Oracle® NoSQL Database

SQL Beginner's Guide

Release 25.1
E85380-32
May 2025

ORACLE

Oracle NoSQL Database SQL Beginner's Guide, Release 25.1

E85380-32

Contents

Preface

This document is intended to provide a rapid introduction to the SQL for Oracle NoSQL Database and related concepts. SQL for Oracle NoSQL Database is an easy to use SQL-like language that supports read-only queries and data definition (DDL) statements. This document focuses on the query part of the language. For a more detailed description of the language (both DDL and query statements), see *SQL Reference Guide*.

This book is aimed at developers who are looking to manipulate Oracle NoSQL Database data using a SQL-like query language. Knowledge of standard SQL is not required but it does allow you to easily learn SQL for Oracle NoSQL Database.

Conventions Used in This Book

The following typographical conventions are used within this manual:

Information that you are to type literally is presented in `monospaced` font.

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

Case-insensitive keywords, like SELECT, FROM, WHERE, ORDER BY, are presented in UPPERCASE.

Case sensitive keywords, like the function size(item) are presented in lowercase.

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

1

Introduction to SQL for Oracle NoSQL Database

Welcome to SQL for Oracle NoSQL Database. This language provides a SQL-like interface to Oracle NoSQL Database. The SQL for Oracle NoSQL Database data model supports flat relational data, hierarchical typed (schema-full) data, and schema-less JSON data. You have the flexibility to create tables with a well-defined schema for applications that require fixed data or a combination of fixed data and schema-less JSON. For pure document-oriented applications, you can use JSON collection tables that do not have any schema definition other than the primary key fields. The SQL for Oracle NoSQL Database is designed to handle all such data seamlessly without any impedance mismatch among the different sub-models. Impedance mismatch is a problem that occurs due to differences between the database model and the programming language mode.

For information on the command line shell you can use to run SQL for Oracle NoSQL Database queries, see Introduction to the SQL for Oracle NoSQL Database Shell.

2

Working with Namesapce

This chapter provides examples on how to manage namespaces.

A namespace in Oracle NoSQL Database groups tables and ensures that table names are unique within it. It enables table privilege management as a group. You can have multiple tables with the same name across different namespaces. To access these tables, you must use the fully qualified table name. A fully qualified table name begins with a namespace, followed by a table name, separated by a colon (:). For example, `ns1:table1`.

> **Note:**
>
> Namespaces are case-insensitive, so `ns1` or `NS1` are treated as same.

You can create multiple namespaces in your store. Each table belongs to a specific namespace. The default Oracle NoSQL Database namespace is `sysdefault`. You do not need a fully qualified name to access tables in the `sysdefault` namespace. For example, you can access the table by specifying `table2` instead of `sysdefault:table2`.

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

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

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

Managing Namespace

To manage namespaces, run the below commands in the SQL Shell.

CREATE NAMESPACE

Example 1: Use the `CREATE NAMESPACE` statement to add a new namespace.

```
CREATE NAMESPACE IF NOT EXISTS ns1
```

> **Note:**
>
> `IF NOT EXISTS` clause is optional.

Output:

```
Statement completed successfully
```

SHOW NAMESPACES

Example 2: Use the `show namespaces` statement to show the existing namespaces.

```
SHOW NAMESPACES
```

Output:

```
namespaces
  ns1
  sysdefault
```

Example 3: To show the namespaces in a JSON format, use the statement below

```
SHOW AS JSON NAMESPACES
```

Output:

```
{"namespaces" : ["ns1","sysdefault"]}
```

DROP NAMESPACE

To delete a namespace, use the `DROP NAMESPACE` statement

Example 4: Delete a namespace from your store.

```
DROP NAMESPACE IF EXISTS ns1 CASCADE
```

Explanation: The above statement removes the namespace, `ns1`.

- `IF EXISTS` is an optional clause. Specifying it prevents an error if the namespace doesn't exist. However, not including results in an error that the namespace is missing.
- `CASCADE` is an optional clause. It deletes the namespace and all the tables in it collectively. If not specified, the system throws an error, stating that the namespace is not empty.

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

Namespace Resolution

Namespace resolution determines which table a SQL query refers to, ensuring that the query targets the correct table, especially when multiple tables with the same name exist across different namespaces.

The rules are as follows:

- If you provide the table name with a namespace, no further resolution is needed because the namespace uniquely identifies the table.
- If you provide the table name without a namespace, the system resolves the table based on the namespace specified in the ExecuteOptions class.
- If ExecuteOptions does not specify a namespace, the system defaults to the `sysdefault` namespace to resolve the table.
- By using different namespaces in ExecuteOptions, you can execute the same queries on similar tables present in different namespace.

Namespace Privileges and Authorization

You can add multiple namespaces to your store, create tables within them, and assign specific permissions to users, allowing them to access specific namespaces and tables. Additionally, you can manage access control by authorizing which users can create and drop namespaces and indexes or modify any data within each namespace, providing greater flexibility and data handling.

To understand more about the user and role privileges, see Namespace Privileges and Permissions (Table 4-1) in *Java Direct Driver Developer's Guide.*

Before granting access to namespaces, create the following using SQL Shell.

First, create a user:

```
CREATE USER John IDENTIFIED BY "NewPwd123!!"
```

Where,

1. John is the *user_name*
2. NewPwd123!! is the *password*

Next, grant `dbadmin` privilege to user, John

```
GRANT DBADMIN TO USER John
```

Where, `DBADMIN` is a built-in *role*. See, Built-in Roles, for more predefined roles.

And now you can grant the user, John, to create tables in the `ns1` namespace.

```
GRANT CREATE_TABLE_IN_NAMESPACE ON NAMESPACE ns1 TO John
```

Now, grant permission to the user to create an index on any table in `ns1` namespace.

```
GRANT CREATE_INDEX_IN_NAMESPACE ON NAMESPACE ns1 TO John
```

Also, you can now grant permission to user to delete items in `ns1` namespace.

```
GRANT DELETE_IN_NAMESPACE ON NAMESPACE ns1 TO John
```

3
Simple SELECT Queries

This section presents examples of simple queries for relational data. To follow along with the examples, get the `Examples` download from here and run the `SQLBasicExamples` script found in the `sql` folder. The script creates the table as shown, and imports the data.

SQLBasicExamples Script

The script `SQLBasicExamples` creates the following table:

```
CREATE TABLE Users (
  id integer,
  firstname string,
  lastname string,
  age integer,
  income integer,
  primary key (id)
);
```

The script also load data into the Users table with the following rows (shown here in JSON format):

```
{
  "id":1,
  "firstname":"David",
  "lastname":"Morrison",
  "age":25,
  "income":100000,
}

{
  "id":2,
  "firstname":"John",
  "lastname":"Anderson",
  "age":35,
  "income":100000,
}

{
  "id":3,
  "firstname":"John",
  "lastname":"Morgan",
  "age":38,
  "income":null,
}

{
  "id":4,
  "firstname":"Peter",
```

```
  "lastname":"Smith",
  "age":38,
  "income":80000,
}

{
  "id":5,
  "firstname":"Dana",
  "lastname":"Scully",
  "age":47,
  "income":400000,
}
```

You run the SQLBasicExamples script using the load command:

```
> cd <installdir>/examples/sql
> java -jar <KVHOME>/lib/sql.jar -helper-hosts <host>:<port> \
-store <storename> load \
-file <KVHOME>/examples/sql/SQLBasicExamples.cli
```

Starting the SQL Shell

You can run SQL queries and execute DDL statements directly from the SQL shell. This is described in Introduction to the SQL for Oracle NoSQL Database Shell. To run the queries shown in this document, start the SQL shell as follows:

```
java -jar KVHOME/lib/sql.jar
-helper-hosts node01:5000 -store kvstore
sql->
```

> **Note:**
>
> This document shows examples displayed in COLUMN mode, although the default output type is JSON. Use the `mode` command to toggle between COLUMN and JSON (or JSON pretty) output.

Choosing column data

You can choose columns from a table. To do so, list the names of the desired table columns after SELECT in the statement, before noting the table after the FROM clause.

The FROM clause can name only one table. To retrieve data from a child table, use dot notation, such as parent.child.

To choose all table columns, use the asterisk (*) wildcard character as follows:

```
sql-> SELECT * FROM Users;
```

The SELECT statement displays these results:

```
 +----+-----------+----------+-----+--------+
 | id | firstname | lastname | age | income |
 +----+-----------+----------+-----+--------+
 |  3 | John      | Morgan   |  38 |   NULL |
 |  4 | Peter     | Smith    |  38 |  80000 |
 |  2 | John      | Anderson |  35 | 100000 |
 |  5 | Dana      | Scully   |  47 | 400000 |
 |  1 | David     | Morrison |  25 | 100000 |
 +----+-----------+----------+-----+--------+

5 rows returned
```

To choose specific column(s) from the table Users, include the column names as a comma-separated list in the SELECT statement:

```
sql-> SELECT firstname, lastname, age FROM Users;
 +-----------+----------+-----+
 | firstname | lastname | age |
 +-----------+----------+-----+
 | John      | Morgan   |  38 |
 | David     | Morrison |  25 |
 | Dana      | Scully   |  47 |
 | Peter     | Smith    |  38 |
 | John      | Anderson |  35 |
 +-----------+----------+-----+

5 rows returned
```

Substituting column names for a query

You can use a different name for a column during a SELECT statement. Substituting a name in a query does not change the column name, but uses the substitute in the returned data returned. In the next example, the query substitutes Surname for the actual column name lastname, by using the actual-name AS substitute-name clause, in the SELECT statement.

```
sql-> SELECT lastname AS Surname FROM Users;
 +----------+
 | Surname  |
 +----------+
 | Scully   |
 | Smith    |
 | Morgan   |
 | Anderson |
 | Morrison |
 +----------+

5 rows returned
```

Computing values for new columns

The SELECT statement can contain computational expressions based on the values of existing columns. For example, in the next statement, you select the values of one column, income, divide each value by 12, and display the output in another column. The SELECT statement can use almost any type of expression. If more than one value is returned, the items are inserted into an array.

This SELECT statement uses the yearly income values divided by 12 to calculate the corresponding values for `monthlysalary`:

```
sql-> SELECT id, lastname, income, income/12
AS monthlysalary FROM users;
 +----+----------+--------+---------------+
 | id | lastname | income | monthlysalary |
 +----+----------+--------+---------------+
 |  2 | Anderson | 100000 |          8333 |
 |  1 | Morrison | 100000 |          8333 |
 |  5 | Scully   | 400000 |         33333 |
 |  4 | Smith    |  80000 |          6666 |
 |  3 | Morgan   |   NULL |          NULL |
 +----+----------+--------+---------------+

5 rows returned
```

This SELECT statement performs an addition operation that adds a bonus of 5000 to income to return `salarywithbonus`:

```
sql-> SELECT id, lastname, income, income+5000
AS salarywithbonus FROM users;
 +----+----------+--------+-----------------+
 | id | lastname | income | salarywithbonus |
 +----+----------+--------+-----------------+
 |  4 | Smith    |  80000 |           85000 |
 |  1 | Morrison | 100000 |          105000 |
 |  5 | Scully   | 400000 |          405000 |
 |  3 | Morgan   |   NULL |            NULL |
 |  2 | Anderson | 100000 |          105000 |
 +----+----------+--------+-----------------+

5 rows returned
```

Identifying tables and their columns

The FROM clause can contain one table only (that is, joins are not supported). The table is specified by its name, which may be followed by an optional alias. The table can be referenced in the other clauses either by its name or its alias. As we will see later, sometimes the use of the table name or alias is mandatory. However, for table columns, the use of the table name or alias is optional. For example, here are three ways to write the same query:

```
sql-> SELECT Users.lastname, age FROM Users;
 +----------+-----+
```

```
 | lastname | age |
 +----------+-----+
 | Scully   |  47 |
 | Smith    |  38 |
 | Morgan   |  38 |
 | Anderson |  35 |
 | Morrison |  25 |
 +----------+-----+

5 rows returned
```

To identify the table Users with the alias `u`:

```
sql-> SELECT lastname, u.age FROM Users u ;
```

The keyword AS can optionally be used before an alias. For example, to identify the table Users with the alias `People`:

```
sql-> SELECT People.lastname, People.age FROM Users AS People;
```

Filtering Results

You can filter query results by specifying a filter condition in the WHERE clause. Typically, a filter condition consists of one or more comparison expressions connected through logical operators AND or OR. The comparison operators are also supported: =, !=, >, >=, <, and <= .

This query filters results to return only users whose first name is John:

```
sql-> SELECT id, firstname, lastname FROM Users WHERE firstname = "John";
 +----+-----------+----------+
 | id | firstname | lastname |
 +----+-----------+----------+
 |  3 | John      | Morgan   |
 |  2 | John      | Anderson |
 +----+-----------+----------+

2 rows returned
```

To return users whose calculated `monthlysalary` is greater than 6000:

```
sql-> SELECT id, lastname, income, income/12 AS monthlysalary
FROM Users WHERE income/12 > 6000;
 +----+----------+--------+---------------+
 | id | lastname | income | monthlysalary |
 +----+----------+--------+---------------+
 |  5 | Scully   | 400000 |         33333 |
 |  4 | Smith    |  80000 |          6666 |
 |  2 | Anderson | 100000 |          8333 |
 |  1 | Morrison | 100000 |          8333 |
 +----+----------+--------+---------------+

5 rows returned
```

To return users whose age is between 30 and 40 or whose income is greater than 100,000:

```
sql-> SELECT lastname, age, income FROM Users
WHERE age >= 30 and age <= 40 or income > 100000;
 +----------+-----+--------+
 | lastname | age | income |
 +----------+-----+--------+
 | Smith    |  38 |  80000 |
 | Morgan   |  38 |   NULL |
 | Anderson |  35 | 100000 |
 | Scully   |  47 | 400000 |
 +----------+-----+--------+

4 rows returned
```

You can use parenthesized expressions to alter the default precedence among operators. For example:

To return the users whose age is greater than 40 and either their age is less than 30 or their income is greater or equal than 100,000:

```
sql-> SELECT id, lastName FROM Users WHERE
(income >= 100000 or age < 30) and age > 40;
 +----+----------+
 | id | lastName |
 +----+----------+
 |  5 | Scully   |
 +----+----------+

1 row returned
```

You can use the IS NULL condition to return results where a field column value is set to `SQL NULL` (`SQL NULL` is used when a non-JSON field is set to null):

```
sql-> SELECT id, lastname from Users WHERE income IS NULL;
 +----+----------+
 | id | lastname |
 +----+----------+
 |  3 | Morgan   |
 +----+----------+

1 row returned
```

You can use the IS NOT NULL condition to return column values that contain non-null data:

```
sql-> SELECT id, lastname from Users WHERE income IS NOT NULL;
 +----+----------+
 | id | lastname |
 +----+----------+
 |  4 | Smith    |
 |  1 | Morrison |
 |  5 | Scully   |
 |  2 | Anderson |
 +----+----------+
```

```
4 rows returned
```

Grouping Results

Use the GROUP BY clause to group the results by one or more table columns. Typically, a GROUP BY clause is used in conjunction with an aggregate expression such as COUNT, SUM, and AVG.

> **Note:**
>
> You can use the GROUP BY clause only if there exists an index that sorts the rows by the grouping columns.

For example, this query returns the average income of users, based on their age.

```
sql-> SELECT age, AVG(income) FROM Users GROUP BY age;
 +-------+-------------+
 | age   | AVG(income) |
 +-------+-------------+
 |  25   |  100000     |
 |  35   |  100000     |
 |  38   |  80000      |
 |  47   |  400000     |
 +-------+-------------+

4 rows returned
```

Ordering Results

Use the ORDER BY clause to order the results by a primary key column or a non-primary key column.

To order using the required column, specify the sort column in the ORDER BY clause:

ORDER BY using the primary key column:

```
SELECT id, lastname FROM Users ORDER BY id;
 +----+----------+
 | id | lastname |
 +----+----------+
 |  1 | Morrison |
 |  2 | Anderson |
 |  3 | Morgan   |
 |  4 | Smith    |
 |  5 | Scully   |
 +----+----------+
```

ORDER BY using a non-primary key column:

```
SELECT id, lastname FROM Users ORDER BY lastname;
 +----+----------+
 | id | lastname |
 +----+----------+
 |  2 | Anderson |
 |  3 | Morgan   |
 |  1 | Morrison |
 |  5 | Scully   |
 |  4 | Smith    |
 +----+----------+
```

Using this example data, you can order by more than one column. For example, to order users by age and income:

```
SELECT id, lastname, age, income FROM Users ORDER BY age, income;
 +----+----------+-----+--------+
 | id | lastname | age | income |
 +----+----------+-----+--------+
 |  1 | Morrison |  25 | 100000 |
 |  2 | Anderson |  35 | 100000 |
 |  4 | Smith    |  38 |  80000 |
 |  3 | Morgan   |  38 |   NULL |
 |  5 | Scully   |  47 | 400000 |
 +----+----------+-----+--------+
```

By default, sorting is performed in ascending order. To sort in descending order use the DESC keyword in the ORDER BY clause:

```
SELECT id, lastname FROM Users ORDER BY id DESC;
 +----+----------+
 | id | lastname |
 +----+----------+
 |  5 | Scully   |
 |  4 | Smith    |
 |  3 | Morgan   |
 |  2 | Anderson |
 |  1 | Morrison |
 +----+----------+
```

Limiting and Offsetting Results

Use the `LIMIT` clause to limit the number of results returned from a SELECT statement. For example, if there are 1000 rows in the Users table, limit the number of rows to return by specifying a LIMIT value. For example, this statement returns the first four ID rows from the table:

```
sql-> SELECT * from Users ORDER BY id LIMIT 4;
 +----+-----------+----------+-----+--------+
 | id | firstname | lastname | age | income |
 +----+-----------+----------+-----+--------+
 |  1 | David     | Morrison |  25 | 100000 |
```

```
 |  2 | John      | Anderson |  35 | 100000 |
 |  3 | John      | Morgan   |  38 |   NULL |
 |  4 | Peter     | Smith    |  38 |  80000 |
 +----+-----------+----------+-----+--------+

4 rows returned
```

To return only results 3 and 4 from the 10000 rows use the `LIMIT` clause to indicate 2 values, and the OFFSET clause to specify where the offset begins (after the first two rows). For example:

```
sql-> SELECT * from Users ORDER BY id LIMIT 2 OFFSET 2;
 +----+-----------+----------+-----+--------+
 | id | firstname | lastname | age | income |
 +----+-----------+----------+-----+--------+
 |  3 | John      | Morgan   |  38 |   NULL |
 |  4 | Peter     | Smith    |  38 |  80000 |
 +----+-----------+----------+-----+--------+

2 rows returned
```

> **Note:**
>
> We recommend using LIMIT and OFFSET with an ORDER BY clause. Otherwise, the results are returned in a random order, producing unpredictable results.

Using External Variables

Using external variables lets a query to written and compiled once, and then run multiple times with different values for the external variables. Binding the external variables to specific values is done through APIs, which you use before executing the query.

You must declare external variables in your SQL query before referencing them in the SELECT statement. For example:

```
DECLARE $age integer;
SELECT firstname, lastname, age
FROM Users
WHERE age > $age;
```

If the variable $age is set to value 39, the result of the above query is:

```
+-----------+----------+-----+
| firstname | lastname | age |
+-----------+----------+-----+
| Dana      | Scully   |  47 |
+-----------+----------+-----+
```

4

Working with complex data

In this chapter, we present query examples that use complex data types (arrays, maps, records). To follow along with the examples, get the `Examples` download from here and run the `SQLAdvancedExamples` script found in the `sql` folder. This script creates the table and imports the data used.

SQLAdvancedExamples Script

The `SQLAdvancedExamples` script creates the following table:

```
CREATE TABLE Persons (
  id integer,
  firstname string,
  lastname string,
  age integer,
  income integer,
  lastLogin timestamp(4),
  address record(street string,
                 city string,
                 state string,
                 phones array(record(type enum(work, home),
                                     areacode integer,
                                     number integer
                                    )
                              )
                ),
  connections array(integer),
  expenses map(integer),
  primary key (id)
);
```

The script also imports the following table rows:

```
{
  "id":1,
  "firstname":"David",
  "lastname":"Morrison",
  "age":25,
  "income":100000,
  "lastLogin" : "2016-10-29T18:43:59.8319",
  "address":{"street":"150 Route 2",
             "city":"Antioch",
             "state":"TN",
             "zipcode" : 37013,
             "phones":[{"type":"home", "areacode":423,
                        "number":8634379}]
            },
```

```
  "connections":[2, 3],
  "expenses":{"food":1000, "gas":180}
}

{
  "id":2,
  "firstname":"John",
  "lastname":"Anderson",
  "age":35,
  "income":100000,
  "lastLogin" : "2016-11-28T13:01:11.2088",
  "address":{"street":"187 Hill Street",
             "city":"Beloit",
             "state":"WI",
             "zipcode" : 53511,
             "phones":[{"type":"home", "areacode":339,
             "number":1684972}]
            },
  "connections":[1, 3],
  "expenses":{"books":100, "food":1700, "travel":2100}
}

{
  "id":3,
  "firstname":"John",
  "lastname":"Morgan",
  "age":38,
  "income":100000000,
  "lastLogin" : "2016-11-29T08:21:35.4971",
  "address":{"street":"187 Aspen Drive",
             "city":"Middleburg",
             "state":"FL",
             "phones":[{"type":"work", "areacode":305,
                        "number":1234079},
                       {"type":"home", "areacode":305,
                        "number":2066401}
                      ]
            },
  "connections":[1, 4, 2],
  "expenses":{"food":2000, "travel":700, "gas":10}
}

{
  "id":4,
  "firstname":"Peter",
  "lastname":"Smith",
  "age":38,
  "income":80000,
  "lastLogin" : "2016-10-19T09:18:05.5555",
  "address":{"street":"364 Mulberry Street",
             "city":"Leominster",
             "state":"MA",
             "phones":[{"type":"work", "areacode":339,
                        "number":4120211},
                       {"type":"work", "areacode":339,
                        "number":8694021},
```

```
                      {"type":"home", "areacode":339,
                       "number":1205678},
                      {"type":"home", "areacode":305,
                       "number":8064321}
                     ]
            },
  "connections":[3, 5, 1, 2],
  "expenses":{"food":6000, "books":240, "clothes":2000, "shoes":1200}
}

{
  "id":5,
  "firstname":"Dana",
  "lastname":"Scully",
  "age":47,
  "income":400000,
  "lastLogin" : "2016-11-08T09:16:46.3929",
  "address":{"street":"427 Linden Avenue",
             "city":"Monroe Township",
             "state":"NJ",
             "phones":[{"type":"work", "areacode":201,
                        "number":3213267},
                       {"type":"work", "areacode":201,
                        "number":8765421},
                       {"type":"home", "areacode":339,
                        "number":3414578}
                      ]
           },
  "connections":[2, 4, 1, 3],
  "expenses":{"food":900, "shoes":1000, "clothes":1500}
}
```

You run the SQLAdvancedExamples script using the load command:

```
> cd <installdir>/examples/sql
> java -jar <KVHOME>/lib/sql.jar -helper-hosts <host>:<port> \
-store <storename> load \
-file <KVHOME>/examples/sql/SQLAdvancedExamples.cli
```

> **Note:**
>
> The Persons table schema models people that can be connected to other people in the table. All connections are stored in the "connections" column, which consists of an array of integers. Each integer is an ID of a person with whom the subject is connected. The entries in the "connections" array are sorted in descending order, indicating the strength of the connection. For example, looking at the record for person 3, we see that John Morgan has these connections: [1, 4, 2]. The order of the array elements specifies that John is most strongly connected with person 1, less connected with person 4, and least connected with person 2.
>
> Records in the Persons table also include an "expenses" column, declared as an integer map. For each person, the map stores key-value pairs of string item types and integers representing money spent on the item. For example, one record has these expenses: {"food":900, "shoes":1000, "clothes":1500}, other records have different items. One benefit of modelling expenses as a map type is to facilitate the categories being different for each person. Later, we may want to add or delete categories dynamically, without changing the table schema, which maps readily support. An item to note about this map is that it is an integer map always contains key-value pairs, and keys are always strings.

