default) or on demand.

Raises ***IllegalArgumentException*** – raises the exception if parameters are not valid.

Versionchanged 5.3.0, added optional CAPACITY_MODE

Methods Summary

get_mode()	Returns the capacity mode of the table.
get_read_units()	Returns the read throughput in terms of read units.
get_storage_gb()	Returns the storage capacity in gigabytes.
get_write_units()	Returns the write throughput in terms of write units.
set_mode(mode)	Sets the mode of the table:
set_read_units(read_units)	Sets the read throughput in terms of read units.
set_storage_gb(storage_gb)	Sets the storage capacity in gigabytes.
set_write_units(write_units)	Sets the write throughput in terms of write units.

Methods Documentation

__init__(*read_units*, *write_units*, *storage_gb*, *mode=1*)

Creates a TableLimits object

Parameters

- **read_units** (*int*) – the desired throughput of read operation in terms of read units. A read unit represents 1 eventually consistent read per second for data up to 1 KB in size. A read that is absolutely consistent is double that, consuming 2 read units for a read of up to

1 KB in size.

- **write_units** (*int*) – the desired throughput of write operation in terms of write units. A write unit represents 1 write per second of data up to 1 KB in size.
- **storage_gb** (*int*) – the maximum storage to be consumed by the table, in gigabytes.
- **mode** (*CAPACITY_MODE*) – the mode of the table: provisioned (the default) or on demand.

Raises ***IllegalArgumentException*** – raises the exception if parameters are not valid.

Versionchanged 5.3.0, added optional CAPACITY_MODE

get_mode()
Returns the capacity mode of the table.

Returns mode: PROVISIONED or ON_DEMAND

Versionadded 5.3.0

get_read_units()
Returns the read throughput in terms of read units.

Returns the read units.

Return type int

get_storage_gb()
Returns the storage capacity in gigabytes.

Returns the storage capacity in gigabytes.

Return type int

get_write_units()
Returns the write throughput in terms of write units.

Returns the write units.

Return type int

set_mode(*mode*)

Sets the mode of the table: PROVISIONED: Fixed maximum read/write units. This is the default. ON_DEMAND: Flexible read/write limits.

Parameters **mode** (*TableLimits.CAPACITY_MODE*) – the capacity to use, in gigabytes.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if mode is invalid.

Versionadded 5.3.0

set_read_units(*read_units*)
Sets the read throughput in terms of read units.

Parameters **read_units** (*int*) – the throughput to use, in read units.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if read_units is not a integer.

set_storage_gb(*storage_gb*)
Sets the storage capacity in gigabytes.

Parameters **storage_gb** (*int*) – the capacity to use, in gigabytes.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if storage_gb is not a integer.

set_write_units(*write_units*)
Sets the write throughput in terms of write units.

Parameters **write_units** (*int*) – the throughput to use, in write units.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if write_units is not a integer.

TableNotFoundException

exception borneo.**TableNotFoundException**(*message*)
The operation attempted to access a table that does not exist or is not in a visible state.

TableRequest

class borneo.**TableRequest**
Bases: borneo.operations.Request

TableRequest is used to create, modify, and drop tables. The operations allowed are those supported by the Data Definition Language (DDL) portion of the query language. The language provides for table creation and removal (drop), index add and drop, as well as schema evolution via alter table. Operations using DDL statements infer the table name from the query statement itself, e.g. "create table mytable(...)". Table creation requires a valid *TableLimits* object to define the throughput desired for the table. If TableLimits is provided with any other type of query statement an exception is thrown.

This request is also used to modify the limits of throughput and storage for an existing table. This case is handled by specifying a table name and limits without a query statement. If all three are specified it is an error.

Execution of operations specified by this request is implicitly asynchronous. These are potentially long-running operations. *NoSQLHandle.table_request()* returns a *TableResult* instance that can be used to poll until the table reaches the desired state.

The statement is required parameter.

Methods Summary

get_compartment()	Cloud service only.
get_statement()	Returns the statement, or None if not set.
get_table_limits()	Returns the table limits, or None if not set.
get_table_name()	Returns the table name to use for the operation.
get_timeout()	Returns the timeout to use for the operation, in milliseconds.
set_compartment(compartment)	Cloud service only.
set_statement(statement)	Sets the query statement to use for the operation.
set_table_limits(table_limits)	Cloud service only.
set_table_name(table_name)	Sets the table name to use for the operation.
set_timeout(timeout_ms)	Sets the request timeout value, in milliseconds.

Methods Documentation

get_compartment()
Cloud service only.

Get the compartment id or name if set for the request.

Returns compartment id or name if set for the request, otherwise None if not set.

Return type str

get_statement()
Returns the statement, or None if not set.

Returns the statement.

Return type str

get_table_limits()
Returns the table limits, or None if not set.

Returns the limits.

Return type *TableLimits*

get_table_name()
Returns the table name to use for the operation.

Returns the table name, or None if not set.

Returns str

get_timeout()
Returns the timeout to use for the operation, in milliseconds. A value of 0 indicates that the timeout has not been set.

Returns the timeout value.

Return type int

set_compartment(*compartment*)
Cloud service only.

Sets the name or id of a compartment to be used for this operation.

The compartment may be specified as either a name (or path for nested compartments) or as an id (OCID). A name (vs id) can only be used when authenticated using a specific user identity. It is *not* available if authenticated as an Instance Principal which can be done when calling the service from a compute instance in the Oracle Cloud Infrastructure. See *borneo.iam.SignatureProvider.create_with_instance_principal()*.

Parameters **compartment** (*str*) – the compartment name or id. If using a nested compartment, specify the full compartment path compartmentA.compartmentB, but exclude the name of the root compartment (tenant).

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if compartment is not a str.

set_statement(*statement*)
Sets the query statement to use for the operation. This parameter is required unless the operation is intended to change the limits of an existing table.

