Oracle

Getting Started with SQL For Oracle NoSQL Database

12c Release 1

Library Version 12.1.4.0

ORACLE®
NOSQL DATABASE

Legal Notice

Published 6/3/2016

Table of 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 the *SQL for Oracle NoSQL Database Specification*.

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.

Note that this is a preview release of SQL for Oracle NoSQL Database. As a preview release its use and feedback is encouraged. For more details on what it means for a feature to be a preview release feature, please see the release notes.

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.

Chapter 1. The SQL for Oracle NoSQL Database Data Model

This chapter gives an overview of the data model. For a more detailed description of the data model see the *SQL for Oracle NoSQL Database Specification*.

In SQL for Oracle NoSQL Database data is modeled as typed items. A typed item (or simply item) is a value and an associated type that contains the value. A type is a definition of a set of values that are said to belong to (or be instances of) that type.

Values can be atomic or complex. An atomic value is a single, indivisible unit of data. A complex value is a value that contains or consists of other values and provides access to its nested values. Similarly, the types supported by SQL for Oracle NoSQL Database can be characterized as atomic types (containing atomic values only) or complex types (containing complex values only).

The data model supports the following kinds of atomic values and associated data types: integer, long, float, double, string, boolean, binaries, and enums. SQL for Oracle NoSQL Database also supports the following complex types:

- Array

  An array is an ordered collection of zero or more items, all of which have the same type.

- Map

  Is an unordered collection of zero or more key-item pairs, where all keys are strings and all the items have the same type.

- Record

  Is an ordered collection of one or more key-item pairs, where all keys are strings and the items associated with different keys may have different types.

Another difference between records and maps is that the keys in records are fixed and known in advance (they are part of the record type definition), whereas maps can contain arbitrary keys (the map keys are not part of the map type).

Chapter 2. SQL for Oracle NoSQL Database Queries

This chapter walks you through query examples run with the interactive SQL shell, which is used to directly execute DDL, DML, user management, security, and informational statements.

If you are interested in using the JAVA API to execute queries see *Getting Started with the Table API*.

Note that this is a preview release of SQL for Oracle NoSQL Database. As a preview release its use and feedback is encouraged. For more details on what it means for a feature to be a preview release feature, please see the release notes.

Expressions

In general, an expression represents a set of operations to be executed in order to produce a result. Expressions are built by combining other subexpressions using operators (arithmetic, logical, value and sequence comparisons), function calls, or other grammatical constructs. As we will see, the simplest kinds of expressions are constants and references to variables or identifiers.

In SQL for Oracle NoSQL Database, the result of any expression is always a sequence of zero or more items. Notice that a single item is considered equivalent to a sequence containing that single item.

In the current version, a query is always a single Select-From-Where (SFW) expression. The SFW expression is essentially a simplified version of the SQL Select-From-Where query block. The two most important simplifications are the lack of support for joins and for subqueries. On the other hand, to manipulate complex data (records, arrays, and maps), provides extensions to traditional SQL through novel kinds of expressions, such as path expressions.

In this chapter we present the kinds of expressions that are currently supported. Expressions will be presented using a series of example queries. For each query, we will show the result it generates on a set of sample data.

For a more detailed description of the language see the *SQL for Oracle NoSQL Database Specification*.

Simple Select-From-Where Queries

In this section we walk you through examples of queries over simple, relational data. If you want to follow along the examples, you can run the `SQLBasicExamples` script found in the Examples folder, which creates the table and imports the data used.

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 populates 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":200000
};

{
  "id":4,
  "firstname":"Peter",
  "lastname":"Smith",
  "age":38,
  "income":80000
};

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

To run the queries, start the shell:

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

To learn more about the shell and its available commands, see Introduction to the SQL for Oracle NoSQL Database shell (page 31).

Selecting columns

You can select columns from a table by specifying the table name in the FROM clause of a SFW expression, and listing the names of the desired columns in the SELECT clause of the same SFW expression. You can also request all the columns of a table by using the short-hand "star" notation in the SELECT clause. For example:

To select all the columns of the table Users:

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

The result as shown by the shell is:

```
+----+-----------+----------+-----+--------+
 | id | firstname | lastname | age | income |
 +----+-----------+----------+-----+--------+
 |  4 | Peter     | Smith    |  38 |  80000 |
 |  3 | John      | Morgan   |  38 | 200000 |
 |  2 | John      | Anderson |  35 | 100000 |
 |  1 | David     | Morrison |  25 | 100000 |
 |  5 | Dana      | Scully   |  47 | 400000 |
 +----+-----------+----------+-----+--------+

5 rows returned
```