Working with Timestamps

To specify a timestamp value in a query, provide it as a string, and cast it to a Timestamp data type. For example:

```
sql-> SELECT id, firstname, lastname FROM Persons WHERE
lastLogin = CAST("2016-10-19T09:18:05.5555" AS TIMESTAMP);
 +----+-----------+----------+
 | id | firstname | lastname |
 +----+-----------+----------+
 |  4 | Peter     | Smith    |
 +----+-----------+----------+

1 row returned
```

Timestamp queries often involve a range of time, which requires multiple casts:

```
sql-> SELECT id, firstname, lastname, lastLogin FROM Persons WHERE
lastLogin > CAST("2016-11-01" AS TIMESTAMP) AND
lastLogin < CAST("2016-11-30" AS TIMESTAMP);
 +----+-----------+----------+-------------------------+
 | id | firstname | lastname |        lastLogin        |
 +----+-----------+----------+-------------------------+
 |  3 | John      | Morgan   | 2016-11-29T08:21:35.4971 |
 |  2 | John      | Anderson | 2016-11-28T13:01:11.2088 |
 |  5 | Dana      | Scully   | 2016-11-08T09:16:46.3929 |
 +----+-----------+----------+-------------------------+

3 rows returned
```

You can also use various Timestamp functions to return specific time and date values from the Timestamp data. For example:

```
sql-> SELECT id, firstname, lastname,
    year(lastLogin) AS Year,
    month(lastLogin) AS Month,
    day(lastLogin) AS Day,
    hour(lastLogin) AS Hour,
    minute(lastLogin) AS Minute
FROM Persons;
 +----+-----------+----------+------+-------+-----+------+--------+
 | id | firstname | lastname | Year | Month | Day | Hour | Minute |
 +----+-----------+----------+------+-------+-----+------+--------+
 |  3 | John      | Morgan   | 2016 |    11 |  29 |    8 |     21 |
 |  2 | John      | Anderson | 2016 |    11 |  28 |   13 |      1 |
 |  4 | Peter     | Smith    | 2016 |    10 |  19 |    9 |     18 |
 |  5 | Dana      | Scully   | 2016 |    11 |   8 |    9 |     16 |
 |  1 | David     | Morrison | 2016 |    10 |  29 |   18 |     43 |
 +----+-----------+----------+------+-------+-----+------+--------+
```

Alternatively, use the EXTRACT function:

```
sql-> SELECT id, firstname, lastname,
    EXTRACT(YEAR FROM lastLogin) AS Year,
    EXTRACT(MONTH FROM lastLogin) AS Month,
    EXTRACT(DAY FROM lastLogin) AS Day,
    EXTRACT(HOUR FROM lastLogin) AS Hour,
    EXTRACT(MINUTE FROM lastLogin) AS Minute
FROM Persons;
 +----+-----------+----------+------+-------+-----+------+--------+
 | id | firstname | lastname | Year | Month | Day | Hour | Minute |
 +----+-----------+----------+------+-------+-----+------+--------+
 |  3 | John      | Morgan   | 2016 |    11 |  29 |    8 |     21 |
 |  4 | Peter     | Smith    | 2016 |    10 |  19 |    9 |     18 |
 |  1 | David     | Morrison | 2016 |    10 |  29 |   18 |     43 |
 |  2 | John      | Anderson | 2016 |    11 |  28 |   13 |      1 |
 |  5 | Dana      | Scully   | 2016 |    11 |   8 |    9 |     16 |
 +----+-----------+----------+------+-------+-----+------+--------+

5 rows returned
sql->
```

Working With Arrays

You can use slice or filter steps to select elements out of an array. We start with some examples using slice steps.

To select and display the second connection of each person, we use this query:

```
sql-> SELECT lastname, connections[1]
AS connection FROM Persons;
 +----------+------------+
 | lastname | connection |
 +----------+------------+
```

```
 | Scully   |          2 |
 | Smith    |          4 |
 | Morgan   |          2 |
 | Anderson |          2 |
 | Morrison |          2 |
 +----------+------------+

5 rows returned
```

In the example, the slice step `[1]` is applied to the connections array. Since array elements start with 0, 1 selects the second connection value.

You can also use a slice step to select all array elements whose positions are within a range: [low:high], where low and high are expressions to specify the range boundaries. You can omit low and high expressions if you do not require a low or high boundary.

For example, the following query returns the lastname and the first 3 connections of person 5 as strongconnections:

```
sql-> SELECT lastname, [connections[0:2]]
AS strongconnections FROM Persons WHERE id = 5;
 +----------+-------------------+
 | lastname | strongconnections |
 +----------+-------------------+
 | Scully   | 2                 |
 |          | 4                 |
 |          | 1                 |
 +----------+-------------------+

1 row returned
```

In the above query for Person 5, the path expression `connections[0:2]` returns the person's first 3 connections. Here, the range is [0:2], so 0 is the low expression and 2 is the high. The path expression returns its result as a list of 3 items. The list is converted to an array (a single item) by enclosing the path expression in an array-constructor expression ([]). The array constructor creates a new array containing the three connections. Notice that although the query shell displays the elements of this constructed array vertically, the number of rows returned by this query is 1.

Use of the array constructor in the select clause is optional. If no array constructor is used, an array will still be constructed, but only if the select-clause expression does indeed return more than one item. If exactly one item is returned, the result will contain just that one item. If the expression returns nothing (an empty result), NULL is used as the result. This behavior is illustrated in the next example, which we will run with and without an array constructor.

As mentioned above, you can omit the low or high expression when specifying the range for a slice step. For example the following query specifies a range of [3:] which returns all connections after the third one. Notice that for persons having only 3 connections or less, an empty array is constructed and returned due to the use of the array constructor.

To fully illustrate this behavior, we display this output in `mode JSON` because the COLUMN mode does not differentiate between a single item and an array containing a single item.

```
sql-> mode JSON
Query output mode is JSON
sql-> SELECT id, [connections[3:]] AS weakConnections FROM Persons;
```

```
{"id":3,"weakConnections":[]}
{"id":4,"weakConnections":[2]}
{"id":2,"weakConnections":[]}
{"id":5,"weakConnections":[3]}
{"id":1,"weakConnections":[]}

5 rows returned
```

Now we run the same query, but without the array constructor. Notice how single items are not contained in an array, and for rows with no match, NULL is returned instead of an empty array.

```
sql-> SELECT id, connections[3:] AS weakConnections FROM Persons;
{"id":2,"weakConnections":null}
{"id":3,"weakConnections":null}
{"id":4,"weakConnections":2}
{"id":5,"weakConnections":3}
{"id":1,"weakConnections":null}

5 rows returned
sql-> mode COLUMN
Query output mode is COLUMN
sql->
```

As a last example of slice steps, the following query returns the last 3 connections of each person. In this query, the slice step is `[size($)-3:]`. In this expression, the $ is an implicitly declared variable that references the array that the slice step is applied to. In this example, $ references the connections array. The size() built-in function returns the size (number of elements) of the input array. So, in this example, size($) is the size of the current connections array. Finally, size($)-3 computes the third position from the end of the current connections array.

```
sql-> SELECT id, [connections[size($)-3:]]
AS weakConnections FROM Persons;
 +----+-------------------+
 | id | weakConnections   |
 +----+-------------------+
 |  5 | 4                 |
 |    | 1                 |
 |    | 3                 |
 +----+-------------------+
 |  4 | 5                 |
 |    | 1                 |
 |    | 2                 |
 +----+-------------------+
 |  3 | 1                 |
 |    | 4                 |
 |    | 2                 |
 +----+-------------------+
 |  2 | 1                 |
 |    | 3                 |
 +----+-------------------+
 |  1 | 2                 |
 |    | 3                 |
 +----+-------------------+
```

```
5 rows returned
```

We now turn our attention to filter steps on arrays. Like slice steps, filter steps also use the square brackets ([]) syntax. However, what goes inside the [] is different. With filter steps there is either nothing inside the [] or a single expression that acts as a condition (returns a boolean result). In the former case, all the elements of the array are selected (the array is "unnested"). In the latter case, the condition is applied to each element in turn, and if the result is true, the element is selected, otherwise it is skipped. For example:

The following query returns the id and connections of persons who are connected to person 4:

```
sql-> SELECT id, connections
FROM Persons p WHERE p.connections[] =any 4;
 +----+-------------+
 | id | connections |
 +----+-------------+
 |  3 | 1           |
 |    | 4           |
 |    | 2           |
 +----+-------------+
 |  5 | 2           |
 |    | 4           |
 |    | 1           |
 |    | 3           |
 +----+-------------+

2 rows returned
```

In the above query, the expression `p.connections[]` returns all the connections of a person. Then, the =any operator returns true if this sequence of connections contains the number 4.

The following query returns the id and connections of persons who are connected with any person having an id greater than 4:

```
sql-> SELECT id, connections FROM Persons p
WHERE p.connections[] >any 4;
 +----+-------------+
 | id | connections |
 +----+-------------+
 |  4 | 3           |
 |    | 5           |
 |    | 1           |
 |    | 2           |
 +----+-------------+

1 row returned
```

The following query returns, for each person, the person's last name and the phone numbers with area code 339:

```
sql-> SELECT lastname,
[ p.address.phones[$element.areacode = 339].number ]
AS phoneNumbers FROM Persons p;
 +----------+--------------+
```

```
 | lastname | phoneNumbers |
 +----------+--------------+
 | Scully   | 3414578      |
 +----------+--------------+
 | Smith    | 4120211      |
 |          | 8694021      |
 |          | 1205678      |
 +----------+--------------+
 | Morgan   |              |
 +----------+--------------+
 | Anderson | 1684972      |
 +----------+--------------+
 | Morrison |              |
 +----------+--------------+

5 rows returned
```

In the above query, the filter step `[$element.areacode = 339]` is applied to the phones array of each person. The filter step evaluates the condition `$element.areacode = 339` on each element of the array. This condition expression uses the implicitly declared variable $element, which references the current element of the array. An empty array is returned for persons that do not have any phone number in the 339 area code. If we wanted to filter out such persons from the result, we would write the following query:

```
sql-> SELECT lastname,
[ p.address.phones[$element.areacode = 339].number ]
AS phoneNumbers FROM Persons p WHERE p.address.phones.areacode =any 339;
 +----------+--------------+
 | lastname | phoneNumbers |
 +----------+--------------+
 | Scully   | 3414578      |
 +----------+--------------+
 | Smith    | 4120211      |
 |          | 8694021      |
 |          | 1205678      |
 +----------+--------------+
 | Anderson | 1684972      |
 +----------+--------------+

3 rows returned
```

The previous query contains the path expression `p.address.phones.areacode`. In that expression, the field step `.areacode` is applied to an array field (phones). In this case, the field step is applied to each element of the array in turn. In fact, the path expression is equivalent to `p.address.phones[].areacode`.

In addition to the implicitly-declared $ and $element variables, the condition inside a filter step can also use the $pos variable (also implicitly declared). $pos references the position within the array of the current element (the element on which the condition is applied). For example, the following query selects the "interesting" connections of each person, where a connection is considered interesting if it is among the 3 strongest connections and connects to a person with an id greater or equal to 4.

```
sql-> SELECT id, [p.connections[$element >= 4 and $pos < 3]]
AS interestingConnections FROM Persons p;
```

```
 +----+------------------------+
 | id | interestingConnections |
 +----+------------------------+
 |  5 | 4                      |
 +----+------------------------+
 |  4 | 5                      |
 +----+------------------------+
 |  3 | 4                      |
 +----+------------------------+
 |  2 |                        |
 +----+------------------------+
 |  1 |                        |
 +----+------------------------+

5 rows returned
```

Finally, two arrays can be compared with each other using the usual comparison operators (=, != , >, >=, >, and >=). For example the following query constructs the array [1,3] and selects persons whose connections array is equal to [1,3].

```
sql-> SELECT lastname FROM Persons p
WHERE p.connections = [1,3];
 +----------+
 | lastname |
 +----------+
 | Anderson |
 +----------+

1 row returned
```

Working with Records

You can use a field step to select the value of a field from a record. For example, to return the id, last name, and city of persons who reside in Florida:

```
sql-> SELECT id, lastname, p.address.city
FROM Persons p WHERE p.address.state = "FL";
 +----+----------+------------+
 | id | lastname |    city    |
 +----+----------+------------+
 |  3 | Morgan   | Middleburg |
 +----+----------+------------+

1 row returned
```

In the above query, the path expression `p.address.state` consists of 2 field steps: `.address` selects the address field of the current row (rows can be viewed as records, whose fields are the row columns), and `.state` selects the state field of the current address.

The example record contains an array of phone numbers. You can form queries against that array using a combination of path steps and sequence comparison operators. For example, to return the last name of persons who have a phone number with area code 423:

```
sql-> SELECT lastname FROM Persons
p WHERE p.address.phones.areacode =any 423;
 +----------+
 | lastname |
 +----------+
 | Morrison |
 +----------+

1 row returned
```

In the above query, the path expression `p.address.phones.areacode` returns all the area codes of a person. Then, the =any operator returns true if this sequence of area codes contains the number 423. Notice also that the field step `.areacode` is applied to an array field (phones). This is allowed if the array contains records or maps. In this case, the field step is applied to each element of the array in turn.

The following example returns all the persons who had three connections. Notice the use of [] after connections: it is an array filter step, which returns all the elements of the connections array as a sequence (it is unnesting the array).

```
sql-> SELECT id, firstName, lastName, connections from Persons where
connections[] =any 3 ORDER BY id;
 +----+-----------+----------+-------------+
 | id | firstName | lastName | connections |
 +----+-----------+----------+-------------+
 |  1 | David     | Morrison | 2           |
 |    |           |          | 3           |
 +----+-----------+----------+-------------+
 |  2 | John      | Anderson | 1           |
 |    |           |          | 3           |
 +----+-----------+----------+-------------+
 |  4 | Peter     | Smith    | 3           |
 |    |           |          | 5           |
 |    |           |          | 1           |
 |    |           |          | 2           |
 +----+-----------+----------+-------------+
 |  5 | Dana      | Scully   | 2           |
 |    |           |          | 4           |
 |    |           |          | 1           |
 |    |           |          | 3           |
 +----+-----------+----------+-------------+

4 rows returned
```

This query can use ORDER BY to sort the results because the sort is being performed on the table's primary key. The next section shows sorting on non-primary key fields through the use of indexes.

For more examples of querying against data contained in arrays, see Working With Arrays.

Using ORDER BY to Sort Results

To sort the results from a SELECT statement using a field that is not the table's primary key, you must first create an index for the column of choice. For example, for the next table, to query based on a Timestamp and sort the results in descending order by the timestamp, create an index:

```
sql-> SELECT id, firstname, lastname, lastLogin FROM Persons;
 +----+-----------+----------+--------------------------+
 | id | firstname | lastname |        lastLogin         |
 +----+-----------+----------+--------------------------+
 |  3 | John      | Morgan   | 2016-11-29T08:21:35.4971 |
 |  4 | Peter     | Smith    | 2016-10-19T09:18:05.5555 |
 |  2 | John      | Anderson | 2016-11-28T13:01:11.2088 |
 |  5 | Dana      | Scully   | 2016-11-08T09:16:46.3929 |
 |  1 | David     | Morrison | 2016-10-29T18:43:59.8319 |
 +----+-----------+----------+--------------------------+

5 rows returned

sql-> CREATE INDEX tsidx1 on Persons (lastLogin);
Statement completed successfully
sql-> SELECT id, firstname, lastname, lastLogin
FROM Persons ORDER BY lastLogin DESC;
 +----+-----------+----------+--------------------------+
 | id | firstname | lastname |        lastLogin         |
 +----+-----------+----------+--------------------------+
 |  3 | John      | Morgan   | 2016-11-29T08:21:35.4971 |
 |  2 | John      | Anderson | 2016-11-28T13:01:11.2088 |
 |  5 | Dana      | Scully   | 2016-11-08T09:16:46.3929 |
 |  1 | David     | Morrison | 2016-10-29T18:43:59.8319 |
 |  4 | Peter     | Smith    | 2016-10-19T09:18:05.5555 |
 +----+-----------+----------+--------------------------+

5 rows returned
```

SQL for Oracle NoSQL Database can also sort query results by the values of nested records. To do so, create an index of the nested field (or fields). For example, you can create an index of address.state from the Persons table, and then order by state:

```
sql-> CREATE INDEX indx1 on Persons (address.state);
Statement completed successfully
sql-> SELECT id, $p.address.state FROM
Persons $p ORDER BY $p.address.state;
 +----+-------+
 | id | state |
 +----+-------+
 |  3 | FL    |
 |  4 | MA    |
 |  5 | NJ    |
 |  1 | TN    |
 |  2 | WI    |
 +----+-------+
```

```
5 rows returned
```

To learn more about indexes, see Working With Indexes.

Working With Maps

The path steps applicable to maps are field and filter steps. Slice steps do not make sense for maps, because maps are unordered, and as a result, their entries do not have any fixed positions.

You can use a field step to select the value of a field from a map. For example, to return the lastname and the food expenses of all persons:

```
sql-> SELECT lastname, p.expenses.food
FROM Persons p;
 +----------+------+
 | lastname | food |
 +----------+------+
 | Morgan   | 2000 |
 | Morrison | 1000 |
 | Scully   |  900 |
 | Smith    | 6000 |
 | Anderson | 1700 |
 +----------+------+

5 rows returned
```

In the above query, the path expression `p.expenses.food` consists of 2 field steps: `.expenses` selects the expenses field of the current row and `.food` selects the value of the food field/entry from the current expenses map.

To return the lastname and amount spent on travel for each person who spent less than $3000 on food:

```
sql-> SELECT lastname, p.expenses.travel
FROM Persons p WHERE p.expenses.food < 3000;
 +----------+--------+
 | lastname | travel |
 +----------+--------+
 | Scully   |   NULL |
 | Morgan   |    700 |
 | Anderson |   2100 |
 | Morrison |   NULL |
 +----------+--------+

4 rows returned
```

Notice that NULL is returned for persons who did not have any travel expenses.

Filter steps are performed using either the `.values()` or `.keys()` path steps. To select values of map entries, use `.values(<cond>)`. To select keys of map entries, use `.keys(<cond>)`. If no condition is used in these steps, all the values or keys of the input map are selected. If the

steps do contain a condition expression, the condition is evaluated for each entry, and the value or key of the entry is selected/skipped if the result is true/false.

The implicitly-declared variables $key and $value can be used inside a map filter condition. $key references the key of the current entry and $value references the associated value. Notice that, contrary to arrays, the $pos variable can not be be used inside map filters (because map entries do not have fixed positions).