Parameters **statement** (*str*) – the statement.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if statement is not a string.

set_table_limits(*table_limits*)
Cloud service only.

Sets the table limits to use for the operation. Limits are used in only 2 cases – table creation statements and limits modification operations. It is not used for other DDL operations.

If limits are set for an on-premise service they are silently ignored.

Parameters **table_limits** (TableLimits) – the limits.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if table_limits is not an instance TableLimits.

set_table_name(*table_name*)
Sets the table name to use for the operation. The table name is only used to modify the limits of an existing table, and must not be set for any other operation.

Parameters **table_name** (*str*) – the name of the table.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if table_name is not a string.

set_timeout(*timeout_ms*)
Sets the request timeout value, in milliseconds. This overrides any default value set in *NoSQLHandleConfig*. The value must be positive.

Parameters **timeout_ms** (*int*) – the timeout value, in milliseconds.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if the timeout value is less than or equal to 0.

TableResult

class borneo.**TableResult**
Bases: borneo.operations.Result

TableResult is returned from *NoSQLHandle.get_table()* and *NoSQLHandle.table_request()* operations. It encapsulates the state of the table that is the target of the request.

Operations available in *NoSQLHandle.table_request()* such as table creation, modification, and drop are asynchronous operations. When such an operation has been performed, it is necessary to call *NoSQLHandle.get_table()* until the status of the table is State.ACTIVE, State.DROPPED or there is an error condition. The method *wait_for_completion()* exists to perform this task and should be used to wait for an operation to complete.

NoSQLHandle.get_table() is synchronous, returning static information about the table as well as its current state.

Methods Summary

get_operation_id()	Returns the operation id for an asynchronous operation.

Continued on next page

Table 47 – continued from previous page

get_schema()	Returns the schema for the table.
get_state()	Returns the table state.
get_table_limits()	Returns the throughput and capacity limits for the table.
get_table_name()	Returns the table name of the target table.
wait_for_completion(handle, wait_millis, ...)	Waits for a table operation to complete.

Methods Documentation

get_operation_id()
Returns the operation id for an asynchronous operation. This is none if the request did not generate a new operation. The value can be used in `set_operation_id()` to find potential errors resulting from the operation.

Returns the operation id for an asynchronous operation.

Return type str

get_schema()
Returns the schema for the table.

Returns the schema for the table.

Return type str

get_state()
Returns the table state. A table in state State.ACTIVE or State.UPDATING is usable for normal operation.

Returns the state.

Return type *State*

get_table_limits()
Returns the throughput and capacity limits for the table. Limits from an on-premise service will always be None.

Returns the limits.

Return type *TableLimits*

get_table_name()
Returns the table name of the target table.

Returns the table name.

Return type str

wait_for_completion(*handle*, *wait_millis*, *delay_millis*)
Waits for a table operation to complete. Table operations are asynchronous. This is a blocking, polling style wait that delays for the specified number of milliseconds between each polling operation. This call returns when the table reaches a *terminal* state, which is either State.ACTIVE or State.DROPPED.

This instance must be the return value of a previous *NoSQLHandle.table_request()* and contain a non-none operation id representing the in-progress operation unless the operation has already completed.

This instance is modified with any change in table state or metadata.

Parameters

- **handle** (NoSQLHandle) – the NoSQLHandle to use.

- **wait_millis** (*int*) – the total amount of time to wait, in milliseconds. This value must be non-zero and greater than delay_millis.
- **delay_millis** (*int*) – the amount of time to wait between polling attempts, in milliseconds. If 0 it will default to 500.

Raises

- ***IllegalArgumentException*** – raises the exception if the parameters are not valid.
- ***RequestTimeoutException*** – raises the exception if the operation times out.

TableUsageRequest

class borneo.**TableUsageRequest**

Bases: borneo.operations.Request

Cloud service only.

Represents the argument of a *NoSQLHandle.get_table_usage()* operation which returns dynamic information associated with a table, as returned in *TableUsageResult*. This information includes a time series of usage snapshots, each indicating data such as read and write throughput, throttling events, etc, as found in TableUsageResult.table_usage().

It is possible to return a range of usage records or, by default, only the most recent usage record. Usage records are created on a regular basis and maintained for a period of time. Only records for time periods that have completed are returned so that a user never sees changing data for a specific range.

The table name is required parameter.

Methods Summary

get_compartment()	Cloud service only.
get_end_time()	Returns the end time to use for the request in milliseconds since the Epoch.
get_end_time_string()	Returns the end time as an ISO 8601 formatted string.
get_limit()	Returns the limit to the number of usage records desired.
get_start_time()	Returns the start time to use for the request in milliseconds since the Epoch.
get_start_time_string()	Returns the start time as an ISO 8601 formatted string.
get_table_name()	Returns the table name to use for the operation.
get_timeout()	Returns the timeout to use for the operation, in milliseconds.
set_compartment(compartment)	Cloud service only.
set_end_time(end_time)	Sets the end time to use for the request in milliseconds since the Epoch in UTC time or an ISO 8601 formatted string accurate to milliseconds.
set_limit(limit)	Sets the limit to the number of usage records desired.
set_start_time(start_time)	Sets the start time to use for the request in milliseconds since the Epoch in UTC time or an ISO 8601 formatted string accurate to milliseconds.

Continued on next page

Table 48 – continued from previous page

set_table_name(table_name)	Sets the table name to use for the request.
set_timeout(timeout_ms)	Sets the request timeout value, in milliseconds.

Methods Documentation

get_compartment()
Cloud service only.

Get the compartment id or name if set for the request.

Returns compartment id or name if set for the request, otherwise None if not set.

Return type str

get_end_time()
Returns the end time to use for the request in milliseconds since the Epoch.

Returns the end time.

Return type int

get_end_time_string()
Returns the end time as an ISO 8601 formatted string. If the end timestamp is not set, None is returned.