To select specific column(s) from the table Users, include them as a comma separated list in the SELECT clause:

```
onql-> 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
```

Renaming columns

To select lastname and rename it as Surname, use the AS keyword in the SELECT clause.

```
onql-> SELECT lastname AS Surname FROM Users;
 +----------+
 | Surname  |
 +----------+
 | Scully   |
 | Smith    |
 | Morgan   |
```

```
 | Anderson |
 | Morrison |
 +----------+

5 rows returned
```

Computing new columns

The SELECT clause can also contain expressions that compute/create new values from existing data. In fact, any kind of expression that returns at most one item can be used in the SELECT list. Here we show two examples demonstrating arithmetic expressions. The usual arithmetic operators: +, -, *, and / are supported.

To select the income column and perform a division operation which calculates `monthlysalary`:

```
onql-> 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   | 200000 |         16666 |
 +----+----------+--------+---------------+

5 rows returned
```

To select the income column and perform an addition operation which calculates `salarywithbonus`:

```
onql-> 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   | 200000 |          205000 |
 |  2 | Anderson | 100000 |          105000 |
 +----+----------+--------+-----------------+

5 rows returned
```

Identifying tables and their columns

Currently, the FROM clause can contain one table only (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 3 ways of writing the same query:

```
onql-> 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:

```
onql-> 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:

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

Filtering results

You can filter the result by specifying a filter condition in the WHERE clause. Typically, a filter condition is an expression that consists of one or more comparison expressions connected through the logical operators AND and/or OR. The usual comparison operators: =, !=, >, >=, <, and <= are supported. For example:

To return users whose firstname is John:

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

2 rows returned
```

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

```
onql-> SELECT id, lastname, income, income/12 AS monthlysalary
FROM Users WHERE income/12 > 6000;
 +----+----------+--------+---------------+
 | id | lastname | income | monthlysalary |
 +----+----------+--------+---------------+
 |  5 | Scully   | 400000 |         33333 |
 |  3 | Morgan   | 200000 |         16666 |
 |  4 | Smith    |  80000 |          6666 |
 |  2 | Anderson | 100000 |          8333 |
```

```
 |  1 | Morrison | 100000 |          8333 |
 +----+----------+--------+---------------+

5 rows returned
```

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

```
onql-> SELECT lastname, age, income FROM Users
WHERE 30 <= age and age <=40 or income > 100000;
 +----------+-----+--------+
 | lastname | age | income |
 +----------+-----+--------+
 | Smith    |  38 |  80000 |
 | Morgan   |  38 | 200000 |
 | 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:

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

1 row returned
```

Ordering Results

You can order the result by a primary key column or a non-primary key column (if you first create an index) using the ORDER BY clause. For example:

To order by using a primary key column (id), specify the sort column in the ORDER BY clause:

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

```
5 rows returned
```

To order by a non-primary key column, you need to first create an index. To create an index and then order by lastname:

```
onql-> create index idx1 on Users(lastname);
Statement completed successfully
onql-> SELECT id, lastname FROM Users ORDER BY lastname;
 +----+----------+
 | id | lastname |
 +----+----------+
 |  2 | Anderson |
 |  3 | Morgan   |
 |  1 | Morrison |
 |  5 | Scully   |
 |  4 | Smith    |
 +----+----------+

5 rows returned
```

You can order by more than one column, if you create an index on those columns. For example, to order users by age and income:

```
onql-> create index idx2 on Users(age, income);
Statement completed successfully
onql-> 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 | 200000 |
 |  5 | Scully   |  47 | 400000 |
 +----+----------+-----+--------+

5 rows returned
```

The idx2 index can also be used to order by age only (but not by income only, nor by income first and age second).

```
onql-> SELECT id, lastname, age from Users ORDER BY age;
 +----+----------+-----+
 | id | lastname | age |
 +----+----------+-----+
 |  1 | Morrison |  25 |
 |  2 | Anderson |  35 |
 |  4 | Smith    |  38 |
 |  3 | Morgan   |  38 |
 |  5 | Scully   |  47 |
 +----+----------+-----+

5 rows returned
```

To learn more about indexes see Working With Indexes (page 22).

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

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

5 rows returned
```

Using External Variables

Use of external variables allows a query to be 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 (see *Getting Started with the Table API*), which must be used before the query is executed. External variables must be declared in the query before they can be referenced in the SFW expression. For example:

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

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

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

Working with complex data

In this section we walk you through query examples that use complex types (arrays, maps, records). If you want to follow along the examples, you can run the `SQLAdvancedExamples` script found in the Examples folder, which creates the table and imports the data used.