To show, for each user, their id and the expense categories where they spent more than $1000:

```
sql-> SELECT id, p.expenses.keys($value > 1000) as Expenses
from Persons p;
 +----+---------------------+
 | id |      Expenses       |
 +----+---------------------+
 |  4 | clothes             |
 |    | food                |
 |    | shoes               |
 +----+---------------------+
 |  3 | food                |
 +----+---------------------+
 |  2 | food                |
 |    | travel              |
 +----+---------------------+
 |  5 | clothes             |
 +----+---------------------+
 |  1 | NULL                |
 +----+---------------------+
```

To return the id and the expense categories in which the user spent more than they spent on clothes, use the following filter step expression. In this query, the context-item variable ($) appearing in the filter step expression [$value > $.clothes] refers to the expenses map as a whole.

```
sql-> SELECT id, p.expenses.keys($value > $.clothes) FROM Persons p;
 +----+---------------------+
 | id |      Column_2       |
 +----+---------------------+
 |  3 | NULL                |
 +----+---------------------+
 |  2 | NULL                |
 +----+---------------------+
 |  5 | NULL                |
 +----+---------------------+
 |  1 | NULL                |
 +----+---------------------+
 |  4 | food                |
 +----+---------------------+
```

To return the id and expenses data of any person who spent more on any category than what they spent on food:

```
sql-> SELECT id, p.expenses
FROM Persons p
```

```
WHERE p.expenses.values() >any p.expenses.food;
 +----+---------------------+
 | id |      expenses       |
 +----+---------------------+
 |  5 | clothes  | 1500     |
 |    | food     | 900      |
 |    | shoes    | 1000     |
 +----+---------------------+
 |  2 | books    | 100      |
 |    | food     | 1700     |
 |    | travel   | 2100     |
 +----+---------------------+

2 rows returned
```

To return the id of all persons who consumed more than $2000 in any category other than food:

```
sql-> SELECT id FROM Persons p
WHERE p.expenses.values($key != "food") >any 2000;
 +----+
 | id |
 +----+
 |  2 |
 +----+

1 row returned
```

Using the size() Function

The size function can be used to return the size (number of fields/entries) of a complex item (record, array, or map). For example:

To return the id and the number of phones that each person has:

```
sql-> SELECT id, size(p.address.phones)
AS registeredphones FROM Persons p;
 +----+------------------+
 | id | registeredphones |
 +----+------------------+
 |  5 |                3 |
 |  3 |                2 |
 |  4 |                4 |
 |  2 |                1 |
 |  1 |                1 |
 +----+------------------+

5 rows returned
```

To return the id and the number of expenses categories for each person: has:

```
sql-> SELECT id, size(p.expenses) AS
categories FROM Persons p;
 +----+------------+
```

```
 | id | categories |
 +----+------------+
 |  4 |          4 |
 |  3 |          3 |
 |  2 |          3 |
 |  1 |          2 |
 |  5 |          3 |
 +----+------------+

5 rows returned
```

To return for each person their id and the number of expenses categories for which the expenses were more than 2000:

```
sql-> SELECT id, size([p.expenses.values($value > 2000)]) AS
expensiveCategories FROM Persons p;
 +----+---------------------+
 | id | expensiveCategories |
 +----+---------------------+
 |  3 |                   0 |
 |  2 |                   1 |
 |  5 |                   0 |
 |  1 |                   0 |
 |  4 |                   1 |
 +----+---------------------+

5 rows returned
```

5

Working with JSON

This chapter provides examples on working with JSON data. If you want to follow along with the examples, get the `Examples` download from here and run the `SQLJSONExamples` script found in the `sql`folder. This creates the table and imports the data used.

JSON data is written to JSON data columns by providing a JSON object. This object can contain any valid JSON data. The input data is parsed and stored internally as Oracle NoSQL Database datatypes:

- When numbers are encountered, they are converted to integer, long, or double items, depending on the actual value of the number (float items are not used for JSON).
- Strings in the input text are mapped to string items.
- Boolean values are mapped to boolean items.
- JSON nulls are mapped to JSON null items.
- When an array is encountered in the input text, an array item is created whose type is `Array(JSON)`. This is done unconditionally, no matter what the actual contents of the array might be.
- When a JSON object is encountered in the input text, a map item is created whose type is `Map(JSON)`, unconditionally.

> **Note:**
>
> There is no JSON equivalent to the TIMESTAMP datatype, so if input text contains a string in the TIMESTAMP format it is simply stored as a string item in the JSON column.

The remainder of this chapter provides an overview to querying JSON data.

SQLJSONExamples Script

The SQLJSONExample is available to illustrate JSON usage. This script creates the following table:

```
create table if not exists JSONPersons (
  id integer,
  person JSON,
  primary key (id)
);
```

The script imports the following table rows. Notice that the content for the `person` column, which is of type `JSON` contains a JSON object. That object contains a series of fields which

represent our person. We have deliberately included inconsistent information in this example so as to illustrate how to handle various queries when working with JSON data.

```
{
  "id":1,
  "person" : {
      "firstname":"David",
      "lastname":"Morrison",
      "age":25,
      "income":100000,
      "lastLogin" : "2016-10-29T18:43:59.8319",
      "address":{"street":"150 Route 2",
                 "city":"Antioch",
                 "state":"TN",
                 "zipcode" : 37013,
                 "phones":[{"type":"home", "areacode":423,
                            "number":8634379}]
                },
      "connections":[2, 3],
      "expenses":{"food":1000, "gas":180}
  }
}

{
  "id":2,
  "person" : {
      "firstname":"John",
      "lastname":"Anderson",
      "age":35,
      "income":100000,
      "lastLogin" : "2016-11-28T13:01:11.2088",
      "address":{"street":"187 Hill Street",
                 "city":"Beloit",
                 "state":"WI",
                 "zipcode" : 53511,
                 "phones":[{"type":"home", "areacode":339,
                            "number":1684972}]
                },
      "connections":[1, 3],
      "expenses":{"books":100, "food":1700, "travel":2100}
  }
}

{
  "id":3,
  "person" : {
      "firstname":"John",
      "lastname":"Morgan",
      "age":38,
      "income":100000000,
      "lastLogin" : "2016-11-29T08:21:35.4971",
      "address":{"street":"187 Aspen Drive",
                 "city":"Middleburg",
                 "state":"FL",
                 "phones":[{"type":"work", "areacode":305,
                            "number":1234079},
```

```
                            {"type":"home", "areacode":305,
                             "number":2066401}
                           ]
                },
      "connections":[1, 4, 2],
      "expenses":{"food":2000, "travel":700, "gas":10}
  }
}
{
  "id":4,
  "person": {
      "firstname":"Peter",
      "lastname":"Smith",
      "age":38,
      "income":80000,
      "lastLogin" : "2016-10-19T09:18:05.5555",
      "address":{"street":"364 Mulberry Street",
                 "city":"Leominster",
                 "state":"MA",
                 "phones":[{"type":"work", "areacode":339,
                            "number":4120211},
                           {"type":"work", "areacode":339,
                            "number":8694021},
                           {"type":"home", "areacode":339,
                            "number":1205678},
                            null,
                           {"type":"home", "areacode":305,
                            "number":8064321}
                           ]
                },
      "connections":[3, 5, 1, 2],
      "expenses":{"food":6000, "books":240, "clothes":2000,
                  "shoes":1200}
  }
}

{
  "id":5,
  "person" : {
      "firstname":"Dana",
      "lastname":"Scully",
      "age":47,
      "income":400000,
      "lastLogin" : "2016-11-08T09:16:46.3929",
      "address":{"street":"427 Linden Avenue",
                 "city":"Monroe Township",
                 "state":"NJ",
                 "phones":[{"type":"work", "areacode":201,
                            "number":3213267},
                           {"type":"work", "areacode":201,
                            "number":8765421},
                           {"type":"home", "areacode":339,
                            "number":3414578}
                           ]
                },
      "connections":[2, 4, 1, 3],
```

```
      "expenses":{"food":900, "shoes":1000, "clothes":1500}
  }
}


{
  "id":6,
  "person" : {
      "mynumber":5,
      "myarray":[1,2,3,4]
  }
}

{
  "id":7,
  "person" : {
      "mynumber":"5",
      "myarray":["1","2","3","4"]
  }
}
```

You run the SQLJSONExamples script using the load command:

```
> cd <installdir>/examples/sql
> java -jar <KVHOME>/lib/sql.jar -helper-hosts <host>:<port> \
-store <storename> load \
-file <KVHOME>/examples/sql/SQLJSONExamples.cli
```

Basic Queries

Because JSON is parsed and stored internally in native data formats with Oracle NoSQL Database, querying JSON data is no different than querying data in other column types. See Simple SELECT Queries and Working with complex data for introductory examples of how to form these queries.

In our JSONPersons example, all of the data for each person is contained in a column of type JSON called `person`. This data is presented as a JSON object, and mapped internally into a Map(JSON) type. You can query information in this column as you would query a Map of any other type. For example:

```
sql-> SELECT id, j.person.lastname, j.person.age FROM JSONPersons j;
 +----+----------------------+------------+
 | id |      lastname        |    age     |
 +----+----------------------+------------+
 |  3 | Morgan               | 38         |
 +----+----------------------+------------+
 |  2 | Anderson             | 35         |
 +----+----------------------+------------+
 |  5 | Scully               | 47         |
 +----+----------------------+------------+
 |  1 | Morrison             | 25         |
 +----+----------------------+------------+
 |  4 | Smith                | 38         |
 +----+----------------------+------------+
```

```
 |  6 | NULL                | NULL       |
 +----+---------------------+------------+
 |  7 | NULL                | NULL       |
 +----+---------------------+------------+

7 rows returned
```

The last two rows in returned from this query contain all NULLs. This is because those rows were populated using JSON objects that are different than the objects used to populate the rest of the table. This capability of JSON is both a strength and a weakness. As a plus, you can modify your schema easily. However, if you are not careful, you can end up with tables containing dissimilar data in both large and small ways.

Because the JSON object is stored as a map, you can use normal map step functions on the column. For example:

```
sql-> SELECT id, j.person.expenses.keys($value > 1000) as Expenses
from JSONPersons j;
+----+---------------------+
 | id |      Expenses       |
 +----+---------------------+
 |  3 | food                |
 +----+---------------------+
 |  2 | food                |
 |    | travel              |
 +----+---------------------+
 |  4 | clothes             |
 |    | food                |
 |    | shoes               |
 +----+---------------------+
 |  6 | NULL                |
 +----+---------------------+
 |  5 | clothes             |
 +----+---------------------+
 |  7 | NULL                |
 +----+---------------------+
 |  1 | NULL                |
 +----+---------------------+

7 rows returned
```

Here, id 1 is NULL because that user had no expenses greater than $1000, while id 6 and 7 are NULL because they have no `j.person.expenses` field.

Using WHERE EXISTS with JSON

As we saw in the previous section, different rows in the same table can have dissimilar information in them when a column type is JSON. To identify whether desired information exists for a given JSON column, use the `EXISTS` operator.

For example, some of the JSON persons have a zip code entered for their address, and others do not. Use this query to see all the users with a zipcode:

```
sql-> SELECT id, j.person.address AS Address FROM JSONPersons j
WHERE EXISTS j.person.address.zipcode;
```

```
 +----+--------------------------------+
 | id |            Address             |
 +----+--------------------------------+
 |  2 | city         | Beloit          |
 |    | phones                         |
 |    |     areacode | 339             |
 |    |     number   | 1684972         |
 |    |     type     | home            |
 |    | state        | WI              |
 |    | street       | 187 Hill Street |
 |    | zipcode      | 53511           |
 +----+--------------------------------+
 |  1 | city         | Antioch         |
 |    | phones                         |
 |    |     areacode | 423             |
 |    |     number   | 8634379         |
 |    |     type     | home            |
 |    | state        | TN              |
 |    | street       | 150 Route 2     |
 |    | zipcode      | 37013           |
 +----+--------------------------------+

2 rows returned
```

When querying data for inconsistencies, it is often more useful to see all rows where information is missing by using `WHERE NOT EXISTS`:

```
sql-> SELECT * FROM JSONPersons j WHERE NOT EXISTS j.person.lastname;
 +----+-------------------+
 | id |      person       |
 +----+-------------------+
 |  7 | myarray           |
 |    |            1      |
 |    |            2      |
 |    |            3      |
 |    |            4      |
 |    | mynumber | 5      |
 +----+-------------------+
 |  6 | myarray           |
 |    |            1      |
 |    |            2      |
 |    |            3      |
 |    |            4      |
 |    | mynumber | 5      |
 +----+-------------------+


1 row returned
```

Seeking NULLS in Arrays

All arrays found in a JSON input stream are stored internally as ARRAY(JSON). This means that it is possible for the array to have inconsistent types for its members.

In our example, the phones array for user id 4 contains a null element:

```
sql-> SELECT j.person.address.phones FROM JSONPersons j WHERE j.id=4;
 +--------------------+
 |       phones       |
 +--------------------+
 | areacode | 339     |
 | number   | 4120211 |
 | type     | work    |
 |                    |
 | areacode | 339     |
 | number   | 8694021 |
 | type     | work    |
 |                    |
 | areacode | 339     |
 | number   | 1205678 |
 | type     | home    |
 | null               |
 |                    |
 | areacode | 305     |
 | number   | 8064321 |
 | type     | home    |
 +--------------------+
```

A way to discover this in your table is to examine the phones array for null values:

```
sql-> SELECT id, j.person.address.phones FROM JSONPersons j
WHERE j.person.address.phones[] =any null;
 +----+--------------------+
 | id |       phones       |
 +----+--------------------+
 |  4 | areacode | 339     |
 |    | number   | 4120211 |
 |    | type     | work    |
 |    |                    |
 |    | areacode | 339     |
 |    | number   | 8694021 |
 |    | type     | work    |
 |    |                    |
 |    | areacode | 339     |
 |    | number   | 1205678 |
 |    | type     | home    |
 |    | null               |
 |    |                    |
 |    | areacode | 305     |
 |    | number   | 8064321 |
 |    | type     | home    |
 +----+--------------------+

1 row returned
```

Notice the use of the array filter step ([]) in the previous query. This is needed to unpack the array into a sequence so that the `=any` comparison operator can be used with it.

Examining Data Types JSON Columns

The example data contains a couple of rows with unusual data:

```
{
  "id":6,
  "person" : {
      "mynumber":5,
      "myarray":[1,2,3,4]
  }
}

{
  "id":7,
  "person" : {
      "mynumber":"5",
      "myarray":["1","2","3","4"]
  }
}
```

You can locate them using the query:

```
sql-> SELECT * FROM JSONPersons j WHERE EXISTS j.person.mynumber;
 +----+--------------------+
 | id |      person        |
 +----+--------------------+
 |  6 | myarray            |
 |    |            1       |
 |    |            2       |
 |    |            3       |
 |    |            4       |
 |    | mynumber | 5       |
 +----+--------------------+
 |  7 | myarray            |
 |    |            1       |
 |    |            2       |
 |    |            3       |
 |    |            4       |
 |    | mynumber | 5       |
 +----+--------------------+

2 rows returned
```

However, notice that these two rows actually contain numbers stored as different types. ID 6 stores integers while ID 7 stores strings. You can select a row based on its type:

```
sql-> SELECT * FROM JSONPersons j
WHERE j.person.mynumber IS OF TYPE (integer);
 +----+--------------------+
 | id |      person        |
 +----+--------------------+
 |  6 | myarray            |
 |    |            1       |
```

```
 |    |            2      |
 |    |            3      |
 |    |            4      |
 |    | mynumber | 5      |
 +----+------------------+
```

Notice that if you use `IS NOT OF TYPE` then every row in the table is returned except id 6. This is because for all the other rows, `j.person.mynumber` evaluates to jnull, which is not an integer.

```
sql-> SELECT id FROM JSONPersons j
WHERE j.person.mynumber IS NOT OF TYPE (integer);
 +----+
 | id |
 +----+
 |  3 |
 |  2 |
 |  5 |
 |  4 |
 |  1 |
 |  7 |
 +----+

6 rows returned
```

To solve this problem, also check for the existence of `j.person.mynumber`:

```
sql-> SELECT id from JSONPersons j WHERE EXISTS j.person.mynumber
and j.person.mynumber IS NOT OF TYPE (integer);
 +----+
 | id |
 +----+
 |  7 |
 +----+

1 row returned
```

You can also perform type checking based on the type of data contained in the array. Recall that our rows contain arrays with integers and arrays with strings. You can return the row with just the array of strings using:

```
sql-> SELECT id, j.person.myarray FROM JSONPersons j
WHERE j.person.myarray[] IS OF TYPE (string+);
 +----+------------------+
 | id |      myarray     |
 +----+------------------+
 |  7 | 1                |
 |    | 2                |
 |    | 3                |
 |    | 4                |
 +----+------------------+
```

```
1 row returned
```

Here, we use the array filter step ([]) in the WHERE clause to unpack the array into a sequence. This allows is-of-type to iterate over the sequence, checking the type of each element. If every element in the sequence matches the identified type (`string`, in this case), then the is-of-type returns true.

Also notice that the query uses the `+` cardinality modifier. This means that is-of-type will return true only if the input sequence (myarray[], in this case) contains ONE OR MORE elements that match the identified type (string). If we used `*`, then 0 or more elements would have to match the identified type in order for true to return. Because our table contains a mix of rows with different schema, the result is that every row except id 6 is returned:

```
sql-> SELECT id, j.person.myarray FROM JSONPersons j
WHERE j.person.myarray[] IS OF TYPE (string*);
 +----+-------------------+
 | id |      myarray      |
 +----+-------------------+
 |  3 | NULL              |
 +----+-------------------+
 |  5 | NULL              |
 +----+-------------------+
 |  1 | NULL              |
 +----+-------------------+
 |  7 | 1                 |
 |    | 2                 |
 |    | 3                 |
 |    | 4                 |
 +----+-------------------+
 |  4 | NULL              |
 +----+-------------------+
 |  2 | NULL              |
 +----+-------------------+

6 rows returned
```

Finally, if we do not provide a cardinality modifier at all, then is-of-type returns true if ONE AND ONLY one member of the input sequence matches the identified type. In this example, the result is that no rows are returned.

```
sql-> SELECT id, j.person.myarray FROM JSONPersons j
WHERE j.person.myarray[] IS OF TYPE (string);

0 row returned
```

Using Map Steps with JSON Data

On import, Oracle NoSQL Database stores JSON objects as MAP(JSON). This means you can use map filter steps with your JSON objects.

For example, if you want to visually examine the JSON fields in use by your rows:

```
sql-> SELECT id, j.person.keys() FROM JSONPersons j;
 +----+------------------------+
 | id |        Column_2        |
 +----+------------------------+
 |  4 | address                |
 |    | age                    |
 |    | connections            |
 |    | expenses               |
 |    | firstname              |
 |    | income                 |
 |    | lastLogin              |
 |    | lastname               |
 +----+------------------------+
 |  6 | myarray                |
 |    | mynumber               |
 +----+------------------------+
 |  3 | address                |
 |    | age                    |
 |    | connections            |
 |    | expenses               |
 |    | firstname              |
 |    | income                 |
 |    | lastLogin              |
 |    | lastname               |
 +----+------------------------+
 |  5 | address                |
 |    | age                    |
 |    | connections            |
 |    | expenses               |
 |    | firstname              |
 |    | income                 |
 |    | lastLogin              |
 |    | lastname               |
 +----+------------------------+
 |  1 | address                |
 |    | age                    |
 |    | connections            |
 |    | expenses               |
 |    | firstname              |
 |    | income                 |
 |    | lastLogin              |
 |    | lastname               |
 +----+------------------------+
 |  7 | myarray                |
 |    | mynumber               |
 +----+------------------------+
 |  2 | address                |
 |    | age                    |
 |    | connections            |
 |    | expenses               |
 |    | firstname              |
 |    | income                 |
 |    | lastLogin              |
 |    | lastname               |
```

```
 +----+------------------------+

7 rows returned
```

Casting Datatypes

You can cast one data type to another using the `cast` expression.

In JSON, casting is particularly useful for timestamp information because JSON has no equivalent to the Oracle NoSQL Database Timestamp data type. Instead, the timestamp information is carried in a JSON object as a string. To work with it as a Timestamp, use `cast`.