Returns the end time, or None if not set.

Return type str

get_limit()
Returns the limit to the number of usage records desired.

Returns the numeric limit.

Return type int

get_start_time()
Returns the start time to use for the request in milliseconds since the Epoch.

Returns the start time.

Return type int

get_start_time_string()
Returns the start time as an ISO 8601 formatted string. If the start timestamp is not set, None is returned.

Returns the start time, or None if not set.

Return type str

get_table_name()
Returns the table name to use for the operation.

Returns the table name, or None if not set.

Returns str

get_timeout()
Returns the timeout to use for the operation, in milliseconds. A value of 0 indicates that the timeout has not been set.

Returns the value.

Return type int

set_compartment(*compartment*)
Cloud service only.

Sets the name or id of a compartment to be used for this operation.

The compartment may be specified as either a name (or path for nested compartments) or as an id (OCID). A name (vs id) can only be used when authenticated using a specific user identity. It is *not* available if authenticated as an Instance Principal which can be done when calling the service from a compute instance in the Oracle Cloud Infrastructure. See *borneo.iam.SignatureProvider.create_with_instance_principal()*.

Parameters **compartment** (*str*) – the compartment name or id. If using a nested compartment, specify the full compartment path compartmentA.compartmentB, but exclude the name of the root compartment (tenant).

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if compartment is not a str.

set_end_time(*end_time*)
Sets the end time to use for the request in milliseconds since the Epoch in UTC time or an ISO 8601 formatted string accurate to milliseconds. If timezone is not specified it is interpreted as UTC.

If no time range is set for this request the most recent complete usage record is returned.

Parameters **end_time** (*str*) – the end time.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if end_time is a negative number and is not an ISO 8601 formatted string.

set_limit(*limit*)
Sets the limit to the number of usage records desired. If this value is 0 there is no limit, but not all usage records may be returned in a single request due to size limitations.

Parameters **limit** (*int*) – the numeric limit.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if limit is a negative number.

set_start_time(*start_time*)
Sets the start time to use for the request in milliseconds since the Epoch in UTC time or an ISO 8601 formatted string accurate to milliseconds. If timezone is not specified it is interpreted as UTC.

If no time range is set for this request the most recent complete usage record is returned.

Parameters **start_time** (*str*) – the start time.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if start_time is a negative number and is not an ISO 8601 formatted string.

set_table_name(*table_name*)
Sets the table name to use for the request. This is a required parameter.

Parameters **table_name** (*str*) – the table name.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if table_name is not a string.

set_timeout(*timeout_ms*)
Sets the request timeout value, in milliseconds. This overrides any default value set in *NoSQLHandleConfig*. The value must be positive.

Parameters **timeout_ms** (*int*) – the timeout value, in milliseconds.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if the timeout value is less than or equal to 0.

TableUsageResult

class borneo.**TableUsageResult**
Bases: borneo.operations.Result

Cloud service only.

TableUsageResult is returned from *NoSQLHandle.get_table_usage()*. It encapsulates the dynamic state of the requested table.

Methods Summary

get_table_name()	Returns the table name used by the operation.
get_usage_records()	Returns a list of usage records based on the parameters of the *TableUsageRequest* used.

Methods Documentation

get_table_name()
Returns the table name used by the operation.

Returns the table name.

Return type str

get_usage_records()
Returns a list of usage records based on the parameters of the *TableUsageRequest* used.

Returns an list of usage records.

Type list(TableUsage)

ThrottlingException

exception borneo.**ThrottlingException**(*message*)
Cloud service only.

ThrottlingException is a base class for exceptions that indicate the application has exceeded a provisioned or implicit limit in terms of size of data accessed or frequency of operation.

Operations resulting in this exception can be retried but it is recommended that callers use a delay before retrying in order to minimize the chance that a retry will also be throttled.

It is recommended that applications use rate limiting to avoid these exceptions.

TimeToLive

class borneo.**TimeToLive**(*value*, *timeunit*)
Bases: object

TimeToLive is a utility class that represents a period of time, similar to java.time.Duration in Java, but specialized to the needs of this driver.

This class is restricted to durations of days and hours. It is only used as input related to time to live (TTL) for row instances.

Construction allows only day and hour durations for efficiency reasons. Durations of days are recommended as they result in the least amount of storage overhead. Only positive durations are allowed on input.

Parameters

- **value** (*int*) – value of time.
- **timeunit** (TimeUnit) – unit of time, cannot be None.

Raises ***IllegalArgumentException*** – raises the exception if parameters are not expected type.

Methods Summary

get_unit()	Returns the time unit used for the duration.
get_value()	Returns the numeric duration value.
of_days(days)	Creates a duration using a period of 24 hour days.
of_hours(hours)	Creates a duration using a period of hours.
to_days()	Returns the number of days in this duration, which may be negative.
to_expiration_time(reference_time)	Returns an absolute time representing the duration plus the absolute time reference parameter.
to_hours()	Returns the number of hours in this duration, which may be negative.

Methods Documentation

get_unit()
Returns the time unit used for the duration.

Returns the timeunit.

Return type *TimeUnit*

get_value()
Returns the numeric duration value.

Returns the duration value, independent of unit.

Return type int

static of_days(*days*)
Creates a duration using a period of 24 hour days.

Parameters **days** (*int*) – the number of days in the duration, must be a non-negative number.

Returns the duration.

Return type *TimeToLive*

Raises ***IllegalArgumentException*** – raises the exception if a negative value is provided.

static of_hours(*hours*)
Creates a duration using a period of hours.

Parameters **hours** (*int*) – the number of hours in the duration, must be a non-negative number.

Returns the duration.

Return type *TimeToLive*

Raises ***IllegalArgumentException*** – raises the exception if a negative value is provided.

to_days()
Returns the number of days in this duration, which may be negative.