In Working with Timestamps we showed how to work with the timestamp data type. In this case, what you do is no different except you must cast both sides of the expression. Also, because the left side of the expression is a sequence, you must specify a type quantifier (`*` in this case):

```
sql-> SELECT id,
             j.person.firstname, j.person.lastname, j.person.lastLogin
             FROM JSONPersons j
             WHERE CAST(j.person.lastLogin AS TIMESTAMP*) >
             CAST("2016-11-01" AS TIMESTAMP) AND
             CAST(j.person.lastLogin AS TIMESTAMP*) <
             CAST("2016-11-30" AS TIMESTAMP);
 +----+------------+--------------+----------------------------+
 | id | firstname  |   lastname   |       lastLogin            |
 +----+------------+--------------+----------------------------+
 |  3 | John       | Morgan       | 2016-11-29T08:21:35.4971   |
 +----+------------+--------------+----------------------------+
 |  2 | John       | Anderson     | 2016-11-28T13:01:11.2088   |
 +----+------------+--------------+----------------------------+
 |  5 | Dana       | Scully       | 2016-11-08T09:16:46.3929   |
 +----+------------+--------------+----------------------------+

3 rows returned
```

As another example, you can cast to an integer and then operate on that number:

```
sql-> SELECT id, j.person.mynumber,
             CAST(j.person.mynumber as integer) * 10 AS TenTimes
             FROM JSONPersons j WHERE EXISTS j.person.mynumber;
 +----+---------------------+----------+
 | id |      mynumber       | TenTimes |
 +----+---------------------+----------+
 |  7 | 5                   |       50 |
 +----+---------------------+----------+
 |  6 | 5                   |       50 |
 +----+---------------------+----------+
```

If you want to operate on just the row that contains the number as a string, use IS OF TYPE:

```
sql-> SELECT id, j.person.mynumber,
             CAST(j.person.mynumber as integer) * 10 AS TenTimes
```

```
            FROM JSONPersons j WHERE EXISTS j.person.mynumber
            AND j.person.mynumber IS OF TYPE (string);
 +----+----------------------+-----------+
 | id |      mynumber        | TenTimes |
 +----+----------------------+-----------+
 |  7 | 5                    |       50 |
 +----+----------------------+-----------+
```

Using Searched Case

A searched case expression can be helpful in identifying specific problems with the JSON data in your JSON columns. The example data we have been using in this chapter sometimes provides a JSONPersons.address field, and sometimes it does not. When an address is present, sometimes it provides a zipcode, and sometimes it does not. We can use a searched case expression to identify and describe the specific problem with each row.

```
sql->  SELECT id,
CASE
    WHEN NOT EXISTS j.person.address
    THEN j.person.keys()
    WHEN NOT EXISTS j.person.address.zipcode
    THEN "No Zipcode"
    ELSE j.person.address.zipcode
END
FROM JSONPersons j;
 +----+-----------------------+
 | id |       Column_2        |
 +----+-----------------------+
 |  4 | No Zipcode            |
 +----+-----------------------+
 |  3 | No Zipcode            |
 +----+-----------------------+
 |  5 | No Zipcode            |
 +----+-----------------------+
 |  1 | 37013                 |
 +----+-----------------------+
 |  7 | myarray               |
 |    | mynumber              |
 +----+-----------------------+
 |  6 | myarray               |
 |    | mynumber              |
 +----+-----------------------+
 |  2 | 53511                 |
 +----+-----------------------+

7 rows returned
```

We can improve the report by adding a third column that uses a second searched case expression:

```
sql-> SELECT id,
CASE
    WHEN NOT EXISTS j.person.address
    THEN "No Address"
```

```
    WHEN NOT EXISTS j.person.address.zipcode
    THEN "No Zipcode"
    ELSE j.person.address.zipcode
END,
CASE
    WHEN NOT EXISTS j.person.address
    THEN j.person.keys()
    ELSE j.person.address
END
FROM JSONPersons j;
```

id	Column_2	Column_3	
3	No Zipcode	city	Middleburg
		phones	
			areacode: 305
			number: 1234079
			type: work
			areacode: 305
			number: 2066401
			type: home
		state	FL
		street	187 Aspen Drive
2	53511	city	Beloit
		phones	
			areacode: 339
			number: 1684972
			type: home
		state	WI
		street	187 Hill Street
		zipcode	53511
5	No Zipcode	city	Monroe Township
		phones	
			areacode: 201
			number: 3213267
			type: work
			areacode: 201
			number: 8765421
			type: work
			areacode: 339
			number: 3414578
			type: home
		state	NJ
		street	427 Linden Avenue
1	37013	city	Antioch
		phones	
			areacode: 423
			number: 8634379
			type: home
		state	TN

```
 |    |                       | street       | 150 Route 2          |
 |    |                       | zipcode      | 37013                |
 +----+-----------------------+-----------------------------------+
 |  7 | No Address            | myarray                           |
 |    |                       | mynumber                          |
 +----+-----------------------+-----------------------------------+
 |  4 | No Zipcode            | city         | Leominster           |
 |    |                       | phones                            |
 |    |                       |     areacode | 339                  |
 |    |                       |     number   | 4120211              |
 |    |                       |     type     | work                 |
 |    |                       |                                   |
 |    |                       |     areacode | 339                  |
 |    |                       |     number   | 8694021              |
 |    |                       |     type     | work                 |
 |    |                       |                                   |
 |    |                       |     areacode | 339                  |
 |    |                       |     number   | 1205678              |
 |    |                       |     type     | home                 |
 |    |                       |                null                |
 |    |                       |                                   |
 |    |                       |     areacode | 305                  |
 |    |                       |     number   | 8064321              |
 |    |                       |     type     | home                 |
 |    |                       | state        | MA                   |
 |    |                       | street       | 364 Mulberry Street  |
 +----+-----------------------+-----------------------------------+
 |  6 | No Address            | myarray                           |
 |    |                       | mynumber                          |
 +----+-----------------------+-----------------------------------+

7 rows returned
```

Finally, it is possible to nest search case expressions. Our sample data also has a spurious null in the phones array (see id 4). We can report that in the following way (output is modified slightly to fit in the space allowed):

```
sql-> SELECT id,
CASE
    WHEN EXISTS j.person.address
    THEN
       CASE
          WHEN EXISTS j.person.address.zipcode
          THEN
             CASE
                WHEN j.person.address.phones[] =any null
                THEN "Zipcode exists but null in the phones array"
                ELSE j.person.address.zipcode
             END
          WHEN j.person.address.phones[] =any null
          THEN "No zipcode and null in phones array"
          ELSE "No zipcode"
       END
    ELSE "No Address"
END,
```

```
CASE
    WHEN NOT EXISTS j.person.address
    THEN j.person.keys()
    ELSE j.person.address
END
FROM JSONPersons j;
```

id	Column_2	Column_3	
3	No zipcode	city	Middleburg
		phones	
		areacode	305
		number	1234079
		type	work
		areacode	305
		number	2066401
		type	home
		state	FL
		street	187 Aspen Drive
2	53511	city	Beloit
		phones	
		areacode	339
		number	1684972
		type	home
		state	WI
		street	187 Hill Street
		zipcode	53511
5	No zipcode	city	Monroe Township
		phones	
		areacode	201
		number	3213267
		type	work
		areacode	201
		number	8765421
		type	work
		areacode	339
		number	3414578
		type	home
		state	NJ
		street	427 Linden Avenue
1	37013	city	Antioch
		phones	
		areacode	423
		number	8634379
		type	home
		state	TN
		street	150 Route 2
		zipcode	37013
7	No Address	myarray	

```
|    |                         | mynumber                            |
+----+-------------------------+-------------------------------------+
|  4 | No zipcode and null     | city          | Leominster          |
|    | in phones array         | phones                              |
|    |                         |     areacode | 339                  |
|    |                         |     number   | 4120211              |
|    |                         |     type     | work                 |
|    |                         |              |                      |
|    |                         |     areacode | 339                  |
|    |                         |     number   | 8694021              |
|    |                         |     type     | work                 |
|    |                         |              |                      |
|    |                         |     areacode | 339                  |
|    |                         |     number   | 1205678              |
|    |                         |     type     | home                 |
|    |                         |                null                 |
|    |                         |              |                      |
|    |                         |     areacode | 305                  |
|    |                         |     number   | 8064321              |
|    |                         |     type     | home                 |
|    |                         | state        | MA                   |
|    |                         | street       | 364 Mulberry Street  |
+----+-------------------------+-------------------------------------+
|  6 | No Address              | myarray                             |
|    |                         | mynumber                            |
+----+-------------------------+-------------------------------------+
```

```
7 rows returned
```

6
Working With GeoJSON Data

The GeoJSON specification (https://tools.ietf.org/html/rfc7946) defines the structure and content of JSON objects representing geographical shapes on earth (called geometries). Oracle NoSQL Database implements several functions that interpret JSON geometry objects. The functions also let you search table rows containing geometries that satisfy certain conditions. Search is made efficient through the use of special indexes, as described in the *SQL Reference Guide*.

> **Note:**
>
> Support for GeoJson data is available only in the Oracle NoSQL Database Enterprise Edition.

Geodetic Coordinates

As described, all kinds of geometries are specified in terms of a set of positions. However, for line strings and polygons, the actual geometrical shape is formed by lines connecting their positions. The GeoJSON specification defines a line between two points as the straight line that connects the points in the (flat) cartesian coordinate system, whose horizontal and vertical axes are the longitude and latitude, respectively. More precisely, the coordinates of every point on a line that does not cross the antimeridian between a point P1 = (lon1, lat1) and P2 = (lon2, lat2) can be calculated as:

```
P = (lon, lat) = (lon1 + (lon2 - lon1) * t, lat1 + (lat2 - lat1) * t)
```

with `t` being a real number, greater than or equal to 0, and less than or equal to 1.

Unlike the GeoJSON specification, the Oracle NoSQL Database uses a *geodetic* coordinate system, as defined in the World Geodetic System, WGS84, (https://gisgeography.com/wgs84-world-geodetic-system). A geodetic line between two points is the shortest line that can be drawn between the two points on the ellipsoidal surface of the earth.

GeoJSON Data Definitions

The GeoJSON specification (https://tools.ietf.org/html/rfc7946) states that for a JSON object to be a geometry, it requires two fields, *type* and *coordinates*. The value of the `type` field specifies the kind of geometric shape the object describes. The value of the `type` field must be one of the following strings, corresponding to different kinds of geometries:

- Point
- LineSegment
- Polygon
- MultiPoint
- MultiLineString
- MultiPolygon
- GeometryCollection

The `coordinates` value is an array with elements that define the geometrical shape. An exception to this is the *GeometryCollection* type, which is described below. The `coordinates` value depends on the geometric shape, but in all cases, specifies a number of positions. A *position* defines a position on the surface of the earth as an array of two double numbers, where the first number is the longitude and the second number is the latitude. Longitude and latitude are specified as degrees and must range between -180 – +180 and -90 – +90, respectively.

> **Note:**
>
> The GeoJSON specification allows a third coordinate for the altitude of the position, but Oracle NoSQL Database does not support altitudes.

The kinds of geometries are defined as follows, each with an example of such an object:

Point — For type Point, the coordinates field is a single position:

```
{ "type" : "point", "coordinates" : [ 23.549, 35.2908 ] }
```

LineString — A LineString is one or more connected lines, with the end-point of one line being the start-point of the next. The coordinates field is an array of two or more positions. The first position is the start point of the first line, and each subsequent position is the end point of the previous line and the start of the next line. Lines can cross each other.

```
{
"type" : "LineString",
"coordinates" : [
[-121.9447, 37.2975],
[-121.9500, 37.3171],
[-121.9892, 37.3182],
[-122.1554, 37.3882],
[-122.2899, 37.4589],
[-122.4273, 37.6032],
[-122.4304, 37.6267],
[-122.3975, 37.6144]
]
}
```

Polygon — A polygon defines a surface area by specifying its outer perimeter and the perimeters of any potential holes inside the area. More precisely, a polygon consists of one or more linear rings, where (a) a linear ring is a closed LineString with four or more positions, (b) the first and last positions are equivalent, and they must contain identical values, (c) a linear ring is the boundary of a surface or the boundary of a hole in a surface, and (d) a linear ring must follow the right-hand rule with respect to the area it bounds. That is, positions for exterior rings must be ordered counterclockwise, and positions for holes must be ordered clockwise. Then, the coordinates field of a polygon must be an array of linear ring coordinate arrays, where the first must be the exterior ring, and any others must be interior rings.

The exterior ring bounds the surface, and the interior rings (if present) bound holes within the surface. The example below shows a polygon with no holes.

```
{
"type" : "polygon",
"coordinates" : [ [
[23.48, 35.16],
[24.30, 35.16],
[24.30, 35.50],
[24.16, 35.61],
[23.74, 35.70],
[23.56, 35.60],
[23.48, 35.16]
]
]
}
```

MultiPoint — For type MultiPoint, the coordinates field is an array of two or more positions:

```
{
"type" : "MultiPoint",
```

```
"coordinates" : [
[-121.9447, 37.2975],
[-121.9500, 37.3171],
[-122.3975, 37.6144]
]
}
```

MultiLineString — For type MultiLineString, the coordinates member is an array of LineString coordinate arrays.

```
{
"type": "MultiLineString",
"coordinates": [
[ [100.0, 0.0], [01.0, 1.0] ],
[ [102.0, 2.0], [103.0, 3.0] ]
]
}
```

MultiPolygon — For type MultiPolygon, the coordinates member is an array of Polygon coordinate arrays.

```
 {
"type": "MultiPolygon",
"coordinates": [
[
[
[102.0, 2.0],
[103.0, 2.0],
[103.0, 3.0],
[102.0, 3.0],
[102.0, 2.0]
]
],

[
[
[100.0, 0.0],
[101.0, 0.0],
[101.0, 1.0],
[100.0, 1.0],
[100.0, 0.0]
]
]
]
}
```

GeometryCollection — Instead of a coordinates field, a GeometryCollection has a geometries” field. The value of geometries is an array. Each element of this array is a GeoJSON object whose kind is one of the six kinds defined above. In general, a GeometryCollection is a heterogeneous composition of smaller geometries.

```
{    "type": "GeometryCollection",
"geometries": [
{
```

```
"type": "Point",
"coordinates": [100.0, 0.0]
},
{"type": "LineString",
"coordinates": [ [101.0, 0.0], [102.0, 1.0] ]
}
]
}
```

> **Note:**
>
> The GeoJSON specification defines two additional kinds of entities, *Feature* and *FeatureCollection*. The Oracle NoSQL Database does not support these entities.

Searching GeoJSON Data

The Oracle NoSQL Database has the following functions to use for searching GeoJSON data that has some relationship with a search geometry.

- `boolean geo_intersect(any*, any*)`
- `boolean geo_inside(any*, any*)`
- `boolean geo_within_distance(any*, any*, double)`
- `boolean geo_near(any*, any*, double)`

In addition to the search functions, two other functions are available, and listed as the last two rows of the table:

Function	Type	Details
`geo_intersect(any*, any*)`	boolean	Raises an error at compile time if the function can detect that any operand will not return a single valid GeoJson object. Otherwise, the runtime behavior is as follows: • Returns false if any operand returns 0 or more than 1 items. • Returns NULL if any operand returns NULL. • Returns false if any operand returns an item that is not a valid GeoJson object. • Finally, if both operands return a single GeoJson object, returns true if the two geometries have any points in common. Otherwise, returns false.
`geo_inside(any*, any*)`	boolean	Raises an error at compile time if the function can detect that any operand will not return a single valid GeoJson object. Otherwise, the runtime behavior is as follows: • Returns false if any operand returns 0 or more than 1 item. • Returns NULL if any operand returns NULL. • Returns false if any operand returns an item that is not a valid GeoJson object. • Finally, if both operands return a single GeoJson object and the second GeoJson is a polygon, the function returns true if the first geometry is completely contained inside the second polygon, with all of its points belonging to the interior of the polygon. The interior of a polygon is all the points in the polygon, except the points of the linear rings that define the polygon's boundary. Otherwise, returns false.

Function	Type	Details
`geo_within_distance(any*, any*, double)`	boolean	Raises an error at compile time if the function detects that the first two operands will not return a single valid GeoJson object. Otherwise, the runtime behavior is as follows: • Returns false if any of the first two operands returns 0 or more than 1 item. • Returns NULL if any of the first two operands returns NULL. • Returns false if any of the first two operands returns an item that is not a valid GeoJson object. • Finally, if both of the first two operands return a single GeoJson object, the function returns true if the first geometry is within a distance of N meters from the second geometry, where N is the number returned by the third operand. The distance between 2 geometries is defined as the minimum among the distances of any pair of points where the first point belongs to the first geometry, and the second point to the second geometry. Otherwise, returns false.
`geo_near(any*, any*, double)`	boolean	The `geo_near` funcion is converted internally to a `geo_within_distance` function, with an an (implicit) order by the distance between the two geometries. However, if the query has an (explicit) order-by already, the function performs no ordering by distance. The `geo_near` function can appear only in the WHERE clause, and must be a top-level predicate. The `geo_near` function cannot be nested under an OR or NOT operator.
`geo_distance(any*, any*)`	double	Raises an error at compile time if the function detects that an operand will not return a single valid GeoJson object. Otherwise, the runtime behavior is as follows: • Returns -1 if any of the operands returns zero or more than 1 item. • Returns -1 if any of the operands is not a geometry. • Returns NULL if any operand returns NULL. • Otherwise the function returns the geodetic distance between the 2 input geometries. The returned distance is the minimum among the distances of any pair of points, where the first point belongs to the first geometry and the second point to the second geometry. Between two such points, their distance is the length of the geodetic line that connects the points.
`geo_is_geometry(any*)`	boolean	• Returns false if an operand returns zero or more than 1 item. • Returns NULL if an operand returns NULL. • Returns true if the input is a single valid GeoJson object. Otherwise, false.

7

Working With Indexes

The SQL for Oracle NoSQL Database query processor can detect which of the existing indexes on a table can be used to optimize the execution of a query. This chapter provides a brief examples-based introduction to index creation, and queries using indexes. For a more detailed description of index creation and usage, see *SQL Reference Guide*.

To make it possible to fit the example output on the page, the examples in this chapter use `mode LINE`.

Basic Indexing

This section builds on the examples that you began in Working with complex data.

```
sql-> mode LINE
Query output mode is LINE
sql-> create index idx_income on Persons (income);
Statement completed successfully
sql-> create index idx_age on Persons (age);
Statement completed successfully
sql-> SELECT * from Persons
WHERE income > 10000000 and age < 40;

 > Row 0
 +-------------+-----------------------------------------+
 | id          | 3                                       |
 +-------------+-----------------------------------------+
 | firstname   | John                                    |
 +-------------+-----------------------------------------+
 | lastname    | Morgan                                  |
 +-------------+-----------------------------------------+
 | age         | 38                                      |
 +-------------+-----------------------------------------+
 | income      | 100000000                               |
 +-------------+-----------------------------------------+
 | lastLogin   | 2016-11-29T08:21:35.4971                |
 +-------------+-----------------------------------------+
 | address     | street                  | 187 Aspen Drive |
 |             | city                    | Middleburg      |
 |             | state                   | FL              |
 |             | zipcode                 | NULL            |
 |             | phones                                    |
 |             |     type                | work            |
 |             |     areacode            | 305             |
 |             |     number              | 1234079         |
 |             |                                           |
 |             |     type                | home            |
 |             |     areacode            | 305             |
 |             |     number              | 2066401         |
 +-------------+-----------------------------------------+
```

```
 | connections | 1                                          |
 |             | 4                                          |
 |             | 2                                          |
 +-------------+--------------------------------------------+
 | expenses    | food                      | 2000           |
 |             | gas                       | 10             |
 |             | travel                    | 700            |
 +-------------+---------------------------+----------------+

1 row returned
```

Using Index Hints

In the previous section, both indexes are applicable. For index idx_income, the query condition income > 10000000 can be used as the starting point for an index scan that will retrieve only the index entries and associated table rows that satisfy this condition. Similarly, for index idx_age, the condition age < 40 can be used as the stopping point for the index scan. SQL for Oracle NoSQL Database has no way of knowing which of the 2 predicates is more selective, and it assigns the same "value" to each index, eventually picking the one whose name is first alphabetically. In the previous example, idx_age was used. To choose the idx_income index instead, the query should be written with an index hint:

```
sql-> SELECT /*+ FORCE_INDEX(Persons idx_income) */ * from Persons
WHERE income > 10000000 and age < 40;

> Row 0
 +-------------+--------------------------------------------+
 | id          | 3                                          |
 +-------------+--------------------------------------------+
 | firstname   | John                                       |
 +-------------+--------------------------------------------+
 | lastname    | Morgan                                     |
 +-------------+--------------------------------------------+
 | age         | 38                                         |
 +-------------+--------------------------------------------+
 | income      | 100000000                                  |
 +-------------+--------------------------------------------+
 | lastLogin   | 2016-11-29T08:21:35.4971                   |
 +-------------+--------------------------------------------+
 | address     | street                    | 187 Aspen Drive |
 |             | city                      | Middleburg      |
 |             | state                     | FL              |
 |             | zipcode                   | NULL            |
 |             | phones                                      |
 |             |     type                  | work            |
 |             |     areacode              | 305             |
 |             |     number                | 1234079         |
 |             |                                             |
 |             |     type                  | home            |
 |             |     areacode              | 305             |
 |             |     number                | 2066401         |
 +-------------+--------------------------------------------+
 | connections | 1                                          |
 |             | 4                                          |
 |             | 2                                          |
```

```
 +-------------+-----------------------------------------+
 | expenses    | food                      | 2000        |
 |             | gas                       | 10          |
 |             | travel                    | 700         |
 +-------------+-----------------------------------------+

1 row returned
```

As shown above, hints are written as a special kind of comment that must be placed immediately after the SELECT keyword. What distinguishes a hint from a regular comment is the "+" character immediately after (without any space) the opening "/*".

Complex Indexes

The following example demonstrates indexing of multiple table fields, indexing of nested fields, and the use of "filtering" predicates during index scans.

```
sql-> create index idx_state_city_income on
Persons (address.state, address.city, income);
Statement completed successfully
sql-> SELECT * from Persons p WHERE p.address.state = "MA"
and income > 79000;

> Row 0
 +-------------+-----------------------------------------------+
 | id          | 4                                             |
 +-------------+-----------------------------------------------+
 | firstname   | Peter                                         |
 +-------------+-----------------------------------------------+
 | lastname    | Smith                                         |
 +-------------+-----------------------------------------------+
 | age         | 38                                            |
 +-------------+-----------------------------------------------+
 | income      | 80000                                         |
 +-------------+-----------------------------------------------+
 | lastLogin   | 2016-10-19T09:18:05.5555                      |
 +-------------+-----------------------------------------------+
 | address     | street                    | 364 Mulberry Street |
 |             | city                      | Leominster          |
 |             | state                     | MA                  |
 |             | zipcode                   | NULL                |
 |             | phones                                          |
 |             |     type                  | work                |
 |             |     areacode              | 339                 |
 |             |     number                | 4120211             |
 |             |                                                 |
 |             |     type                  | work                |
 |             |     areacode              | 339                 |
 |             |     number                | 8694021             |
 |             |                                                 |
 |             |     type                  | home                |
 |             |     areacode              | 339                 |
 |             |     number                | 1205678             |
 |             |                                                 |
 |             |     type                  | home                |
```

```
|             |     areacode             | 305                |
|             |     number               | 8064321            |
+-------------+--------------------------------------------+
| connections | 3                                          |
|             | 5                                          |
|             | 1                                          |
|             | 2                                          |
+-------------+--------------------------------------------+
| expenses    | books                    | 240                |
|             | clothes                  | 2000               |
|             | food                     | 6000               |
|             | shoes                    | 1200               |
+-------------+--------------------------------------------+

1 row returned
```

Index idx_state_city_income is applicable to the above query. Specifically, the state = "MA" condition can be used to establish the boundaries of the index scan (only index entries whose first field is "MA" will be scanned). Further, during the index scan, the income condition can be used as a "filtering" condition, to skip index entries whose third field is less or equal to 79000. As a result, only rows that satisfy both conditions are retrieved from the table.

Multi-Key Indexes

A multi-key index indexes all the elements of an array, or all the elements and/or all the keys of a map. For such indexes, for each table row, the index contains as many entries as the number of elements/entries in the array/map that is being indexed. Only one array/map may be indexed.

```
sql-> create index idx_areacode on
Persons (address.phones[].areacode);
Statement completed successfully
sql-> SELECT * FROM Persons p WHERE
p.address.phones.areacode =any 339;

 > Row 0
 +-------------+--------------------------------------------+
 | id          | 2                                          |
 +-------------+--------------------------------------------+
 | firstname   | John                                       |
 +-------------+--------------------------------------------+
 | lastname    | Anderson                                   |
 +-------------+--------------------------------------------+
 | age         | 35                                         |
 +-------------+--------------------------------------------+
 | income      | 100000                                     |
 +-------------+--------------------------------------------+
 | lastLogin   | 2016-11-28T13:01:11.2088                   |
 +-------------+--------------------------------------------+
 | address     | street                   | 187 Hill Street |
 |             | city                     | Beloit          |
 |             | state                    | WI              |
 |             | zipcode                  | 53511           |
 |             | phones                                     |
 |             |     type                 | home            |
```

```
|             |     areacode             | 339                 |
|             |     number               | 1684972             |
+-------------+--------------------------------------------------+
| connections | 1                                                |
|             | 3                                                |
+-------------+--------------------------------------------------+
| expenses    | books                    | 100                 |
|             | food                     | 1700                |
|             | travel                   | 2100                |
+-------------+--------------------------------------------------+

> Row 1
+-------------+--------------------------------------------------+
| id          | 4                                                |
+-------------+--------------------------------------------------+
| firstname   | Peter                                            |
+-------------+--------------------------------------------------+
| lastname    | Smith                                            |
+-------------+--------------------------------------------------+
| age         | 38                                               |
+-------------+--------------------------------------------------+
| income      | 80000                                            |
+-------------+--------------------------------------------------+
| lastLogin   | 2016-10-19T09:18:05.5555                         |
+-------------+--------------------------------------------------+
| address     | street                   | 364 Mulberry Street |
|             | city                     | Leominster          |
|             | state                    | MA                  |
|             | zipcode                  | NULL                |
|             | phones                                           |
|             |     type                 | work                |
|             |     areacode             | 339                 |
|             |     number               | 4120211             |
|             |                                                  |
|             |     type                 | work                |
|             |     areacode             | 339                 |
|             |     number               | 8694021             |
|             |                                                  |
|             |     type                 | home                |
|             |     areacode             | 339                 |
|             |     number               | 1205678             |
|             |                                                  |
|             |     type                 | home                |
|             |     areacode             | 305                 |
|             |     number               | 8064321             |
+-------------+--------------------------------------------------+
| connections | 3                                                |
|             | 5                                                |
|             | 1                                                |
|             | 2                                                |
+-------------+--------------------------------------------------+
| expenses    | books                    | 240                 |
|             | clothes                  | 2000                |
|             | food                     | 6000                |
|             | shoes                    | 1200                |
+-------------+--------------------------------------------------+
```

```
> Row 2
+-------------+-----------------------------------------------+
| id          | 5                                             |
+-------------+-----------------------------------------------+
| firstname   | Dana                                          |
+-------------+-----------------------------------------------+
| lastname    | Scully                                        |
+-------------+-----------------------------------------------+
| age         | 47                                            |
+-------------+-----------------------------------------------+
| income      | 400000                                        |
+-------------+-----------------------------------------------+
| lastLogin   | 2016-11-08T09:16:46.3929                      |
+-------------+-----------------------------------------------+
| address     | street                   | 427 Linden Avenue  |
|             | city                     | Monroe Township    |
|             | state                    | NJ                 |
|             | zipcode                  | NULL               |
|             | phones                                        |
|             |     type                 | work               |
|             |     areacode             | 201                |
|             |     number               | 3213267            |
|             |                                               |
|             |     type                 | work               |
|             |     areacode             | 201                |
|             |     number               | 8765421            |
|             |                                               |
|             |     type                 | home               |
|             |     areacode             | 339                |
|             |     number               | 3414578            |
+-------------+-----------------------------------------------+
| connections | 2                                             |
|             | 4                                             |
|             | 1                                             |
|             | 3                                             |
+-------------+-----------------------------------------------+
| expenses    | clothes                  | 1500               |
|             | food                     | 900                |
|             | shoes                    | 1000               |
+-------------+-----------------------------------------------+


3 rows returned
```

In the above example, a multi-key index is created on all the area codes in the Persons table, mapping each area code to the persons that have a phone number with that area code. The query is looking for persons who have a phone number with area code 339. The index is applicable to the query and so the key 339 will be searched for in the index and all the associated table rows will be retrieved.

```
sql-> create index idx_expenses on
Persons (expenses.keys(), expenses.values());
Statement completed successfully
sql-> SELECT * FROM Persons p WHERE p.expenses.food > 1000;
```

```
> Row 0
+-------------+-----------------------------------------+
| id          | 2                                       |
+-------------+-----------------------------------------+
| firstname   | John                                    |
+-------------+-----------------------------------------+
| lastname    | Anderson                                |
+-------------+-----------------------------------------+
| age         | 35                                      |
+-------------+-----------------------------------------+
| income      | 100000                                  |
+-------------+-----------------------------------------+
| lastLogin   | 2016-11-28T13:01:11.2088                |
+-------------+-----------------------------------------+
| address     | street                  | 187 Hill Street |
|             | city                    | Beloit          |
|             | state                   | WI              |
|             | zipcode                 | 53511           |
|             | phones                                    |
|             |     type                | home            |
|             |     areacode            | 339             |
|             |     number              | 1684972         |
+-------------+-----------------------------------------+
| connections | 1                                       |
|             | 3                                       |
+-------------+-----------------------------------------+
| expenses    | books                   | 100             |
|             | food                    | 1700            |
|             | travel                  | 2100            |
+-------------+-----------------------------------------+

> Row 1
+-------------+-----------------------------------------+
| id          | 3                                       |
+-------------+-----------------------------------------+
| firstname   | John                                    |
+-------------+-----------------------------------------+
| lastname    | Morgan                                  |
+-------------+-----------------------------------------+
| age         | 38                                      |
+-------------+-----------------------------------------+
| income      | 100000000                               |
+-------------+-----------------------------------------+
| lastLogin   | 2016-11-29T08:21:35.4971                |
+-------------+-----------------------------------------+
| address     | street                  | 187 Aspen Drive |
|             | city                    | Middleburg      |
|             | state                   | FL              |
|             | zipcode                 | NULL            |
|             | phones                                    |
|             |     type                | work            |
|             |     areacode            | 305             |
|             |     number              | 1234079         |
|             |                                           |
|             |     type                | home            |
```

```
|             |     areacode             | 305             |
|             |     number               | 2066401         |
+-------------+--------------------------------------------+
| connections | 1                                          |
|             | 4                                          |
|             | 2                                          |
+-------------+--------------------------------------------+
| expenses    | food                     | 2000            |
|             | gas                      | 10              |
|             | travel                   | 700             |
+-------------+--------------------------------------------+

> Row 2
+-------------+--------------------------------------------+
| id          | 4                                          |
+-------------+--------------------------------------------+
| firstname   | Peter                                      |
+-------------+--------------------------------------------+
| lastname    | Smith                                      |
+-------------+--------------------------------------------+
| age         | 38                                         |
+-------------+--------------------------------------------+
| income      | 80000                                      |
+-------------+--------------------------------------------+
| lastLogin   | 2016-10-19T09:18:05.5555                   |
+-------------+--------------------------------------------+
| address     | street                   | 364 Mulberry Street |
|             | city                     | Leominster          |
|             | state                    | MA                  |
|             | zipcode                  | NULL                |
|             | phones                                     |
|             |     type                 | work                |
|             |     areacode             | 339                 |
|             |     number               | 4120211             |
|             |                                            |
|             |     type                 | work                |
|             |     areacode             | 339                 |
|             |     number               | 8694021             |
|             |                                            |
|             |     type                 | home                |
|             |     areacode             | 339                 |
|             |     number               | 1205678             |
|             |                                            |
|             |     type                 | home                |
|             |     areacode             | 305                 |
|             |     number               | 8064321             |
+-------------+--------------------------------------------+
| connections | 3                                          |
|             | 5                                          |
|             | 1                                          |
|             | 2                                          |
+-------------+--------------------------------------------+
| expenses    | books                    | 240                 |
|             | clothes                  | 2000                |
|             | food                     | 6000                |
|             | shoes                    | 1200                |
```

```
 +------------+-------------------------------------------------+

3 rows returned
```

In the above example, a multi-key index is created on all the expenses entries in the Persons table, mapping each category C and each amount A associated with that category to the persons that have an entry (C, A) in their expenses map. The query is looking for persons who spent more than 1000 on food. The index is applicable to the query and so only the index entries whose first field (the map key) is equal to "food" and second key (the amount) is greater than 1000 will be scanned and the associated rows retrieved.

Indexing JSON Data

An index is a JSON index if it indexes at least one field that is contained inside JSON data.

Because JSON is schema-less, it is possible for JSON data to differ in type across table rows. However, when indexing JSON data, the data type must be consistent across table rows or the index creation will fail. Further, once one or more JSON indexes have been created, any attempt to write data of an incorrect type will fail.

With the exception of the previous restriction, indexing JSON data and working with JSON indexes behaves in much the same way as indexing non-JSON data. To create the index, specify a path to the JSON field using dot notation. You must also specify the data's type, using the `AS` keyword.

The following examples are built on the examples shown in Working with JSON.

```
sql-> create index idx_json_income on JSONPersons (person.income
as integer);
Statement completed successfully
sql-> create index idx_json_age on JSONPersons (person.age as integer);
Statement completed successfully
sql->
```

You can then run a query in the normal way, and the index idx_json_income will be automatically used. But as shown at the beginning of this chapter (Basic Indexing), the query processor will not know which index to use. To require the use of a particular index provide an index hint as normal:

```
sql-> SELECT /*+ FORCE_INDEX(JSONPersons idx_json_income) */ *
from JSONPersons j WHERE j.person.income > 10000000 and
j.person.age < 40;

 > Row 0
 +-----------+-------------------------------------------+
 | id        | 3                                         |
 +-----------+-------------------------------------------+
 | person    | address                                   |
 |           |     city         | Middleburg             |
 |           |     phones                                |
 |           |         areacode | 305                    |
 |           |         number   | 1234079                |
 |           |         type     | work                   |
 |           |                                           |
 |           |         areacode | 305                    |
```

```
 |           |         number   | 2066401                  |
 |           |         type     | home                     |
 |           |     state        | FL                       |
 |           |     street       | 187 Aspen Drive          |
 |           | age              | 38                       |
 |           | connections                                 |
 |           |                    1                        |
 |           |                    4                        |
 |           |                    2                        |
 |           | expenses                                    |
 |           |     food         | 2000                     |
 |           |     gas          | 10                       |
 |           |     travel       | 700                      |
 |           | firstname        | John                     |
 |           | income           | 100000000                |
 |           | lastLogin        | 2016-11-29T08:21:35.4971 |
 |           | lastname         | Morgan                   |
 +-----------+------------------------------------------+

1 row returned
sql->
```

Finally, when creating a multi-key index on a JSON map, a type must not be given for the `.keys()` expression. This is because the type will always be `String`. However, a type declaration is required for the `.values()` expression:

```
sql-> create index idx_json_expenses on JSONPersons
(person.expenses.keys(), person.expenses.values() as integer);
Statement completed successfully
sql-> SELECT * FROM JSONPersons j WHERE j.person.expenses.food > 1000;

 > Row 0
 +-----------+------------------------------------------+
 | id        | 2                                        |
 +-----------+------------------------------------------+
 | person    | address                                  |
 |           |     city         | Beloit                |
 |           |     phones                               |
 |           |         areacode | 339                   |
 |           |         number   | 1684972               |
 |           |         type     | home                  |
 |           |     state        | WI                    |
 |           |     street       | 187 Hill Street       |
 |           |     zipcode      | 53511                 |
 |           | age              | 35                    |
 |           | connections                              |
 |           |                    1                     |
 |           |                    3                     |
 |           | expenses                                 |
 |           |     books        | 100                   |
 |           |     food         | 1700                  |
 |           |     travel       | 2100                  |
 |           | firstname        | John                  |
 |           | income           | 100000                |
 |           | lastLogin        | 2016-11-28T13:01:11.2088 |
```

```
|           | lastname         | Anderson                  |
+-----------+------------------------------------------+

> Row 1
+-----------+------------------------------------------+
| id        | 3                                        |
+-----------+------------------------------------------+
| person    | address                                  |
|           |     city         | Middleburg            |
|           |     phones                               |
|           |         areacode | 305                   |
|           |         number   | 1234079               |
|           |         type     | work                  |
|           |                                          |
|           |         areacode | 305                   |
|           |         number   | 2066401               |
|           |         type     | home                  |
|           |     state        | FL                    |
|           |     street       | 187 Aspen Drive       |
|           | age              | 38                    |
|           | connections                              |
|           |                    1                     |
|           |                    4                     |
|           |                    2                     |
|           | expenses                                 |
|           |     food         | 2000                  |
|           |     gas          | 10                    |
|           |     travel       | 700                   |
|           | firstname        | John                  |
|           | income           | 100000000             |
|           | lastLogin        | 2016-11-29T08:21:35.4971 |
|           | lastname         | Morgan                |
+-----------+------------------------------------------+

> Row 2
+-----------+------------------------------------------+
| id        | 4                                        |
+-----------+------------------------------------------+
| person    | address                                  |
|           |     city         | Leominster            |
|           |     phones                               |
|           |         areacode | 339                   |
|           |         number   | 4120211               |
|           |         type     | work                  |
|           |                                          |
|           |         areacode | 339                   |
|           |         number   | 8694021               |
|           |         type     | work                  |
|           |                                          |
|           |         areacode | 339                   |
|           |         number   | 1205678               |
|           |         type     | home                  |
|           |                    null                  |
|           |                                          |
|           |         areacode | 305                   |
|           |         number   | 8064321               |
```

```
 |            |         type      | home                       |
 |            |     state         | MA                         |
 |            |     street        | 364 Mulberry Street        |
 |            | age               | 38                         |
 |            | connections                                    |
 |            |                     3                          |
 |            |                     5                          |
 |            |                     1                          |
 |            |                     2                          |
 |            | expenses                                       |
 |            |     books         | 240                        |
 |            |     clothes       | 2000                       |
 |            |     food          | 6000                       |
 |            |     shoes         | 1200                       |
 |            | firstname         | Peter                      |
 |            | income            | 80000                      |
 |            | lastLogin         | 2016-10-19T09:18:05.5555 |
 |            | lastname          | Smith                      |
 +------------+--------------------------------------------+

3 rows returned
sql->
```

Be aware that all the other constraints that apply to a non-JSON multi-keyed index also apply to a JSON multi-keyed index.

8
Working with Table Rows

This chapter provides examples on how to insert and update table rows using SQL for Oracle NoSQL Database INSERT and UPDATE statements.

Adding Table Rows using INSERT and UPSERT

This topic provides examples on how to add table rows using the SQL for Oracle NoSQL Database INSERT and UPSERT statements.

You use the INSERT statement to insert or update a single row in an existing table.

Examples:

If you executed the SQLBasicExamples Script, you should already have created the table named `Users`. The table had this definition:

```
CREATE TABLE Users
(
    id integer,
    firstname string,
    lastname string,
    age integer,
    income integer,
    primary key (id)
);
sql-> describe table Users;
 === Information ===
 +-------+-----+-------+----------+----------+--------+----------+---------
+-------------+
 | name  | ttl | owner | sysTable | r2compat | parent | children | indexes |
description |
 +-------+-----+-------+----------+----------+--------+----------+---------
+-------------+
 | Users |     |       | N        | N        |        |          |
|             |
 +-------+-----+-------+----------+----------+--------+----------+---------
+-------------+

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

```
 |  2 | firstname | String  | Y        | NullValue |          |
|          |
 +----+-----------+---------+----------+-----------+----------+------------
+----------+
 |  3 | lastname  | String  | Y        | NullValue |          |
|          |
 +----+-----------+---------+----------+-----------+----------+------------
+----------+
 |  4 | age       | Integer | Y        | NullValue |          |
|          |
 +----+-----------+---------+----------+-----------+----------+------------
+----------+
 |  5 | income    | Integer | Y        | NullValue |          |
|          |
 +----+-----------+---------+----------+-----------+----------+------------
+----------+
```

To insert a new row into the `Users` table, use the `INSERT` statement as follows. Because you are adding values to all table columns, you do not need to specify column names explicitly:

```
sql-> INSERT INTO Users VALUES (10, "John", "Smith", 22, 45000);
{"NumRowsInserted":1}
1 row returned
sql-> select * from Users;
{"id":10,"firstname":"John","lastname":"Smith","age":22,"income":45000}
```

To insert data into some, but not all, table columns, specify the column names explicitly in the INSERT statement. Any columns that you do not specify are assigned either `NULL` or the default value supplied when you created the table:

```
sql-> INSERT INTO Users (id, firstname, income)
VALUES (11, "Mary", 5000);
{"NumRowsInserted":1}
1 row returned

sql-> select * from Users;
{"id":11,"firstname":"Mary","lastname":null,"age":null,"income":5000}
{"id":10,"firstname":"John","lastname":"Smith","age":22,"income":45000}
2 rows returned
```

Using the UPSERT Statement

The word `UPSERT` combines `UPDATE` and `INSERT`, describing it statement's function. Use an `UPSERT` statement to insert a row where it does not exist, or to update the row with new values when it does.

For example, if you already inserted a new row as described in the previous section, executing the next statement *updates* user John's age to 27, and income to 60,000. If you did not execute the previous `INSERT` statement, the `UPSERT` statement *inserts* a new row with user id 10 to the `Users` table.

```
sql-> UPSERT INTO Users VALUES (10, "John", "Smith", 27, 60000);
{"NumRowsInserted":0}
1 row returned
```

```
sql-> UPSERT INTO Users VALUES (11, "Mary", "Brown", 28, 70000);
{"NumRowsInserted":0}
1 row returned

sql-> select * from Users;
{"id":10,"firstname":"John","lastname":"Smith","age":22,"income":60000}
{"id":11,"firstname":"Mary","lastname":"Brown","age":28,"income":70000}
2 rows returned
```

Using an IDENTITY Column

You can use IDENTITY columns to automatically generate values for a table column each time you insert a new table row. See Identity Column in the *SQL Reference Guide*.

Here are a few examples for how to use the INSERT statements for both flavors of an IDENTITY column:

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

Create a table named `Employee_test` using one column, DeptId, as GENERATED ALWAYS AS IDENTITY. This IDENTITY column is not the primary key. Insert a few rows into the table.

```
sql-> CREATE TABLE EmployeeTest
(
    Empl_id INTEGER,
    Name STRING,
    DeptId INTEGER GENERATED ALWAYS AS IDENTITY (CACHE 1),
    PRIMARY KEY(Empl_id)
);

INSERT INTO Employee_test VALUES (148, 'Sally', DEFAULT);
INSERT INTO Employee_test VALUES (250, 'Joe', DEFAULT);
INSERT INTO Employee_test VALUES (346, 'Dave', DEFAULT);
```

The INSERT statement inserts the following rows with the system generates values 1, 2, and 3 for the IDENTITY column `DeptId`.

Empl_id	Name	DeptId
148	Sally	1
250	Joe	2
346	Dave	3

You cannot specify a value for the `DeptId` IDENTITY column when inserting a row to the `Employee_test` table, because you defined that column as `GENERATED ALWAYS AS IDENTITY`. Specifying DEFAULT as the column value, the system generates the next IDENTITY value. Conversely, trying to execute the following SQL statement causes an exception, because you supply a value (200) for the `DeptId` column.

```
sql-> INSERT INTO Employee_test VALUES (566, 'Jane', 200);
```

If you create the column as `GENERATED BY DEFAULT AS IDENTITY` for the `Employee_test` table, the system generates a value only if you fail to supply one. For example, if you define the

Employee_test table as follows, then execute the INSERT statement as above, the statement inserts the value 200 for the employee's DeptId column.

```
CREATE Table Employee_test
(
    Empl_id INTEGER,
    Name STRING,
    DeptId INTEGER GENERATED BY DEFAULT AS IDENTITY (CACHE 1),
    PRIMARY KEY(Empl_id)
);
```

Modifying Table Rows using UPDATE Statements

This topic provides examples of how to update table rows using SQL for Oracle NoSQL Database UPDATE statements. These are an efficient way to update table row data, because UPDATE statements make *server-side updates* directly, without requiring a Read/Modify/Write update cycle.

> **Note:**
>
> You can use UPDATE statements to update only an existing row. You cannot use UPDATE to either create new rows, or delete existing rows. An UPDATE statement can modify only a single row at a time.

Example Data

This chapter's examples uses the data loaded by the SQLJSONExamples script, which can be found in the Examples download package. For details on using this script, the sample data it loads, and the Examples download, see See SQLJSONExamples Script.

Changing Field Values

In the simplest case, you can change the value of a field using the Update Statement SET clause. The JSON example data set has a row which contains just an array and an integer. This is row ID 6:

```
sql-> mode column
Query output mode is COLUMN
sql-> SELECT * from JSONPersons j WHERE j.id = 6;
 +----+-------------------+
 | id |      person       |
 +----+-------------------+
 |  6 | myarray           |
 |    |             1     |
 |    |             2     |
 |    |             3     |
 |    |             4     |
 |    | mynumber | 5      |
 +----+-------------------+
```

```
1 row returned
```

You can change the value of `mynumber` in that row using the following statement:

```
sql-> UPDATE JSONPersons j
          SET j.person.mynumber = 100
          WHERE j.id = 6;
 +----------+
 | Column_1 |
 +----------+
 |        1 |
 +----------+

1 row returned
sql-> SELECT * from JSONPersons j WHERE j.id = 6;
 +----+--------------------+
 | id |      person        |
 +----+--------------------+
 |  6 | myarray            |
 |    |            1       |
 |    |            2       |
 |    |            3       |
 |    |            4       |
 |    | mynumber | 100     |
 +----+--------------------+

1 row returned
```

In the previous example, the results returned by the Update statement was not very informative, so we were required to reissue the Select statement in order to view the results of the update. You can avoid that by using a RETURNING clause. This functions exactly like a Select statement:

```
sql-> UPDATE JSONPersons j
          SET j.person.mynumber = 200
          WHERE j.id = 6
          RETURNING *;
 +----+--------------------+
 | id |      person        |
 +----+--------------------+
 |  6 | myarray            |
 |    |            1       |
 |    |            2       |
 |    |            3       |
 |    |            4       |
 |    | mynumber | 200     |
 +----+--------------------+

1 row returned
sql->
```

You can further limit and customize the displayed results in the same way that you can do so using a SELECT statement:

```
sql-> UPDATE JSONPersons j
          SET j.person.mynumber = 300
          WHERE j.id = 6
          RETURNING id, j.person.mynumber AS MyNumber;
 +----+----------------------+
 | id |      MyNumber        |
 +----+----------------------+
 |  6 | 300                  |
 +----+----------------------+

1 row returned
sql->
```

It is normally possible to update the value of a non-JSON field using the SET clause. However, you cannot change a field if it is a primary key. For example:

```
sql-> UPDATE JSONPersons j
          SET j.id = 1000
          WHERE j.id = 6
          RETURNING *;
Error handling command UPDATE JSONPersons j
SET j.id = 1000
WHERE j.id = 6
RETURNING *: Error: at (2, 4) Cannot update a primary key column
Usage:

Unknown statement

sql->
```

Modifying Array Values

You use the Update statement ADD clause to add elements into an array. You use a SET clause to change the value of an existing array element. And you use a REMOVE clause to remove elements from an array.

Adding Elements to an Array

The ADD clause requires you to identify the array position that you want to operate on, followed by the value you want to set to that position in the array. If the index value that you set is `0` or a negative number, the value that you specify is inserted at the beginning of the array.

If you do not provide an index position, the array value that you specify is appended to the end of the array.