Returns the number of days.

Return type int

to_expiration_time(*reference_time*)
Returns an absolute time representing the duration plus the absolute time reference parameter. If an expiration time from the current time is desired the parameter should be the current system time in millisecond. If the duration of this object is 0, indicating no expiration time, this method will return 0, regardless of the reference time.

Parameters **reference_time** (*int*) – an absolute time in milliseconds since January 1, 1970.

Returns time in milliseconds, 0 if this object's duration is 0.

Return type int

Raises ***IllegalArgumentException*** – raises the exception if reference_time is not positive.

to_hours()
Returns the number of hours in this duration, which may be negative.

Returns the number of hours.

Return type int

TimeUnit

class borneo.**TimeUnit**
Bases: object

The time unit to use.

Attributes Summary

DAYS	Set TimeUnit.DAYS to use day as time unit
HOURS	Set TimeUnit.HOURS to use hour as time unit

Attributes Documentation

DAYS = 2
Set TimeUnit.DAYS to use day as time unit

HOURS = 1
Set TimeUnit.HOURS to use hour as time unit

UserInfo

class borneo.**UserInfo**(*user_id*, *user_name*)
Bases: object

On-premise only.

A class that encapsulates the information associated with a user including the user id and name in the system.

Methods Summary

get_id()	Returns the id associated with the user.
get_name()	Returns the name associated with the user.

Methods Documentation

get_id()
Returns the id associated with the user.

Returns the user id string.

Return type str

get_name()
Returns the name associated with the user.

Returns the user name string.

Return type str

Version

class borneo.**Version**(*version*)
Bases: object

Version is an opaque class that represents the version of a row in the database. It is returned by successful *GetRequest* and can be used in *PutRequest.set_match_version()* and *DeleteRequest.set_match_version()* to conditionally perform those operations to ensure an atomic read-modify-write cycle. This is an opaque object from an application perspective.

Use of Version in this way adds cost to operations so it should be done only if necessary.

Parameters **version** (*bytearray*) – a bytearray.

Raises ***IllegalArgumentException*** – raises the exception if version is not a bytearray.

Methods Summary

create_version(version)	Returns an instance of *Version*.
get_bytes()	Returns the bytearray from the Version.

Methods Documentation

static create_version(*version*)
Returns an instance of *Version*.

Parameters **version** (*bytearray*) – a bytearray or None.

Returns an instance of Version.

Return type *Version*

Raises ***IllegalArgumentException*** – raises the exception if version is not a bytearray or None.

get_bytes()
Returns the bytearray from the Version.

Returns the bytearray from the Version.

Return type bytearray

WriteMultipleRequest

class borneo.**WriteMultipleRequest**
Bases: borneo.operations.Request

Represents the input to a *NoSQLHandle.write_multiple()* operation.

This request can be used to perform a sequence of *PutRequest* or *DeleteRequest* operations associated with a table that share the same shard key portion of their primary keys, the WriteMultiple operation as whole is atomic. It is an efficient way to atomically modify multiple related rows.

On a successful operation *WriteMultipleResult.get_success()* returns True. The execution result of each operations can be retrieved using *WriteMultipleResult.get_results()*.

If the WriteMultiple operation is aborted because of the failure of an operation with abort_if_unsuccessful set to True, then *WriteMultipleResult.get_success()* return False, the index of failed operation can be accessed using *WriteMultipleResult.get_failed_operation_index()*, and the execution result of failed operation can be accessed using *WriteMultipleResult.get_failed_operation_result()*.

Methods Summary

add(request, abort_if_unsuccessful)	Adds a Request to the operation list, do validation check before adding it.
clear()	Removes all of the operations from the WriteMultiple request.
get_compartment()	Cloud service only.
get_durability()	On-premise only.
get_num_operations()	Returns the number of Requests.
get_request(index)	Returns the Request at the given position, it may be either a *PutRequest* or *DeleteRequest* object.
get_table_name()	Returns the table name to use for the operation.
get_timeout()	Returns the timeout to use for the operation, in milliseconds.

Continued on next page

Table 54 – continued from previous page

set_compartment(compartment)	Cloud service only.
set_durability(durability)	On-premise only.
set_timeout(timeout_ms)	Sets the request timeout value, in milliseconds.

Methods Documentation

add(*request*, *abort_if_unsuccessful*)
Adds a Request to the operation list, do validation check before adding it.

Parameters

- **request** (*Request*) – the Request to add, either *PutRequest* or *DeleteRequest*.
- **abort_if_unsuccessful** (*bool*) – True if this operation should cause the entire WriteMultiple operation to abort when this operation fails.

Returns self.

Raises

- ***BatchOperationNumberLimitException*** – raises the exception if the number of requests exceeds the limit, or IllegalArgumentException if the request is neither a *PutRequest* or *DeleteRequest*. Or any invalid state of the Request.
- ***IllegalArgumentException*** – raises the exception if parameters are not expected type.

clear()
Removes all of the operations from the WriteMultiple request.

get_compartment()
Cloud service only.

Get the compartment id or name if set for the request.

Returns compartment id or name if set for the request, otherwise None if not set.

Return type str

get_durability()
On-premise only. Gets the durability to use for the operation or None if not set :returns: the Durability :versionadded: 5.3.0

get_num_operations()
Returns the number of Requests.

Returns the number of Requests.

Return type int

get_request(*index*)
Returns the Request at the given position, it may be either a *PutRequest* or *DeleteRequest* object.

Parameters **index** (*int*) – the position of Request to get.

Returns the Request at the given position.

Return type *Request*

Raises

- **IndexOutOfBoundsException** – raises the exception if the position is negative or greater or equal to the number of Requests.

- ***IllegalArgumentException*** – raises the exception if index is a negative number.

get_table_name()
Returns the table name to use for the operation.

Returns the table name, or None if not set.