```
sql-> SELECT * from JSONPersons j WHERE j.id = 6;
 +----+--------------------+
 | id |      person        |
 +----+--------------------+
 |  6 | myarray            |
 |    |            1       |
```

```
 |    |            2      |
 |    |            3      |
 |    |            4      |
 |    | mynumber | 300    |
 +----+-------------------+

1 row returned
sql-> UPDATE JSONPersons j
          ADD j.person.myarray 0 50,
          ADD j.person.myarray 100
          WHERE j.id = 6
          RETURNING *;
 +----+-------------------+
 | id |      person       |
 +----+-------------------+
 |  6 | myarray           |
 |    |            50     |
 |    |            1      |
 |    |            2      |
 |    |            3      |
 |    |            4      |
 |    |            100    |
 |    | mynumber | 300    |
 +----+-------------------+

1 row returned
sql->
```

Notice that multiple ADD clauses are used in the query above.

Array values get appended to the end of the array, even if you provide an array position that is larger than the size of the array. You can either provide an arbitrarily large number, or make use of the `size()` function:

```
sql-> UPDATE JSONPersons j
          ADD j.person.myarray (size(j.person.myarray) + 1) 400
          WHERE j.id = 6
          RETURNING *;
 +----+-------------------+
 | id |      person       |
 +----+-------------------+
 |  6 | myarray           |
 |    |            50     |
 |    |            1      |
 |    |            2      |
 |    |            3      |
 |    |            4      |
 |    |            100    |
 |    |            400    |
 |    | mynumber | 300    |
 +----+-------------------+

1 row returned
sql->
```

You can append values to the array using the built-in `seq_concat()` function:

```
sql-> UPDATE JSONPersons j
          ADD j.person.myarray seq_concat(66, 77, 88)
          WHERE j.id = 6
          RETURNING *;
 +----+-------------------+
 | id |      person       |
 +----+-------------------+
 |  6 | myarray           |
 |    |            50     |
 |    |            1      |
 |    |            2      |
 |    |            3      |
 |    |            4      |
 |    |            100    |
 |    |            400    |
 |    |            66     |
 |    |            77     |
 |    |            88     |
 |    | mynumber | 300    |
 +----+-------------------+

1 row returned
sql->
```

If you provide an array position that is between `0` and the array's size, then the value you specify will be inserted into the array *before* the specified position. To determine the correct position, start counting from 0:

```
UPDATE JSONPersons j
    ADD j.person.myarray 3 250
    WHERE j.id = 6
    RETURNING *;
 +----+-------------------+
 | id |      person       |
 +----+-------------------+
 |  6 | myarray           |
 |    |            50     |
 |    |            1      |
 |    |            2      |
 |    |            250    |
 |    |            3      |
 |    |            4      |
 |    |            100    |
 |    |            400    |
 |    |            66     |
 |    |            77     |
 |    |            88     |
 |    | mynumber | 300    |
 +----+-------------------+

1 row returned
sql->
```

Changing an Existing Element in an Array

To change an existing value in an array, use the SET clause and identify the value's position using `[]`. To determine the value's position, start counting from `0`:

```
sql-> UPDATE JSONPersons j
          SET j.person.myarray[3] = 1000
          WHERE j.id = 6
          RETURNING *;
 +----+-------------------+
 | id |      person       |
 +----+-------------------+
 |  6 | myarray           |
 |    |            50     |
 |    |            1      |
 |    |            2      |
 |    |            1000   |
 |    |            3      |
 |    |            4      |
 |    |            100    |
 |    |            400    |
 |    |            66     |
 |    |            77     |
 |    |            88     |
 |    | mynumber | 300    |
 +----+-------------------+

1 row returned
sql->
```

Removing Elements from Arrays

To remove an existing element from an array, use the REMOVE clause. To do this, you must identify the position of the element in the array that you want to remove. To determine the value's position, start counting from `0`:

```
sql-> UPDATE JSONPersons j
          REMOVE j.person.myarray[3]
          WHERE j.id = 6
          RETURNING *;
 +----+-------------------+
 | id |      person       |
 +----+-------------------+
 |  6 | myarray           |
 |    |            50     |
 |    |            1      |
 |    |            2      |
 |    |            3      |
 |    |            4      |
 |    |            100    |
 |    |            400    |
 |    |            66     |
 |    |            77     |
 |    |            88     |
```

```
 |    | mynumber | 300    |
 +----+--------------------+

1 row returned
sql->
```

It is possible for the array position to be identified by an expression. For example, in our sample data, some records include an array of phone numbers, and some of those phone numbers include a work number:

```
sql-> SELECT * FROM JSONPersons j WHERE j.id = 3;
 +----+-------------------------------------------+
 | id |                  person                   |
 +----+-------------------------------------------+
 |  3 | address                                   |
 |    |     city         | Middleburg             |
 |    |     phones                                |
 |    |         areacode | 305                    |
 |    |         number   | 1234079                |
 |    |         type     | work                   |
 |    |                                           |
 |    |         areacode | 305                    |
 |    |         number   | 2066401                |
 |    |         type     | home                   |
 |    |     state        | FL                     |
 |    |     street       | 187 Aspen Drive        |
 |    | age              | 38                     |
 |    | connections                               |
 |    |                    1                      |
 |    |                    4                      |
 |    |                    2                      |
 |    | expenses                                  |
 |    |     food         | 2000                   |
 |    |     gas          | 10                     |
 |    |     travel       | 700                    |
 |    | firstname        | John                   |
 |    | income           | 100000000              |
 |    | lastLogin        | 2016-11-29T08:21:35.4971 |
 |    | lastname         | Morgan                 |
 +----+-------------------------------------------+

1 row returned
sql->
```

We can remove the work number from the array in one of two ways. First, we can directly specify its position in the array (position 0), but that only removes a single element at a time. If we want to remove all the work numbers, we can do it by using the $element variable. To illustrate, we first add another work number to the array:

```
sql-> UPDATE JSONPersons j
          ADD j.person.address.phones 0
          {"type":"work", "areacode":415, "number":9998877}
          WHERE j.id = 3
          RETURNING *;
 +----+-------------------------------------------+
```

```
 | id |                   person                      |
 +----+-----------------------------------------------+
 |  3 | address                                       |
 |    |     city          | Middleburg                |
 |    |     phones                                    |
 |    |         areacode  | 415                       |
 |    |         number    | 9998877                   |
 |    |         type      | work                      |
 |    |                                               |
 |    |         areacode  | 305                       |
 |    |         number    | 1234079                   |
 |    |         type      | work                      |
 |    |                                               |
 |    |         areacode  | 305                       |
 |    |         number    | 2066401                   |
 |    |         type      | home                      |
 |    |     state         | FL                        |
 |    |     street        | 187 Aspen Drive           |
 |    | age               | 38                        |
 |    | connections                                   |
 |    |                     1                         |
 |    |                     4                         |
 |    |                     2                         |
 |    | expenses                                      |
 |    |     food          | 2000                      |
 |    |     gas           | 10                        |
 |    |     travel        | 700                       |
 |    | firstname         | John                      |
 |    | income            | 100000000                 |
 |    | lastLogin         | 2016-11-29T08:21:35.4971  |
 |    | lastname          | Morgan                    |
 +----+-----------------------------------------------+

1 row returned
sql->
```

Now we can remove all the work numbers as follows:

```
sql-> UPDATE JSONPersons j
          REMOVE j.person.address.phones[$element.type = "work"]
          WHERE j.id = 3
          RETURNING *;
 +----+-----------------------------------------------+
 | id |                   person                      |
 +----+-----------------------------------------------+
 |  3 | address                                       |
 |    |     city          | Middleburg                |
 |    |     phones                                    |
 |    |         areacode  | 305                       |
 |    |         number    | 2066401                   |
 |    |         type      | home                      |
 |    |     state         | FL                        |
 |    |     street        | 187 Aspen Drive           |
 |    | age               | 38                        |
 |    | connections                                   |
```

```
 |     |                    1                          |
 |     |                    4                          |
 |     |                    2                          |
 |     | expenses                                      |
 |     |     food          | 2000                      |
 |     |     gas           | 10                        |
 |     |     travel        | 700                       |
 |     | firstname         | John                      |
 |     | income            | 100000000                 |
 |     | lastLogin         | 2016-11-29T08:21:35.4971  |
 |     | lastname          | Morgan                    |
 +-----+-----------------------------------------------+

1 row returned
sql->
```

Modifying Map Values

To write a new field to a map, use the PUT clause. You can also use the PUT clause to change an existing map value. To remove a map field, use the REMOVE clause.

For example, consider the following two rows from our sample data:

```
sql-> SELECT * FROM JSONPersons j WHERE j.id = 6 OR j.id = 3;
 +-----+-----------------------------------------------+
 | id  |                   person                      |
 +-----+-----------------------------------------------+
 |   3 | address                                       |
 |     |     city          | Middleburg                |
 |     |     phones                                    |
 |     |         areacode  | 305                       |
 |     |         number    | 2066401                   |
 |     |         type      | home                      |
 |     |     state         | FL                        |
 |     |     street        | 187 Aspen Drive           |
 |     | age               | 38                        |
 |     | connections                                   |
 |     |                    1                          |
 |     |                    4                          |
 |     |                    2                          |
 |     | expenses                                      |
 |     |     food          | 2000                      |
 |     |     gas           | 10                        |
 |     |     travel        | 700                       |
 |     | firstname         | John                      |
 |     | income            | 100000000                 |
 |     | lastLogin         | 2016-11-29T08:21:35.4971  |
 |     | lastname          | Morgan                    |
 +-----+-----------------------------------------------+
 |   6 | myarray                                       |
 |     |                    50                         |
 |     |                    1                          |
 |     |                    2                          |
 |     |                    3                          |
 |     |                    4                          |
```

```
|    |                    100                    |
|    |                    400                    |
|    |                    66                     |
|    |                    77                     |
|    |                    88                     |
|    | mynumber         | 300                    |
+----+-------------------------------------------+

2 rows returned
sql->
```

These two rows look nothing alike. Row 3 contains information about a person, while row 6 contains, essentially, random data. This is possible because the `person` column is of type JSON, which is not strongly typed. But because we interact with JSON columns as if they are maps, we can fix row 6 by modifying it as a map.

Removing Elements from a Map

To begin, we remove the two existing elements from row six (`myarray` and `mynumber`). We do this with a single UPDATE statement, which allows us to execute multiple update clauses so long as they are comma-separated:

```
sql-> UPDATE JSONPersons j
          REMOVE j.person.myarray,
          REMOVE j.person.mynumber
          WHERE j.id = 6
          RETURNING *;
 +----+-----------------+
 | id |     person      |
 +----+-----------------+
 |  6 |                 |
 +----+-----------------+

1 row returned
sql->
```

Adding Elements to a Map

Next, we add person data to this table row. We could do this with a single UPDATE statement by specifying the entire map with a single PUT clause, but for illustration purposes we do this in multiple steps.

To begin, we specify the person's name. Here, we use a single PUT clause that specifies a map with multiple elements:

```
sql-> UPDATE JSONPersons j
          PUT j.person {"firstname" : "Wendy",
                        "lastname" : "Purvis"}
          WHERE j.id = 6
          RETURNING *;
 +----+--------------------+
 | id |       person       |
 +----+--------------------+
 |  6 | firstname | Wendy  |
```

```
 |    | lastname  | Purvis |
 +----+-------------------+

1 row returned
sql->
```

Next, we specify the age, connections, expenses, income, and lastLogin fields using multiple PUT clauses on a single UPDATE statement:

```
sql-> UPDATE JSONPersons j
          PUT j.person {"age" : 43},
          PUT j.person {"connections" : [2,3]},
          PUT j.person {"expenses" : {"food" : 1100,
                                      "books" : 210,
                                      "travel" : 50}},
          PUT j.person {"income" : 80000},
          PUT j.person {"lastLogin" : "2017-06-29T16:12:35.0285"}
          WHERE j.id = 6
          RETURNING *;
 +----+---------------------------------------+
 | id |                person                 |
 +----+---------------------------------------+
 |  6 | age         | 43                      |
 |    | connections                           |
 |    |               2                       |
 |    |               3                       |
 |    | expenses                              |
 |    |     books   | 210                     |
 |    |     food    | 1100                    |
 |    |     travel  | 50                      |
 |    | firstname   | Wendy                   |
 |    | income      | 80000                   |
 |    | lastLogin   | 2017-06-29T16:12:35.0285 |
 |    | lastname    | Purvis                  |
 +----+---------------------------------------+

1 row returned
sql->
```

We still need an address. Again, we could do this with a single PUT clause, but for illustration purposes we will use multiple clauses. Our first PUT creates the `address` element, which uses a map as a value. Our second PUT adds elements to the `address` map:

```
sql-> UPDATE JSONPersons j
        PUT j.person {"address" : {"street" : "479 South Way Dr"}},
        PUT j.person.address {"city" : "St. Petersburg",
                              "state" : "FL"}
        WHERE j.id = 6
        RETURNING *;
 +----+---------------------------------------+
 | id |                person                 |
 +----+---------------------------------------+
 |  6 | address                               |
 |    |     city    | St. Petersburg          |
 |    |     state   | FL                      |
```

```
 |     |     street  | 479 South Way Dr          |
 |     | age         | 43                        |
 |     | connections                             |
 |     |               2                         |
 |     |               3                         |
 |     | expenses                                |
 |     |     books   | 210                       |
 |     |     food    | 1100                      |
 |     |     travel  | 50                        |
 |     | firstname   | Wendy                     |
 |     | income      | 80000                     |
 |     | lastLogin   | 2017-06-29T16:12:35.0285 |
 |     | lastname    | Purvis                    |
 +-----+-------------------------------------------+

1 row returned
sql->
```

Finally, we provide phone numbers for this person. These are specified as an array of maps:

```
sql-> UPDATE JSONPersons j
          PUT j.person.address {"phones" :
                 [{"type":"work", "areacode":727, "number":8284321},
                  {"type":"home", "areacode":727, "number":5710076},
                  {"type":"mobile", "areacode":727, "number":8913080}
                 ]
                                 }
          WHERE j.id = 6
          RETURNING *;
 +----+-------------------------------------------+
 | id |                  person                   |
 +----+-------------------------------------------+
 |  6 | address                                   |
 |    |     city         | St. Petersburg         |
 |    |     phones                                |
 |    |         areacode | 727                    |
 |    |         number   | 8284321                |
 |    |         type     | work                   |
 |    |                                           |
 |    |         areacode | 727                    |
 |    |         number   | 5710076                |
 |    |         type     | home                   |
 |    |                                           |
 |    |         areacode | 727                    |
 |    |         number   | 8913080                |
 |    |         type     | mobile                 |
 |    |     state        | FL                     |
 |    |     street       | 479 South Way Dr       |
 |    | age              | 43                     |
 |    | connections                               |
 |    |                    2                      |
 |    |                    3                      |
 |    | expenses                                  |
 |    |     books        | 210                    |
 |    |     food         | 1100                   |
```

```
 |     |     travel       | 50                        |
 |     | firstname        | Wendy                     |
 |     | income           | 80000                     |
 |     | lastLogin        | 2017-06-29T16:12:35.0285  |
 |     | lastname         | Purvis                    |
 +-----+-------------------------------------------+

1 row returned
sql->
```

Updating Existing Map Elements

To update an existing element in a map, you can use the PUT clause in exactly the same way as you add a new element to map. For example, to update the lastLogin time:

```
sql-> UPDATE JSONPersons j
          PUT j.person {"lastLogin" : "2017-06-29T20:36:04.9661"}
          WHERE j.id = 6
          RETURNING *;
 +-----+-------------------------------------------+
 | id |                  person                    |
 +-----+-------------------------------------------+
 |  6 | address                                    |
 |     |     city         | St. Petersburg         |
 |     |     phones                                |
 |     |         areacode | 727                    |
 |     |         number   | 8284321                |
 |     |         type     | work                   |
 |     |                                           |
 |     |         areacode | 727                    |
 |     |         number   | 5710076                |
 |     |         type     | home                   |
 |     |                                           |
 |     |         areacode | 727                    |
 |     |         number   | 8913080                |
 |     |         type     | mobile                 |
 |     |     state        | FL                     |
 |     |     street       | 479 South Way Dr       |
 |     | age              | 43                     |
 |     | connections                               |
 |     |                    2                      |
 |     |                    3                      |
 |     | expenses                                  |
 |     |     books        | 210                    |
 |     |     food         | 1100                   |
 |     |     travel       | 50                     |
 |     | firstname        | Wendy                  |
 |     | income           | 80000                  |
 |     | lastLogin        | 2017-06-29T20:36:04.9661 |
 |     | lastname         | Purvis                 |
 +-----+-------------------------------------------+

1 row returned
sql->
```

Alternatively, use a SET clause:

```
sql-> UPDATE JSONPersons j
          SET j.person.lastLogin = "2017-06-29T20:38:56.2751"
          WHERE j.id = 6
          RETURNING *;
 +----+-------------------------------------------+
 | id |                  person                   |
 +----+-------------------------------------------+
 |  6 | address                                   |
 |    |     city         | St. Petersburg         |
 |    |     phones                                |
 |    |         areacode | 727                    |
 |    |         number   | 8284321                |
 |    |         type     | work                   |
 |    |                                           |
 |    |         areacode | 727                    |
 |    |         number   | 5710076                |
 |    |         type     | home                   |
 |    |                                           |
 |    |         areacode | 727                    |
 |    |         number   | 8913080                |
 |    |         type     | mobile                 |
 |    |     state        | FL                     |
 |    |     street       | 479 South Way Dr       |
 |    | age              | 43                     |
 |    | connections                               |
 |    |                    2                      |
 |    |                    3                      |
 |    | expenses                                  |
 |    |     books        | 210                    |
 |    |     food         | 1100                   |
 |    |     travel       | 50                     |
 |    | firstname        | Wendy                  |
 |    | income           | 80000                  |
 |    | lastLogin        | 2017-06-29T20:38:56.2751 |
 |    | lastname         | Purvis                 |
 +----+-------------------------------------------+

1 row returned
sql->
```

If you want to set the timestamp to the current time, use the `current_time()` built-in function.

```
sql-> UPDATE JSONPersons j
          SET j.person.lastLogin = cast(current_time() AS String)
          WHERE j.id = 6
          RETURNING *;
 +----+-------------------------------------------+
 | id |                  person                   |
 +----+-------------------------------------------+
 |  6 | address                                   |
 |    |     city         | St. Petersburg         |
 |    |     phones                                |
 |    |         areacode | 727                    |
```

```
 |    |         number   | 8284321                 |
 |    |         type     | work                    |
 |    |                                            |
 |    |         areacode | 727                     |
 |    |         number   | 5710076                 |
 |    |         type     | home                    |
 |    |                                            |
 |    |         areacode | 727                     |
 |    |         number   | 8913080                 |
 |    |         type     | mobile                  |
 |    |     state        | FL                      |
 |    |     street       | 479 South Way Dr        |
 |    | age              | 43                      |
 |    | connections                                |
 |    |                    2                       |
 |    |                    3                       |
 |    | expenses                                   |
 |    |     books        | 210                     |
 |    |     food         | 1100                    |
 |    |     travel       | 50                      |
 |    | firstname        | Wendy                   |
 |    | income           | 80000                   |
 |    | lastLogin        | 2017-06-29T04:40:15.917 |
 |    | lastname         | Purvis                  |
 +----+-----------------------------------------+

1 row returned
sql->
```

If an element in the map is an array, you can modify it in the same way as you would any array. For example:

```
sql-> UPDATE JSONPersons j
          ADD j.person.connections seq_concat(1, 4)
          WHERE j.id = 6
          RETURNING *;
 +----+-----------------------------------------+
 | id |                  person                 |
 +----+-----------------------------------------+
 |  6 | address                                 |
 |    |     city         | St. Petersburg       |
 |    |     phones                              |
 |    |         areacode | 727                  |
 |    |         number   | 8284321              |
 |    |         type     | work                 |
 |    |                                         |
 |    |         areacode | 727                  |
 |    |         number   | 5710076              |
 |    |         type     | home                 |
 |    |                                         |
 |    |         areacode | 727                  |
 |    |         number   | 8913080              |
 |    |         type     | mobile               |
 |    |     state        | FL                   |
 |    |     street       | 479 South Way Dr     |
```

```
 |     | age              | 43                          |
 |     | connections                                    |
 |     |                    2                           |
 |     |                    3                           |
 |     |                    1                           |
 |     |                    4                           |
 |     | expenses                                       |
 |     |     books        | 210                         |
 |     |     food         | 1100                        |
 |     |     travel       | 50                          |
 |     | firstname        | Wendy                       |
 |     | income           | 80000                       |
 |     | lastLogin        | 2017-06-29T04:40:15.917     |
 |     | lastname         | Purvis                      |
 +-----+------------------------------------------------+

1 row returned
```

If you are unsure of an element being an array or a map, you can use both ADD and PUT within the same UPDATE statement. For example:

```
sql-> UPDATE JSONPersons j
          ADD j.person.connections seq_concat(5, 7),
          PUT j.person.connections seq_concat(5, 7)
          WHERE j.id = 6
          RETURNING *;
 +-----+------------------------------------------------+
 | id |                   person                       |
 +-----+------------------------------------------------+
 |  6 | address                                        |
 |     |     city         | St. Petersburg              |
 |     |     phones                                     |
 |     |         areacode | 727                         |
 |     |         number   | 8284321                     |
 |     |         type     | work                        |
 |     |                                                |
 |     |         areacode | 727                         |
 |     |         number   | 5710076                     |
 |     |         type     | home                        |
 |     |                                                |
 |     |         areacode | 727                         |
 |     |         number   | 8913080                     |
 |     |         type     | mobile                      |
 |     |     state        | FL                          |
 |     |     street       | 479 South Way Dr            |
 |     | age              | 43                          |
 |     | connections                                    |
 |     |                    2                           |
 |     |                    3                           |
 |     |                    1                           |
 |     |                    4                           |
 |     |                    5                           |
 |     |                    7                           |
 |     | expenses                                       |
 |     |     books        | 210                         |
```

```
 |    |     food          | 1100                       |
 |    |     travel        | 50                         |
 |    | firstname         | Wendy                      |
 |    | income            | 80000                      |
 |    | lastLogin         | 2017-06-29T04:40:15.917    |
 |    | lastname          | Purvis                     |
 +----+-------------------------------------------------+

1 row returned
```

If the element is an array, the ADD gets applied and the PUT is a noop. If it is a map, then the PUT gets applied and ADD is a noop. In this example, since the element is an array, the ADD gets applied.

Managing Time to Live Values

Time to Live (TTL) values indicate how long data can exist in a table before it expires. Expired data can no longer be returned as part of a query.

Default TTL values can be set on either a table-level or a row level when the table is first defined. Using UPDATE statements, you can change the TTL value for a single row.

You can see a row's TTL value using the `remaining_hours()`, `remaining_days()` or `expiration_time()` built-in functions. These TTL functions require a row as input. We accomplish this by using the $ as part of the table alias. This causes the table alias to function as a row variable.

```
sql-> SELECT remaining_days($j) AS Expires
    FROM JSONPersons $j WHERE id = 6;
 +---------+
 | Expires |
 +---------+
 |      -1 |
 +---------+

1 row returned
sql->
```

The previous query returns `-1`. This means that the row has no expiration time. We can specify an expiration time for the row by using an UPDATE statement with a `set TTL` clause. This clause computes a new TTL by specifying an offset from the current expiration time. If the row never expires, then the current expiration time is `1970-01-01T00:00:00.000`. The value you provide to `set TTL` must specify units of either `HOURS` or `DAYS`.

```
sql-> UPDATE JSONPersons $j
          SET TTL 1 DAYS
          WHERE id = 6
          RETURNING remaining_days($j) AS Expires;
 +---------+
 | Expires |
 +---------+
 |       1 |
 +---------+
```

```
1 row returned
sql->
```

To see the new expiration time, we can use the built-in `expiration_time()` function. Because we specified an expiration time based on a day boundary, the row expires at midnight of the following day (expiration rounds up):

```
sql-> SELECT current_time() AS Now,
    expiration_time($j) AS Expires
    FROM JSONPersons $j WHERE id = 6;
 +-------------------------+-------------------------+
 |           Now           |         Expires         |
 +-------------------------+-------------------------+
 | 2017-07-03T21:56:47.778 | 2017-07-05T00:00:00.000 |
 +-------------------------+-------------------------+

1 row returned
sql->
```

To turn off the TTL so that the row will never expire, specify a negative value, using either `HOURS` or `DAYS` as the unit:

```
sql-> UPDATE JSONPersons $j
          SET TTL -1 DAYS
          WHERE id = 6
          RETURNING remaining_days($j) AS Expires;
 +---------+
 | Expires |
 +---------+
 |       0 |
 +---------+

1 row returned
sql->
```

Notice that the RETURNING clause provides a value of 0 days. This indicates that the row will never expire. Further, if we look at the remaining_days() using a SELECT statement, we will once again see a negative value, indicating that the row never expires:

```
sql-> SELECT remaining_days($j) AS Expires
    FROM JSONPersons $j WHERE id = 6;
 +---------+
 | Expires |
 +---------+
 |      -1 |
 +---------+

1 row returned
sql->
```

Avoiding the Read-Modify-Write Cycle

An important aspect of UPDATE Statements is that you do not have to read a value in order to update it. Instead, you can blindly modify a value directly in the store without ever retrieving (reading) it. To do this, you refer to the value you want to modify using the `$` variable.

For example, we have a row in JSONPersons that looks like this:

```
sql-> SELECT * FROM JSONPersons WHERE id=6;
 +----+-------------------------------------------+
 | id |                  person                   |
 +----+-------------------------------------------+
 |  6 | address                                   |
 |    |     city          | St. Petersburg        |
 |    |     phones                                |
 |    |         areacode  | 727                   |
 |    |         number    | 8284321               |
 |    |         type      | work                  |
 |    |                                           |
 |    |         areacode  | 727                   |
 |    |         number    | 5710076               |
 |    |         type      | home                  |
 |    |                                           |
 |    |         areacode  | 727                   |
 |    |         number    | 8913080               |
 |    |         type      | mobile                |
 |    |     state         | FL                    |
 |    |     street        | 479 South Way Dr      |
 |    | age               | 43                    |
 |    | connections                               |
 |    |                     2                     |
 |    |                     3                     |
 |    |                     1                     |
 |    |                     4                     |
 |    | expenses                                  |
 |    |     books         | 210                   |
 |    |     food          | 1100                  |
 |    |     travel        | 50                    |
 |    | firstname         | Wendy                 |
 |    | income            | 80000                 |
 |    | lastLogin         | 2017-07-25T22:50:06.482 |
 |    | lastname          | Purvis                |
 +----+-------------------------------------------+

1 row returned
```

We can blindly update the value of the `person.expenses.books` field by referencing `$`. In the following statement, no read is performed on the store. Instead, the write operation is performed directly at the store.

```
sql-> UPDATE JSONPersons j
   ->     SET j.person.expenses.books = $ + 100
   ->     WHERE id = 6;
 +----------------+
```

```
 | NumRowsUpdated |
 +----------------+
 |              1 |
 +----------------+

1 row returned
```

To see that the books expenses value has indeed been incremented by 100, we perform a second `SELECT` statement.

```
sql-> SELECT * FROM JSONPersons WHERE id=6;
 +----+-------------------------------------------+
 | id |                  person                   |
 +----+-------------------------------------------+
 |  6 | address                                   |
 |    |     city         | St. Petersburg         |
 |    |     phones                                |
 |    |         areacode | 727                    |
 |    |         number   | 8284321                |
 |    |         type     | work                   |
 |    |                                           |
 |    |         areacode | 727                    |
 |    |         number   | 5710076                |
 |    |         type     | home                   |
 |    |                                           |
 |    |         areacode | 727                    |
 |    |         number   | 8913080                |
 |    |         type     | mobile                 |
 |    |     state        | FL                     |
 |    |     street       | 479 South Way Dr       |
 |    | age              | 43                     |
 |    | connections                               |
 |    |                    2                      |
 |    |                    3                      |
 |    |                    1                      |
 |    |                    4                      |
 |    | expenses                                  |
 |    |     books        | 310                    |
 |    |     food         | 1100                   |
 |    |     travel       | 50                     |
 |    | firstname        | Wendy                  |
 |    | income           | 80000                  |
 |    | lastLogin        | 2017-07-25T22:50:06.482 |
 |    | lastname         | Purvis                 |
 +----+-------------------------------------------+

1 row returned
```

9
Working with Multi-Region Setup

This chapter provides examples on how to create regions, Multi-Region tables, and use MR_COUNTERs in Multi-Region tables.

A Multi-Region architecture helps you create tables in multiple data stores. Each data store in a Multi-Region Oracle NoSQL Database setup is called a Region. In a Multi-Region setup, Oracle NoSQL Database automatically replicates data across the regions.

Managing Regions

Learn to use the SQL statements to register regions with your local Oracle NoSQL Database and view them.

In a Multi-Region Oracle NoSQL Database setup, you must register all regions, local and remote regions with your local Oracle NoSQL Database. You use the CREATE REGION statement to register a region.

Use the following command to set your local region:

```
SET LOCAL REGION my_local_region;
```

The following CREATE REGION statements register remote regions named LON and FRA.

```
CREATE REGION LON;

CREATE REGION FRA;
```

You can use the SHOW REGIONS statement to view the list of regions present in Oracle NoSQL Database. The following statement fetches all the existing regions in a JSON format. The output shows the local and remote regions. The `state` field indicates if a region is active.

```
SHOW AS JSON REGIONS;
```

Output:

```
{"regions" : [{"name" : "my_local_region", "type" : "local", "state" :
"active"},{"name" : "LON", "type" : "remote", "state" : "active"},{"name" :
"FRA", "type" : "remote", "state" : "active"}]}
```

You can use the DROP REGION statement to remove the registration of a specified remote region from your local Oracle NoSQL Database. The following statement removes the FRA region. The output shows the `state` as `dropped`.

```
DROP REGION FRA;
```

Output:

```
{"regions" : [{"name" : "my_local_region", "type" : "local", "state" :
"active"},{"name" : "LON", "type" : "remote", "state" : "active"},{"name" :
"FRA", "type" : "remote", "state" : "dropped"}]}
```

Using MR_COUNTERs

Learn to use SQL statements to create and manage MR_COUNTERs in Multi-Region tables.

The MR_COUNTER data type is a Conflict-free Replicated Data Type (CRDT) counter. CRDTs provide a way for concurrent modifications to be merged across regions without user intervention.

In a Multi-Region setup of an Oracle NoSQL Database, copies of the same data must be stored in multiple regions and data may be concurrently modified in different regions. The MR_COUNTER data type ensures that though data modifications happen simultaneously on different regions, data always gets automatically merged into a consistent state.

Currently, Oracle NoSQL Database supports only Positive-Negative (PN) MR_COUNTER data type. The PN counters are suitable for increment and decrement operations. For example, you can use these counters to count the number of viewers live streaming a football match from a website at any point. When the viewers go offline, you need to decrement the counter.

You can only define MR_COUNTERs while creating a table or while modifying a table.

Create table using MR_COUNTER data type

You can declare a table column of the MR_COUNTER data type in a CREATE TABLE statement. MR_COUNTER is a subtype of one of the following data types: INTEGER, LONG, NUMBER.

```
CREATE TABLE Users (
  id integer,
  firstname string,
  lastname string,
  age integer,
  income integer,
  count integer AS MR_COUNTER,
  primary key (id)
) IN REGIONS FRA,LON;
```

You can use the MR_COUNTER data type for a Multi-Region table only. You can't use it in regular tables. In the statement above, you create a Multi-Region table in FRA and LON regions with `count` as an INTEGER MR_COUNTER data type. You can define multiple columns as MR_COUNTER data type in a Multi-Region table.

You can also declare a field in a JSON document as MR_COUNTER.

```
CREATE TABLE IF NOT EXISTS JSONPersons (
  id integer,
  person JSON (counter as INTEGER MR_COUNTER,
               books.count as LONG MR_COUNTER),
  primary key (id)
) IN REGIONS FRA,LON;
```

In the statement above, you are identifying two of the fields in the JSON document `person` as MR_COUNTERs. The first field `counter` is an INTEGER MR_COUNTER data type. The second field `count` is within a nested JSON document `books`. The `count` field is of LONG MR_COUNTER data type.

Insert rows into a Multi-Region table

You can use the INSERT statement to insert data into a Multi-Region table with the MR_COUNTER column. You can add rows using one of the following options. Both the options insert a default value of zero to the MR_COUNTER column.

1. **Option 1**: Supply the keyword DEFAULT to the MR_COUNTER column.

   ```
   INSERT INTO Users VALUES (10, "David", "Morrison", 25, 100000,
       DEFAULT);
   ```

   In the statement above, you supply a value DEFAULT to the `count` MR_COUNTER.

   ```
   SELECT * FROM Users;
   ```

   Output:

   ```
   {"id":10,"firstname":"David","lastname":"Morrison","age":25,"income":100000
   ,"count":0}
   ```

2. **Option 2**: Skip the MR_COUNTER column value by including only the required column values in the INSERT statement.

   ```
   INSERT INTO Users(id, firstname, lastname) VALUES (20, "John", "Anderson");
   ```

   In the statement above, you supply values to specific columns. The SQL engine inserts the values to the corresponding columns, a default value zero to the MR_COUNTER, and a null value to all the other columns.

   ```
   SELECT * FROM Users WHERE id = 20;
   ```

   Output:

   ```
   {"id":20,"firstname":"John","lastname":"Anderson","age":null,"income":null,
   "count":0}
   ```

If an MR_COUNTER is a part of the JSON document, you must supply a zero value explicitly to the MR_COUNTER.

> **Note:**
>
> - You can't supply the keyword DEFAULT while inserting a JSON MR_COUNTER.
> - The system will return an error if you try to insert data into an MR table without supplying a value to the declared JSON MR_COUNTER field or using the keyword DEFAULT.

In the sample below, you insert a row into `JSONPersons` table. As it includes JSON MR_COUNTERs `counter` and `count` in the `people` document, you supply a zero value explicitly to these MR_COUNTERs.

```
INSERT INTO JSONPersons VALUES (
   1,
  {
      "firstname":"David",
      "lastname":"Morrison",
      "age":25,
      "income":100000,
      "counter": 0,
      "books" : {
        "Title1" : "Gone with the wind",
        "Title2" : "Oliver Twist",
        "count" : 0
      }
    }
);
```

The SELECT statement displays the following result:

```
{"id":1,"person":{"age":25,"books":{"Title1":"Gone with the
wind","Title2":"Oliver
Twist","count":0},"counter":0,"firstname":"David","income":100000,"lastname":"
Morrison"}};
```

Update MR_COUNTER

You can use the SET clause of the UPDATE statement to update MR_COUNTER in a Multi-Region table. You must only use the standard arithmetic computations to increment or decrement the value of MR_COUNTER. You can't use the UPDATE clauses to explicitly supply a value to MR_COUNTER or remove one from the table.

```
UPDATE Users SET count = count + 10 WHERE id = 10 RETURNING *;
```

In the statement above, you increment the `count` value in the `Users` table by 10. The RETURNING clause fetches the following output:

```
{"id":10,"firstname":"David","lastname":"Morrison","age":25,"income":100000,"c
ount":10}
```

Similarly, you can update MR_COUNTER in a JSON document by incrementing or decrementing its value. You can access MR_COUNTER using its path expression as follows:

```
UPDATE JSONPersons p SET p.person.books.count = p.person.books.count + 1
WHERE id = 1 RETURNING *;
```

In the statement above, you increment the MR_COUNTER `count` in the nested books document by one.

```
{"id":1,"person":{"age":25,"books":{"Title1":"Gone with the
wind","Title2":"Oliver
```

```
Twist","count":1},"counter":0,"firstname":"David","income":100000,"lastname":"
Morrison"}}
```

How system uses MR_COUNTER to handle concurrent modifications

When you create a Multi-Region table in different regions, it has the same definition. This implies, if you define any MR_COUNTER data type, it exists in both the remote and local regions. Every region can update the MR_COUNTER concurrently at its end. As all the Multi-Region tables in the participating regions are synchronized, the system automatically performs a merge on these concurrent modifications to reflect the latest updates of the MR_COUNTER without any user intervention.

Modify table to add or remove MR_COUNTER

You can use an ALTER TABLE statement to add or remove MR_COUNTER.

Adding MR_COUNTER

To add MR_COUNTER, use the ADD clause in the ALTER TABLE statement.

```
ALTER TABLE Users (ADD countTwo INTEGER AS MR_COUNTER);
```

The statement above adds `countTwo` field as MR_COUNTER with a default value zero to the `Users` table.

The SELECT statement displays the following result:

```
{"id":10,"firstname":"David","lastname":"Morrison","age":25,"income":100000,"c
ount":10,"countTwo":0}
{"id":20,"firstname":"John","lastname":"Anderson","age":null,"income":null,"co
unt":0,"countTwo":0}
```

You can add MR_COUNTER to a JSON column as follows:

```
ALTER TABLE JSONPersons (ADD JsonTwo JSON(counterTwo AS NUMBER MR_COUNTER));
```

The statement above adds a `JsonTwo` nested JSON document to the `JSONPersons` table and includes `counterTwo` field as MR_COUNTER with zero value:

```
{
  "id" : 1,
  "person" : {
    "age" : 25,
    "books" : {
      "Title1" : "Gone with the wind",
      "Title2" : "Oliver Twist",
      "count" : 1
    },
    "counter" : 0,
    "firstname" : "David",
    "income" : 100000,
    "lastname" : "Morrison"
  },
  "JsonTwo" : {
    "counterTwo" : 0
```

```
  }
}
```

Removing MR_COUNTER

To remove MR_COUNTER, use the DROP clause in the ALTER TABLE statement.

```
ALTER TABLE Users (DROP countTwo);
```

The statement above removes `countTwo` MR_COUNTER from the `Users` table.

The SELECT statement displays the following result:

```
{"id":10,"firstname":"David","lastname":"Morrison","age":25,"income":100000,"c
ount":10}
{"id":20,"firstname":"John","lastname":"Anderson","age":null,"income":null,"co
unt":0}
```

You can remove a JSON document and its MR_COUNTER as follows:

```
ALTER TABLE JSONPersons (DROP JsonTwo);
```

The statement above removes the `JSONTwo` nested JSON document from the `JSONPersons` table.

```
{
  "id" : 1,
  "person" : {
    "age" : 25,
    "books" : {
      "Title1" : "Gone with the wind",
      "Title2" : "Oliver Twist",
      "count" : 1
    },
    "counter" : 0,
    "firstname" : "David",
    "income" : 100000,
    "lastname" : "Morrison"
  }
}
```

A

Introduction to the SQL for Oracle NoSQL Database Shell

This appendix describes how to configure, start and use the SQL for Oracle NoSQL Database shell to execute SQL statements. This section also describes the available shell commands.

You can directly execute DDL, DML, user management, security, and informational statements using the SQL shell.

Running the SQL Shell

You can run the SQL shell interactively or use it to run single commands. Here is the general usage to start the shell:

```
java -jar KVHOME/lib/sql.jar
       -helper-hosts <host:port[,host:port]*> -store <storeName>
       [-username <user>] [-security <security-file-path>]
       [-timeout <timeout ms>]
       [-consistency <NONE_REQUIRED(default) |
                          ABSOLUTE | NONE_REQUIRED_NO_MASTER>]
       [-durability <COMMIT_SYNC(default) |
                          COMMIT_NO_SYNC | COMMIT_WRITE_NO_SYNC>]
       [single command and arguments]
```

The following are the mandatory parameters:

`-helper-hosts`: Specifies a comma-separated list of hosts and ports.

`-store`: Specifies the name of the store.

`-security`: Specifies the path to the security file in a secure deployment of the store.

For example: `$KVROOT/security/user.security`

The store supports the following optional parameters:

`-consistency`: Configures the read consistency used for this session. The read operations are serviced either on a master or a replica node depending on the configured value. For more details on consistency, see Consistency Guarantees. The following policies are supported. They are defined in the `Consistency` class of Java APIs.

If you do not specify this value, the default value ABSOLUTE is applied for this session.

- ABSOLUTE - The read operation is serviced on a master node. With ABSOLUTE consistency, you are guaranteed to obtain the latest updated data.
- NONE-REQUIRED - The read operation can be serviced on a replica node. This implies, that if the data is read from the replica node, it may not match what is on the master. However, eventually, it will be consistent with the master.

For more details on the policies, see Consistency in the *Java Direct Driver API Reference* Guide.

`-durability`: Configures the write durability setting used in this session. This value defines the durability policies to be applied for achieving master commit synchronization, that is, the actions performed by the master node to return with a normal status from the write operations. For more details on durability, see Durability Guarantees.

If you do not specify this value, the default value COMMIT_SYNC is applied for this session.

- COMMIT_NO_SYNC - The data is written to the host's in-memory cache, but the master node does not wait for the data to be written to the file system's data buffers or subsequent physical storage.
- COMMIT_SYNC - The data is written to the in-memory cache, transferred to the file system's data buffers, and then synchronized to a stable storage before the write operation completes normally.
- COMMIT_WRITE_NO_SYNC - The data is written to the in-memory cache, and transferred to the file system's data buffers, but not necessarily into physical storage.

For more details on the policies, see Durability in the *Java Direct Driver API Reference* Guide.

`-timeout`: Configures the request timeout used for this session. The default value is 5000ms.

`-username`: Specifies the username to log in as.

For example, you can start the shell like this:

```
java -jar KVHOME/lib/sql.jar
-helper-hosts node01:5000 -store kvstore
sql->
```

This command assumes that a store `kvstore` is running at port `5000`. After the SQL starts successfully, you execute queries. In the next part of this document, you will find an introduction to `SQL for Oracle NoSQL Database` and how to create query statements.

If you want to import records from a file in either JSON or CSV format, you can use the import command. For more information see import.

If you want to run a script, use the `load` command. For more information see load.