Returns str

get_timeout()
Returns the timeout to use for the operation, in milliseconds. A value of 0 indicates that the timeout has not been set.

Returns the timeout value.

Return type int

set_compartment(*compartment*)
Cloud service only.

Sets the name or id of a compartment to be used for this operation.

The compartment may be specified as either a name (or path for nested compartments) or as an id (OCID). A name (vs id) can only be used when authenticated using a specific user identity. It is *not* available if authenticated as an Instance Principal which can be done when calling the service from a compute instance in the Oracle Cloud Infrastructure. See *borneo.iam.SignatureProvider.create_with_instance_principal()*.

Parameters **compartment** (*str*) – the compartment name or id. If using a nested compartment, specify the full compartment path compartmentA.compartmentB, but exclude the name of the root compartment (tenant).

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if compartment is not a str.

set_durability(*durability*)
On-premise only. Sets the durability to use for the operation.

Parameters **durability** (Durability) – the Durability to use

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if Durability is not valid

Versionadded 5.3.0

set_timeout(*timeout_ms*)
Sets the request timeout value, in milliseconds. This overrides any default value set in *NoSQLHandleConfig*. The value must be positive.

Parameters **timeout_ms** (*int*) – the timeout value, in milliseconds.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if the timeout value is less than or equal to 0.

WriteMultipleResult

class borneo.**WriteMultipleResult**
Bases: borneo.operations.Result

Represents the result of a *NoSQLHandle.write_multiple()* operation.

If the WriteMultiple succeeds, the execution result of each sub operation can be retrieved using *get_results()*.

If the WriteMultiple operation is aborted because of the failure of an operation with abort_if_unsuccessful set to True, then the index of failed operation can be accessed using *get_failed_operation_index()*, and the execution result of failed operation can be accessed using *get_failed_operation_result()*.

Methods Summary

get_failed_operation_index()	Returns the index of failed operation that results in the entire WriteMultiple operation aborting.
get_failed_operation_result()	Returns the result of the operation that results in the entire WriteMultiple operation aborting.
get_read_kb()	Returns the read throughput consumed by this operation, in KBytes.
get_read_units()	Returns the read throughput consumed by this operation, in read units.
get_results()	Returns the list of execution results for the operations.
get_success()	Returns True if the WriteMultiple operation succeeded, or False if the operation is aborted due to the failure of a sub operation.
get_write_kb()	Returns the write throughput consumed by this operation, in KBytes.
get_write_units()	Returns the write throughput consumed by this operation, in write units.
size()	Returns the number of results.

Methods Documentation

get_failed_operation_index()
Returns the index of failed operation that results in the entire WriteMultiple operation aborting.

Returns the index of operation, -1 if not set.

Return type int

get_failed_operation_result()
Returns the result of the operation that results in the entire WriteMultiple operation aborting.

Returns the result of the operation, None if not set.

Return type *OperationResult* or None

get_read_kb()
Returns the read throughput consumed by this operation, in KBytes. This is the actual amount of data read by the operation. The number of read units consumed is returned by *get_read_units()* which may be a larger number because this was an update operation.

Returns the read KBytes consumed.

Return type int

get_read_units()
Returns the read throughput consumed by this operation, in read units. This number may be larger than that returned by *get_read_kb()* because it was an update operation.

Returns the read units consumed.

Return type int

get_results()
Returns the list of execution results for the operations.

Returns the list of execution results.

Return type list(*OperationResult*)

get_success()
Returns True if the WriteMultiple operation succeeded, or False if the operation is aborted due to the failure of a sub operation.

The failed operation index can be accessed using *get_failed_operation_index()* and its result can be accessed using *get_failed_operation_result()*.

Returns True if the operation succeeded.

Return type bool

get_write_kb()
Returns the write throughput consumed by this operation, in KBytes.

Returns the write KBytes consumed.

Return type int

get_write_units()
Returns the write throughput consumed by this operation, in write units.

Returns the write units consumed.

Return type int

size()
Returns the number of results.

Returns the number of results.

Return type int

WriteThrottlingException

exception borneo.**WriteThrottlingException**(*message*)
Cloud service only.

This exception indicates that the provisioned write throughput has been exceeded.

Operations resulting in this exception can be retried but it is recommended that callers use a delay before retrying in order to minimize the chance that a retry will also be throttled. Applications should attempt to avoid throttling exceptions by rate limiting themselves to the degree possible.

4.2 borneo.iam Package

4.2.1 Classes

SignatureProvider([provider, config_file, . . .])	Cloud service only.

SignatureProvider

class borneo.iam.**SignatureProvider**(*provider=None*, *config_file=None*, *profile_name=None*, *tenant_id=None*, *user_id=None*, *fingerprint=None*, *private_key=None*, *pass_phrase=None*, *region=None*, *duration_seconds=240*, *refresh_ahead=10*)

Bases: borneo.auth.AuthorizationProvider

Cloud service only.

An instance of *borneo.AuthorizationProvider* that generates and caches signature for each request as authorization string. A number of pieces of information are required for configuration. See Required Keys and OCIDs for information and instructions on how to create the required keys and OCIDs for configuration. The required information includes:

- A signing key, used to sign requests.
- A pass phrase for the key, if it is encrypted.
- The fingerprint of the key pair used for signing.
- The OCID of the tenancy.
- The OCID of a user in the tenancy.

All of this information is required to authenticate and authorize access to the service. See *Acquire Credentials for the Oracle NoSQL Database Cloud Service* for information on how to acquire this information.

There are three different ways to authorize an application:

1. Using a specific user's identity.
2. Using an Instance Principal, which can be done when running on a compute instance in the Oracle Cloud Infrastructure (OCI). See *create_with_instance_principal()* and Calling Services from Instances.
3. Using a Resource Principal, which can be done when running within a Function within the Oracle Cloud Infrastructure (OCI). See *create_with_resource_principal()* and Accessing Other Oracle Cloud Infrastructure Resources from Running Functions.

The latter 2 limit the ability to use a compartment name vs OCID when naming compartments and tables in Request classes and when naming tables in queries. A specific user identity is best for naming flexibility, allowing both compartment names and OCIDs.

When using a specific user's identity there are 3 options for providing the required information:

1. Using a instance of oci.signer.Signer or oci.auth.signers.SecurityTokenSigner
2. Directly providing the credentials via parameters
3. Using a configuration file

Only one method of providing credentials can be used, and if they are mixed the priority from high to low is:

- Signer or SecurityTokenSigner(provider is used)
- Credentials as arguments (tenant_id, etc used)
- Configuration file (config_file is used)

Parameters

- **provider** (*Signer or SecurityTokenSigner*) – an instance of oci.signer.Signer or oci.auth.signers.SecurityTokenSigner.
- **config_file** (*str*) – path of configuration file.

- **profile_name** (*str*) – user profile name. Only valid with config_file.
- **tenant_id** (*str*) – id of the tenancy
- **user_id** (*str*) – id of a specific user
- **private_key** (*str*) – path to private key or private key content
- **fingerprint** (*str*) – fingerprint for the private key
- **pass_phrase** (*str*) – pass_phrase for the private key if created
- **region** (Region) – identifies the region will be accessed by the NoSQLHandle
- **duration_seconds** (*int*) – the signature cache duration in seconds.
- **refresh_ahead** (*int*) – the refresh time before signature cache expiry in seconds.

Raises ***IllegalArgumentException*** – raises the exception if the parameters are not valid.

Attributes Summary

DEFAULT_REFRESH_AHEAD	Default refresh time before signature expiry, 10 seconds.
MAX_ENTRY_LIFE_TIME	Maximum lifetime of signature 240 seconds.

Methods Summary

close()	Closes the signature provider.
create_with_instance_principal([...])	Creates a SignatureProvider using an instance principal.
create_with_resource_principal([logger])	Creates a SignatureProvider using a resource principal.
get_authorization_string([request])	Returns an authorization string for the specified request.
get_logger()	Returns the logger of this provider if set, None if not.
get_resource_principal_claim(key)	Resource principal session tokens carry JWT claims.
set_logger(logger)	Sets a logger instance for this provider.

Attributes Documentation

DEFAULT_REFRESH_AHEAD = 10
Default refresh time before signature expiry, 10 seconds.

MAX_ENTRY_LIFE_TIME = 240
Maximum lifetime of signature 240 seconds.

Methods Documentation

close()
Closes the signature provider.

static create_with_instance_principal(*iam_auth_uri=None*, *region=None*, *logger=None*)
Creates a SignatureProvider using an instance principal. This method may be used when calling the Oracle

NoSQL Database Cloud Service from an Oracle Cloud compute instance. It authenticates with the instance principal and uses a security token issued by IAM to do the actual request signing.

When using an instance principal the compartment id (OCID) must be specified on each request or defaulted by using *borneo.NoSQLHandleConfig.set_default_compartment()*. If the compartment is not specified for an operation an exception will be thrown.

See Calling Services from Instances

Parameters

- **iam_auth_uri** (*str*) – the URI is usually detected automatically, specify the URI if you need to overwrite the default, or encounter the *Invalid IAM URI* error, it is optional.
- **region** (Region) – identifies the region will be accessed by the NoSQLHandle, it is optional.
- **logger** (*Logger*) – the logger used by the SignatureProvider, it is optional.

Returns a SignatureProvider.

Return type *SignatureProvider*

static create_with_resource_principal(*logger=None*)

Creates a SignatureProvider using a resource principal. This method may be used when calling the Oracle NoSQL Database Cloud Service from other Oracle Cloud service resource such as Functions. It uses a resource provider session token (RPST) that enables the resource such as function to authenticate itself.

When using an resource principal the compartment id (OCID) must be specified on each request or defaulted by using *borneo.NoSQLHandleConfig.set_default_compartment()*. If the compartment id is not specified for an operation an exception will be thrown.

See Accessing Other Oracle Cloud Infrastructure Resources from Running Functions.

Parameters **logger** (*Logger*) – the logger used by the SignatureProvider, it is optional.

Returns a SignatureProvider.

Return type *SignatureProvider*

get_authorization_string(*request=None*)

Returns an authorization string for the specified request. The string is sent to the server in the request and is used for authorization. Authorization information can be request-dependent.

Parameters **request** (Request) – the request to be issued. This is an instance of Request().

Returns a string indicating that the application is authorized to perform the request.

Return type str

get_logger()

Returns the logger of this provider if set, None if not.

Returns the logger.

Return type Logger or None

get_resource_principal_claim(*key*)

Resource principal session tokens carry JWT claims. Permit the retrieval of the value from the token by given key. See *borneo.ResourcePrincipalClaimKeys*.

Parameters **key** (*str*) – the name of a claim in the session token.

Returns the claim value.

Return type str

set_logger(*logger*)
Sets a logger instance for this provider. If not set, the logger associated with the driver is used.

Parameters **logger** (*Logger*) – the logger to use.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if logger is not an instance of Logger.

4.3 borneo.kv Package

4.3.1 Classes

AuthenticationException(message[, cause])	On-premise only.
StoreAccessTokenProvider([user_name, password])	On-premise only.

AuthenticationException

exception borneo.kv.**AuthenticationException**(*message*, *cause=None*)
On-premise only.

This exception is thrown when use StoreAccessTokenProvider in following cases:

Authentication information was not provided in the request header.

The authentication session has expired. By default *StoreAccessTokenProvider* will automatically retry authentication operation based on its authentication information.

StoreAccessTokenProvider

class borneo.kv.**StoreAccessTokenProvider**(*user_name=None*, *password=None*)
Bases: borneo.auth.AuthorizationProvider

On-premise only.