```
sql-> command [arguments]
```

`-single command and arguments`: Specifies the utility commands that can be accessed from the SQL shell. You can use them with the syntax shown above.

For a complete list of utility commands accessed through "`java -jar`" `<kvhome>/lib/sql.jar <command>` see Shell Utility Commands.

Configuring the shell

You can also set the shell start-up arguments by modifying the configuration file `.kvclirc` found in your home directory.

Arguments can be configured in the `.kvclirc` file using the `name=value` format. This file is shared by all shells, each having its named section. `[sql]` is used for the Query shell, while `[kvcli]` is used for the Admin Command Line Interface (CLI).

For example, the `.kvclirc` file would then contain content like this:

```
[sql]
helper-hosts=node01:5000
store=kvstore
timeout=10000
consistency=NONE_REQUIRED
durability=COMMIT_NO_SYNC
username=root
security=/tmp/login_root

[kvcli]
host=node01
port=5000
store=kvstore
admin-host=node01
admin-port=5001
username=user1
security=/tmp/login_user
admin-username=root
admin-security=/tmp/login_root
timeout=10000
consistency=NONE_REQUIRED
durability=COMMIT_NO_SYNC
```

Shell Utility Commands

The following sections describe the utility commands accessed through "`java -jar` `<kvhome>/lib/sql.jar <command>`".

The interactive prompt for the shell is:

```
sql->
```

The shell comprises a number of commands. All commands accept the following flags:

- -help

  Displays online help for the command.

- ?

  Synonymous with -help. Displays online help for the command.

The shell commands have the following general format:

1. All commands are structured like this:

   ```
   sql-> command [arguments]
   ```

2. All arguments are specified using flags that start with "-"
3. Commands and subcommands are case-insensitive and match on partial strings(prefixes) if possible. The arguments, however, are case-sensitive.

connect

```
connect -host <hostname> -port <port> -name <storeName>
[-timeout <timeout ms>]
[-consistency <NONE_REQUIRED(default) |
                               ABSOLUTE | NONE_REQUIRED_NO_MASTER>]
[-durability <COMMIT_SYNC(default) |
                               COMMIT_NO_SYNC | COMMIT_WRITE_NO_SYNC>]
[-username <user>] [-security <security-file-path>]
```

Connects to a KVStore to perform data access functions. If the instance is secured, you may need to provide login credentials.

consistency

```
consistency [[NONE_REQUIRED | NONE_REQUIRED_NO_MASTER |
ABSOLUTE] [-time -permissible-lag <time_ms> -timeout <time_ms>]]
```

Configures the read consistency used for this session.

describe

```
describe | desc [as json]
 {table table_name [field_name[,...] ] |
 index index_name on table_name
 }
```

Describes information about a table or index, optionally in JSON format.

Specify a fully-qualified *table_name* as follows:

Entry specification	Description
table_name	Required. Specifies the full table name. Without further qualification, this entry indicates a table created in the default namespace (sysdefault), which you do not have to specify.
parent-table.child-table	Specifies a child table of a parent. Specify the parent table followed by a period (.) before the child name. For example, if the parent table is `Users`, specify the child table named `MailingAddress` as `Users.MailingAddress`.
namespace-name:table-name	Specifies a table created in the non-default namespace. Use the namespace followed by a colon (:). For example, to reference table `Users`, created in the `Sales` namespace, enter *table_name* as `Sales:Users`.

Following is the output of `describe` for table `ns1:t1`:

```
sql-> describe table ns1:t1;
 === Information ===
 +-----------+------+-----+-------+----------+----------+--------+----------
+---------+-------------+
 | namespace | name | ttl | owner | sysTable | r2compat | parent | children |
indexes | description |
 +-----------+------+-----+-------+----------+----------+--------+----------
+---------+-------------+
 | ns1       | t1   |     |       | N        | N        |        |
|         |             |
 +-----------+------+-----+-------+----------+----------+--------+----------
+---------+-------------+

 === Fields ===
 +----+------+---------+----------+-----------+----------+------------
+-----------+
 | id | name |  type   | nullable |  default  | shardKey | primaryKey |
identity |
 +----+------+---------+----------+-----------+----------+------------
+-----------+
 |  1 | id   | Integer | N        | NullValue | Y        | Y
|           |
 +----+------+---------+----------+-----------+----------+------------
+-----------+
 |  2 | name | String  | Y        | NullValue |          |
|           |
 +----+------+---------+----------+-----------+----------+------------
+-----------+

sql->
```

This example shows using `describe as json` for the same table:

```
sql-> describe as json table ns1:t1;
{
  "json_version" : 1,
  "type" : "table",
  "name" : "t1",
  "namespace" : "ns1",
  "shardKey" : [ "id" ],
  "primaryKey" : [ "id" ],
  "fields" : [ {
    "name" : "id",
    "type" : "INTEGER",
    "nullable" : false,
    "default" : null
  }, {
    "name" : "name",
    "type" : "STRING",
    "nullable" : true,
    "default" : null
  } ]
}
```

durability

```
durability [[COMMIT_WRITE_NO_SYNC | COMMIT_SYNC |
COMMIT_NO_SYNC] | [-master-sync <sync-policy> -replica-sync <sync-policy>
-replica-ask <ack-policy>]] <sync-policy>: SYNC, NO_SYNC, WRITE_NO_SYNC
<ack-policy>: ALL, NONE, SIMPLE_MAJORITY
```

Configures the write durability used for this session.

exit

```
exit | quit
```

Exits the interactive command shell.

help

```
help [command]
```

Displays help message for all shell commands and sql command.

history

```
history [-last <n>] [-from <n>] [-to <n>]
```

Displays command history. By default all history is displayed. Optional flags are used to choose ranges for display.

import

```
import -table table_name -file file_name [JSON | CSV]
```

Imports records from the specified file into table *table_name*.

Specify a fully-qualified *table_name* as follows:

Entry specification	Description
table_name	Required. Specifies the full table name. Without further qualification, this entry indicates a table created in the default namespace (sysdefault), which you do not have to specify.
parent-table.child-table	Specifies a child table of a parent. Specify the parent table followed by a period (.) before the child name. For example, if the parent table is `Users`, specify the child table named `MailingAddress` as `Users.MailingAddress`.

Entry specification	Description
namespace-name:table-name	Specifies a table created in the non-default namespace. Use the namespace followed by a colon (:). For example, to reference table `Users`, created in the `Sales` namespace, enter *table_name* as `Sales:Users`.

Use `-table` to specify the name of a table into which the records are loaded. The alternative way to specify the table is to add the table specification "Table: *table_name*" before its records in the file.

For example, this file contains the records to insert into two tables, `users` and `email`:

```
Table: users
<records of users>
...
Table: emails
<record of emails>
...
```

The imported records can be either in JSON or CSV format. If you do not specify the format, JSON is assumed.

load

```
load -file <path to file>
```

Load the named file and interpret its contents as a script of commands to be executed. If any command in the script fails execution will end.

For example, suppose the following commands are collected in the script file `test.sql`:

```
### Begin Script ###
load -file test.ddl
import -table users -file users.json
### End Script ###
```

Where the file `test.ddl` would contain content like this:

```
DROP TABLE IF EXISTS users;
CREATE TABLE users(id INTEGER, firstname STRING, lastname STRING,
age INTEGER, primary key (id));
```

And the file `users.json` would contain content like this:

```
{"id":1,"firstname":"Dean","lastname":"Morrison","age":51}
{"id":2,"firstname":"Idona","lastname":"Roman","age":36}
{"id":3,"firstname":"Bruno","lastname":"Nunez","age":49}
```

Then, the script can be run by using the `load` command in the shell:

```
> java -jar KVHOME/lib/sql.jar -helper-hosts node01:5000 \
-store kvstore
sql-> load -file ./test.sql
Statement completed successfully.
Statement completed successfully.
Loaded 3 rows to users.
```

mode

```
mode [COLUMN | LINE | JSON [-pretty] | CSV]
```

Sets the output mode of query results. The default value is JSON.

For example, a table shown in COLUMN mode:

```
sql-> mode column;
sql-> SELECT * from users;
 +-----+-----------+-----------+-----+
 | id  | firstname | lastname  | age |
 +-----+-----------+-----------+-----+
 |   8 | Len       | Aguirre   |  42 |
 |  10 | Montana   | Maldonado |  40 |
 |  24 | Chandler  | Oneal     |  25 |
 |  30 | Pascale   | Mcdonald  |  35 |
 |  34 | Xanthus   | Jensen    |  55 |
 |  35 | Ursula    | Dudley    |  32 |
 |  39 | Alan      | Chang     |  40 |
 |   6 | Lionel    | Church    |  30 |
 |  25 | Alyssa    | Guerrero  |  43 |
 |  33 | Gannon    | Bray      |  24 |
 |  48 | Ramona    | Bass      |  43 |
 |  76 | Maxwell   | Mcleod    |  26 |
 |  82 | Regina    | Tillman   |  58 |
 |  96 | Iola      | Herring   |  31 |
 | 100 | Keane     | Sherman   |  23 |
 +-----+-----------+-----------+-----+
 ...

100 rows returned
```

Empty strings are displayed as an empty cell.

```
sql-> mode column;
sql-> SELECT * from tab1 where id = 1;
 +----+------+----+------+
 | id |  s1  | s2 |  s3  |
 +----+------+----+------+
 |  1 | NULL |    | NULL |
 +----+------+----+------+

1 row returned
```

For nested tables, identation is used to indicate the nesting under column mode:

```
sql-> SELECT * from nested;
+----+-------+-----------------------------------------------------------+
| id | name  |                          details                          |
+----+-------+-----------------------------------------------------------+
|  1 | one   | address                                                   |
|    |       |     city    | Waitakere                                   |
|    |       |     country | French Guiana                               |
|    |       |     zipcode | 7229                                        |
|    |       | attributes                                                |
|    |       |     color   | blue                                        |
|    |       |     price   | expensive                                   |
|    |       |     size    | large                                       |
|    |       | phone       | [(08)2435-0742, (09)8083-8862, (08)0742-2526]|
+----+-------+-----------------------------------------------------------+
|  3 | three | address                                                   |
|    |       |     city    | Viddalba                                    |
|    |       |     country | Bhutan                                      |
|    |       |     zipcode | 280071                                      |
|    |       | attributes                                                |
|    |       |     color   | blue                                        |
|    |       |     price   | cheap                                       |
|    |       |     size    | small                                       |
|    |       | phone       | [(08)5361-2051, (03)5502-9721, (09)7962-8693]|
+----+-------+-----------------------------------------------------------+
...
```

For example, a table shown in LINE mode, where the result is displayed vertically and one value is shown per line:

```
sql-> mode line;
sql-> SELECT * from users;

 > Row 1
 +-----------+-----------+
 | id        | 8         |
 | firstname | Len       |
 | lastname  | Aguirre   |
 | age       | 42        |
 +-----------+-----------+

  > Row 2
 +-----------+-----------+
 | id        | 10        |
 | firstname | Montana   |
 | lastname  | Maldonado |
 | age       | 40        |
 +-----------+-----------+

  > Row 3
 +-----------+-----------+
 | id        | 24        |
 | firstname | Chandler  |
 | lastname  | Oneal     |
```

```
 | age       | 25        |
 +-----------+-----------+
 ...
100 rows returned
```

Just as in COLUMN mode, empty strings are displayed as an empty cell:

```
sql-> mode line;
sql-> SELECT * from tab1 where id = 1;

 > Row 1
 +---------+------+
 | id      | 1    |
 | s1      | NULL |
 | s2      |      |
 | s3      | NULL |
 +---------+------+

1 row returned
```

For example, a table shown in JSON mode:

```
sql-> mode json;
sql-> SELECT * from users;
{"id":8,"firstname":"Len","lastname":"Aguirre","age":42}
{"id":10,"firstname":"Montana","lastname":"Maldonado","age":40}
{"id":24,"firstname":"Chandler","lastname":"Oneal","age":25}
{"id":30,"firstname":"Pascale","lastname":"Mcdonald","age":35}
{"id":34,"firstname":"Xanthus","lastname":"Jensen","age":55}
{"id":35,"firstname":"Ursula","lastname":"Dudley","age":32}
{"id":39,"firstname":"Alan","lastname":"Chang","age":40}
{"id":6,"firstname":"Lionel","lastname":"Church","age":30}
{"id":25,"firstname":"Alyssa","lastname":"Guerrero","age":43}
{"id":33,"firstname":"Gannon","lastname":"Bray","age":24}
{"id":48,"firstname":"Ramona","lastname":"Bass","age":43}
{"id":76,"firstname":"Maxwell","lastname":"Mcleod","age":26}
{"id":82,"firstname":"Regina","lastname":"Tillman","age":58}
{"id":96,"firstname":"Iola","lastname":"Herring","age":31}
{"id":100,"firstname":"Keane","lastname":"Sherman","age":23}
{"id":3,"firstname":"Bruno","lastname":"Nunez","age":49}
{"id":14,"firstname":"Thomas","lastname":"Wallace","age":48}
{"id":41,"firstname":"Vivien","lastname":"Hahn","age":47}
...
100 rows returned
```

Empty strings are displayed as "".

```
sql-> mode json;
sql-> SELECT * from tab1 where id = 1;
{"id":1,"s1":null,"s2":"","s3":"NULL"}

1 row returned
```

Finally, a table shown in CSV mode:

```
sql-> mode csv;
sql-> SELECT * from users;
8,Len,Aguirre,42
10,Montana,Maldonado,40
24,Chandler,Oneal,25
30,Pascale,Mcdonald,35
34,Xanthus,Jensen,55
35,Ursula,Dudley,32
39,Alan,Chang,40
6,Lionel,Church,30
25,Alyssa,Guerrero,43
33,Gannon,Bray,24
48,Ramona,Bass,43
76,Maxwell,Mcleod,26
82,Regina,Tillman,58
96,Iola,Herring,31
100,Keane,Sherman,23
3,Bruno,Nunez,49
14,Thomas,Wallace,48
41,Vivien,Hahn,47
...
100 rows returned
```

Like in JSON mode, empty strings are displayed as "".

```
sql-> mode csv;
sql-> SELECT * from tab1 where id = 1;
1,NULL,"","NULL"

1 row returned
```

> **Note:**
>
> Only rows that contain simple type values can be displayed in CSV format. Nested values are not supported.

output

```
output [stdout | file]
```

Enables or disables output of query results to a file. If no argument is specified, it shows the current output.

page

```
page [on | <n> | off]
```

Turns query output paging on or off. If specified, `n` is used as the page height.

If `n` is 0, or "on" is specified, the default page height is used. Setting `n` to "off" turns paging off.

show faults

```
show faults [-last] [-command <index>]
```

Encapsulates commands that display the state of the store and its components.

show ddl

```
show ddl <table>
```

The `show ddl` query retrieves the DDL statement for a specified table. If the table has indexes, the statement returns the DDLs for the table and the indexes.

Example : Fetch the DDL for a specified table.

The following statement fetches the DDL for the `BaggageInfo` table.

```
show ddl BaggageInfo;
```

Output:

```
CREATE TABLE IF NOT EXISTS BaggageInfo (ticketNo LONG, fullName STRING,
gender STRING,
      contactPhone STRING, confNo STRING, bagInfo JSON, PRIMARY
    KEY(SHARD(ticketNo)))
```

In the following example, the `fixedschema_contact` index exists in the `BaggageInfo` table. The statement retrieves the DDLs for the `BaggageInfo` table and `fixedschema_contact` index on the table.

```
show ddl BaggageInfo;
```

Output:

```
CREATE TABLE IF NOT EXISTS BaggageInfo (ticketNo LONG, fullName STRING,
gender STRING,
        contactPhone STRING, confNo STRING, bagInfo JSON, PRIMARY
      KEY(SHARD(ticketNo)))CREATE INDEX IF NOT EXISTS fixedschema_contact ON
    BaggageInfo(contactPhone)
```

show indexes

```
show_indexes_statement ::= SHOW [AS JSON] INDEXES ON table_name
```

The `show indexes` statement provides the list of indexes present on a specified table. The parameter `AS JSON` is optional and can be specified if you want the output to be in JSON format.

Example 1: List indexes on the specified table

The following statement lists the indexes present on the `users2` table.

```
SHOW INDEXES ON users2;
indexes
   idx1
```

Example 2: List indexes on the specified table in JSON format

The following statement lists the indexes present on the `users2` table in JSON format.

```
SHOW AS JSON INDEXES ON users2;
{"indexes" :
     ["idx1"]
}
```

show namespaces

```
show [AS JSON] namespaces
```

Shows a list of all namespaces in the system.

For example:

```
sql-> show namespaces
namespaces
  ns1
  sysdefault
sql-> show as json namespaces
{"namespaces" : ["ns1","sysdefault"]}
```

show query

```
show query <statement>
```

Displays the query plan for a query.

For example:

```
sql-> show query SELECT * from Users;
RECV([6], 0, 1, 2, 3, 4)
[
  DistributionKind : ALL_PARTITIONS,
  Number of Registers :7,
  Number of Iterators :12,
  SFW([6], 0, 1, 2, 3, 4)
```

```
  [
    FROM:
    BASE_TABLE([5], 0, 1, 2, 3, 4)
    [Users via primary index] as $$Users

    SELECT:
    *
  ]
]
```

show regions

```
show_regions_statement ::= SHOW [AS JSON] REGIONS
```

The `show regions` statement provides the list of regions present in a multi-region Oracle NoSQL Database setup. The parameter `AS JSON` is optional and can be specified if you want the output to be in JSON format.

Example 1: Fetching all regions in a multi-region database setup

```
SHOW REGIONS;
 regions
    my_region1 (remote, active)
    my_region2 (remote, active)
```

Example 2: Fetching all regions in a multi-region database setup in JSON format

```
SHOW AS JSON REGIONS;
{"regions" : [
    {"name" : "my_region1", "type" : "remote", "state" : "active"},
    {"name" : "my_region2", "type" : "remote", "state" : "active"}
]}
```

show roles

```
show [as json] roles | role <role_name>
```

Shows either all the roles currently defined for the store, or the named role.

show tables

```
show [as json] {tables | table table_name}
```

Shows either all tables in the data store, or one specific table, *table_name*.

Specify a fully-qualified *`table_name`* as follows:

Entry specification	Description
table_name	Required. Specifies the full table name. Without further qualification, this entry indicates a table created in the default namespace (sysdefault), which you do not have to specify.
parent-table.child-table	Specifies a child table of a parent. Specify the parent table followed by a period (.) before the child name. For example, if the parent table is `Users`, specify the child table named `MailingAddress` as `Users.MailingAddress`.
namespace-name:table-name	Specifies a table created in the non-default namespace. Use the namespace followed by a colon (:). For example, to reference table `Users`, created in the `Sales` namespace, enter *table_name* as `Sales:Users`.

The following example indicates how to list all tables, or just one table. The empty `tableHierarchy` field indicates that table `t1` was created in the default namespace:

```
sql-> show tables
tables
  SYS$IndexStatsLease
  SYS$PartitionStatsLease
  SYS$SGAttributesTable
  SYS$TableStatsIndex
  SYS$TableStatsPartition
  ns10:t10
  parent
  parent.child
  sg1
  t1

sql-> show table t1
tableHierarchy
  t1
```

To show a table created in a namespace, as shown in the list of all tables, fully-qualify *table_name* as follows. In this case, `tableHierarchy` field lists namespace `ns1` in which table `t1` was created. The example also shows how the table is presented as json:

```
sql-> show tables;
tables
  SYS$IndexStatsLease
  SYS$PartitionStatsLease
  SYS$SGAttributesTable
  SYS$TableStatsIndex
  SYS$TableStatsPartition
  ns1:foo
  ns1:t1

sql-> show table ns1:t1;
tableHierarchy(namespace ns1)
  t1
```

```
sql-> show as json table ns1:t1;
{"namespace": "ns1"
"tableHierarchy" : ["t1"]}
```

show users

```
show [as json] users | user <user_name>
```

Shows either all the users currently existing in the store, or the named user.

timeout

```
timeout [<timeout_ms>]
```

The `timeout` command configures or displays the request timeout for this session in milliseconds(ms).

The request timeout is the amount of time that the client will wait to get a response to a request that it has sent.

If the optional `timeout_ms` attribute is specified, then the request timeout is set to the specified value.

If the optional `timeout_ms` attribute is not specified, then the current value of request timeout is displayed.

Example A-1 timeout

The following example gets the current value of the request timeout.

```
sql-> timeout
Request timeout used: 5,000ms
```

Example A-2 timeout

The following example set the request timeout value to 20000 milliseconds (20 seconds).

```
sql-> timeout 20000
Request timeout used: 20,000ms
```

> **Note:**
>
> A shell command may require multiple requests to a server or servers. The timeout applies to each such individual request. A shell command sends out multiple requests and has to wait for each of them to return before the command is finished. As a result, a shell command may have to wait for longer time than the specified timeout and this total wait could be greater than the wait time of the individual request.

timer

```
timer [on | off]
```

Turns the measurement and display of execution time for commands on or off. If not specified, it shows the current state of `timer`. For example:

```
sql-> timer on
sql-> SELECT * from users where id <= 10 ;
 +----+-----------+-----------+-----+
 | id | firstname | lastname  | age |
 +----+-----------+-----------+-----+
 |  8 | Len       | Aguirre   |  42 |
 | 10 | Montana   | Maldonado |  40 |
 |  6 | Lionel    | Church    |  30 |
 |  3 | Bruno     | Nunez     |  49 |
 |  2 | Idona     | Roman     |  36 |
 |  4 | Cooper    | Morgan    |  39 |
 |  7 | Hanae     | Chapman   |  50 |
 |  9 | Julie     | Taylor    |  38 |
 |  1 | Dean      | Morrison  |  51 |
 |  5 | Troy      | Stuart    |  30 |
 +----+-----------+-----------+-----+

10 rows returned

Time: 0sec 98ms
```

verbose

```
verbose [on | off]
```

Toggles or sets the global verbosity setting. This property can also be set on a per-command basis using the `-verbose` flag.

version

```
version
```

Display client version information.