StoreAccessTokenProvider is an *borneo.AuthorizationProvider* that performs the following functions:

Initial (bootstrap) login to store, using credentials provided.

Storage of bootstrap login token for re-use.

Optionally renews the login token before it expires.

Logs out of the store when closed.

If accessing an insecure instance of Oracle NoSQL Database the default constructor is used, with no arguments.

If accessing a secure instance of Oracle NoSQL Database a user name and password must be provided. That user must already exist in the NoSQL Database and have sufficient permission to perform table operations. That user's identity is used to authorize all database operations.

To access to a store without security enabled, no parameter need to be set to the constructor.

To access to a secure store, the constructor requires a valid user name and password to access the target store. The user must exist and have sufficient permission to perform table operations required by the application. The user identity is used to authorize all operations performed by the application.

Parameters

- **user_name** (*str*) – the user name to use for the store. This user must exist in the NoSQL Database and is the identity that is used for authorizing all database operations.
- **password** (*str*) – the password for the user.

Raises ***IllegalArgumentException*** – raises the exception if one or more of the parameters is malformed or a required parameter is missing.

Methods Summary

close()	Close the provider, releasing resources such as a stored login token.
get_logger()	Returns the logger of this provider if set, None if not.
is_auto_renew()	Returns whether the login token is to be automatically renewed.
is_secure()	Returns whether the provider is accessing a secured store.
set_auto_renew(auto_renew)	Sets the auto-renew state.
set_logger(logger)	Sets a logger instance for this provider.

Methods Documentation

__init__(*user_name=None*, *password=None*)
Creates a StoreAccessTokenProvider

Parameters

- **user_name** (*str*) – the user name to use for the store. This user must exist in the NoSQL Database and is the identity that is used for authorizing all database operations.
- **password** (*str*) – the password for the user.

Raises ***IllegalArgumentException*** – raises the exception if one or more of the parameters is malformed or a required parameter is missing.

close()
Close the provider, releasing resources such as a stored login token.

get_logger()
Returns the logger of this provider if set, None if not.

Returns the logger.

Return type Logger or None

is_auto_renew()
Returns whether the login token is to be automatically renewed.

Returns True if auto-renew is set, otherwise False.

Return type bool

is_secure()
Returns whether the provider is accessing a secured store.

Returns True if accessing a secure store, otherwise False.

Return type bool

set_auto_renew(*auto_renew*)
Sets the auto-renew state. If True, automatic renewal of the login token is enabled.

Parameters **auto_renew** (*bool*) – set to True to enable auto-renew.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if auto_renew is not True or False.

set_logger(*logger*)
Sets a logger instance for this provider. If not set, the logger associated with the driver is used.

Parameters **logger** (*Logger*) – the logger to use.

Returns self.

Raises ***IllegalArgumentException*** – raises the exception if logger is not an instance of Logger.

CHAPTER 5

How to find client statistics

StatsControl allows user to control the collection of driver statistics at runtime.

The statistics data is collected for an interval of time. At the end of the interval, the stats data is logged in a specified JSON format that can be filtered and parsed. After the logging, the counters are cleared and collection of data resumes.

Collection intervals are aligned to the top of the hour. This means first interval logs may contain stats for a shorter interval.

5.1 How to enable and configure from command line

Collection of stats are controlled by the following environment variables:

- **`NOSQL_STATS_PROFILE=[none|regular|more|all]`** Specifies the stats profile:
  - `none` - disabled.
  - **`regular` - per request: counters, errors, latencies, delays, retries.** This incurs minimum overhead.
  - **`more` - stats above with 95th and 99th percentile latencies.** This may add 0.5% overhead compared to none stats profile.
  - **`all` - stats above with per query information.** This may add 1% overhead compared to none stats profile.
- **`NOSQL_STATS_INTERVAL=600`** Interval in seconds to log the stats, by default is 10 minutes.
- **`NOSQL_STATS_PRETTY_PRINT=true`** Option to enable pretty printing of the JSON data, default value is false.

5.2 How to enable and configure using the API

Collection of stats can also be used by using the API: `NoSQLHandleConfig.set_stats_profile()` or `StatsControl.set_profile()`. At runtime stats collection can be enabled selectively by using `StatsControl.start()` ond `StatsControl.stop()`. The following example shows how to use a stats handler and how to control the stats at runtime:

```
def stats_handler(stats):
    # type: (Dict) -> None
    print("Stats : " + str(stats))
...
config = NoSQLHandleConfig( endpoint )
config.set_stats_profile(StatsProfile.REGULAR)
config.set_stats_interval(600)
config.set_stats_pretty_print(False)
config.set_stats_handler(stats_handler)

handle = NoSQLHandle(config)

handle = get_handle(tenant_id)

stats_control = handle.get_stats_control()

#... application code without stats

# enable observations
stats_control.start();

#... application code with REGULAR stats

# For particular parts of code profile can be changed to collect more stats.
stats_control.set_stats_profile(StatsProfile.ALL)
#... more sensitive code with ALL stats

stats_control.set_stats_profile(StatsProfile.REGULAR)
#... application code with REGULAR stats

# disable observations
stats_control.stop()

#... application code without stats
handle.close()
```

5.3 Example log entry

The following is an example of stats log entry using the ALL profile:

- A one time entry containing stats id and options:

```
INFO: Client stats|{    // INFO log entry
"sdkName" : "Oracle NoSQL SDK for Python",  // SDK name
"sdkVersion" : "5.2.4",                  // SDK version
"clientId" : "f595b333",                   // NoSQLHandle id
"profile" : "ALL",                         // stats profile
"intervalSec" : 600,                       // interval length in seconds
```

(continues on next page)

(continued from previous page)

```
"prettyPrint" : true,                       // JSON pretty print
"rateLimitingEnabled" : false}              // if rate limiting is enabled
```

- An entry at the end of each interval containing the stats values:

```
INFO: Client stats|{
"clientId" : "b7bc7734",                 // id of NoSQLHandle object
"startTime" : "2021-09-20T20:11:42Z", // UTC start interval time
"endTime" : "2021-09-20T20:11:47Z",   // UTC end interval time
"requests" : [{                          // array of types of requests
  "name" : "Get",                          // stats for GET request type
  "httpRequestCount" : 2,                  // count of http requests
  "errors" : 0,                            // number of errors in interval
  "httpRequestLatencyMs" : {               // response time of http requests
    "min" : 4,                               // minimum value in interval
    "avg" : 4.5,                             // average value in interval
    "max" : 5,                               // maximum value in interval
    "95th" : 5,                              // 95th percentile value
    "99th" : 5                               // 99th percentile value
  },
  "requestSize" : {                        // http request size in bytes
    "min" : 42,                              // minimum value in interval
    "avg" : 42.5,                            // average value in interval
    "max" : 43                               // maximum value in interval
  },
  "resultSize" : {                         // http result size in bytes
    "min" : 193,                             // minimum value in interval
    "avg" : 206.5,                           // average value in interval
    "max" : 220                              // maximum value in interval
  },
  "rateLimitDelayMs" : 0,                  // delay in milliseconds introduced by the␣
↪rate limiter
  "retry" : {                              // retries
    "delayMs" : 0,                           // delay in milliseconds introduced by␣
↪retries
    "authCount" : 0,                         // no of auth retries
    "throttleCount" : 0,                     // no of throttle retries
    "count" : 0                              // total number of retries
  }
}, {
  "name" : "Query",                      // stats for all QUERY type requests
  "httpRequestCount" : 14,
  "errors" : 0,
  "httpRequestLatencyMs" : {
    "min" : 3,
    "avg" : 13.0,
    "max" : 32,
    "95th" : 32,
    "99th" : 32
  },
  "resultSize" : {
    "min" : 146,
    "avg" : 7379.71,
    "max" : 10989
  },
  "requestSize" : {
    "min" : 65,
```

(continues on next page)

(continued from previous page)

```
    "avg" : 709.85,
    "max" : 799
  },
  "rateLimitDelayMs" : 0,
  "retry" : {
    "delayMs" : 0,
    "authCount" : 0,
    "throttleCount" : 0,
    "count" : 0
  }
}, {
  "name" : "Put",                     // stats for PUT type requests
  "httpRequestCount" : 1002,
  "errors" : 0,
  "httpRequestLatencyMs" : {
    "min" : 1,
    "avg" : 4.41,
    "max" : 80,
    "95th" : 8,
    "99th" : 20
  },
  "requestSize" : {
    "min" : 90,
    "avg" : 90.16,
    "max" : 187
  },
  "resultSize" : {
    "min" : 58,
    "avg" : 58.0,
    "max" : 58
  },
  "rateLimitDelayMs" : 0,
  "retry" : {
    "delayMs" : 0,
    "authCount" : 0,
    "throttleCount" : 0,
    "count" : 0
  }
}],
"queries" : [{            // query stats aggregated by query statement
                            // query statement
  "query" : "SELECT * FROM audienceData ORDER BY cookie_id",
                            // query plan description

  "plan" : "SFW([6])
     [
       FROM:
         RECV([3])
           [
             DistributionKind : ALL_PARTITIONS,
             Sort Fields : sort_gen,
           ] as $from-0
       SELECT:
        FIELD_STEP([6])
          [
           VAR_REF($from-0)([3]),
           audienceData
```

(continues on next page)

(continued from previous page)

```
          ]
     ]",
  "doesWrites" : false,
  "httpRequestCount" : 12,  // number of http calls to the server
  "unprepared" : 1,         // number of query requests without prepare
  "simple" : false,         // type of query
  "count" : 20,             // number of handle.query() API calls
  "errors" : 0,             // number of calls trowing exception
  "httpRequestLatencyMs" : {// response time of http requests in milliseconds
    "min" : 8,                // minimum value in interval
    "avg" : 14.58,            // average value in interval
    "max" : 32,               // maximum value in interval
    "95th" : 32,              // 95th percentile value in interval
    "99th" : 32               // 99th percentile value in interval
  },
  "requestSize" : {         // http request size in bytes
    "min" : 65,               // minimum value in interval
    "avg" : 732.5,            // average value in interval
    "max" : 799               // maximum value in interval
  },
  "resultSize" : {          // http result size in bytes
    "min" : 914,              // minimum value in interval
    "avg" : 8585.33,          // average value in interval
    "max" : 10989             // maximum value in interval
  },
  "rateLimitDelayMs" : 0,   // total delay introduced by rate limiter in milliseconds
  "retry" : {               // automatic retries
    "delayMs" : 0,            // delay introduced by retries
    "authCount" : 0,          // count of auth related retries
    "throttleCount" : 0,      // count of throttle related retries
    "count" : 0               // total count of retries
  }
}]
```

The log entries go to the logger configured in NoSQLHandlerConfig. By default, if no logger is configured the statistics entries, if enabled, will be logged to file **logs/driver.log** in the local directory.

Stats collection is not dependent of logging configuration, even if logging is disabled, collection of stats will still happen if stats profile other than *none* is used. In this case, the stats are available by using the stats handler.

Depending on the type of query, if client processing is required, for example in the case of ordered or aggregate queries, indicated by the false **simple** field of the **query** entry, the **count** and **httpRequestsCount** numbers will differ. **count** represents the number of `handle.query()` API calls and **httpRequestCount** represents the number of internal http requests from server. For these type of queries, the driver executes several simpler queries, per shard or partition, and than combines the results locally.

Note: connection statistics are not available for NoSQL Python driver.

Python Module Index

b

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W