The `SQLAdvancedExamples` script creates the following table:

```
create table Persons (
  id integer,
  firstname string,
  lastname string,
  age integer,
  income integer,
  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,
  "address":{"street":"150 Route 2",
             "city":"Antioch",
             "state":"TN",
             "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,
  "address":{"street":"187 Hill Street",
             "city":"Beloit",
             "state":"WI",
             "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,
```

```
  "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,
  "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,
  "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}
};
```

Note

The Persons table models persons that may be connected to other persons in the same table. These connections are stored in the "connections" column, which is an array holding the ids of other persons that a person is connected with. It is assumed that the entries of each "connections" array are sorted (in descending order) by a measure of the strength of the connection. For example, person 3 is most strongly connected with person 1, less strongly connected with person 4, and the least strongly connected with person 2.

The Persons table includes an "expenses" column, which is a map of integers. It stores, for each person, the amount of money spent on various categories of items. Because the categories may be different for each person, and/or because we may want to add or delete categories dynamically (without changing the schema of the table), it makes sense to model this information in a map.

To navigate inside complex values and select their nested values, SQL for Oracle NoSQL Database supports path expressions. Path expressions consist of a number of steps. There are 3 kinds of steps: field, filter, and slice steps. Field steps are used to select field/entry values from records or maps. Filter steps are used to select array or map entries that satisfy some condition. Slice steps are used to select array entries based on their position inside the containing array. A path expression can mix different kinds of steps.

Note

A path expression over a table row must always start with the table's name or the table's alias (if one was included in the FROM clause).

In general, path expressions may return more than one item as their result. Such multi-item results can be used as input in two other kinds of expressions: sequence-comparison operators and array constructors.

The following sections demonstrate examples of path expressions, sequence comparisons, and array constructors.

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:

```
onql-> 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 following example demonstrates sequence comparisons, which are done using the any operators: =any, >any, >=any, <any, <=any, and !=any. These should be used when one or both of the operands may be a sequence with more than one item. The any operators returns true if there is a pair of items, one from the left-hand-side and the other from the right-hand-side, that have the required relationship (equal, greater, etc). For example, to return the last name of persons who have a phone number with area code 423:

```
onql-> 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.

SQL for Oracle NoSQL Database can also sort query results by the value of fields nested inside records, if again, an index on the nested field (or fields) exists. For example, to create an index and then order by state:

```
create index indx1 on Persons (address.state);
Statement completed successfully
onql-> 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 (page 22).

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 the familiar array-indexing syntax:

```
onql-> 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 above example, the slice step `[1]` is applied to the connections array. Array elements are numbered starting with 0, so 1 is used to select the second connection.

A slice step can also be used to select all array elements whose positions are within a range: [low:high], where low and high are expressions that compute the range boundaries. The low and/or the high expressions may be missing if no low and/or high boundary is desired.

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

```
onql-> 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 expression. The path expression returns its result as a sequence of 3 items.

The path expression appears inside the SELECT clause, which does not allow expressions that return more than one item. Therefore, the path expression should be enclosed in an array-constructor expression ([]), which creates a new array (single item) containing the

3 connections. Although the query shell displays the elements of this constructed array vertically, the number of rows returned by this query is 1.

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.

```
onql-> SELECT id, [connections[3:]]
AS weakConnections FROM Persons;
+----+-----------------+
| id | weakConnections |
+----+-----------------+
|  4 | 2               |
+----+-----------------+
|  3 |                 |
+----+-----------------+
|  2 |                 |
+----+-----------------+
|  1 |                 |
+----+-----------------+
|  5 | 3               |
+----+-----------------+

5 rows returned
```

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.

```
onql-> 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 use the square brackets ([]) syntax as well. 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:

```
onql-> 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:

```
onql-> 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:

```
onql-> 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. Because the whole path expression may return more than one phone number, it is enclosed in an array constructor to collect the selected phone numbers into a single 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:

```
onql-> 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
```

In addition to the implicitly-declared $ and $element variables, the condition inside a filter step can also use the $elementPos variable (also implicitly declared). $elementPos 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.

```
onql-> SELECT id, [p.connections[$element >= 4 and $elementPos < 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].

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

1 row returned
```

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:

```
onql-> 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:

```
onql-> 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 on maps work the same way as in arrays. Empty square brackets return all the values in a map. If the square brackets contain a condition expression the condition is evaluated for each entry, and the entry is selected/skipped if the result is true/false. The implicitly-declared variables $key and $element can be used inside a map filter condition. $key references the key of the current entry and $element references the associated value. Notice that, contrary to arrays, the $elementPos variable can not be be used inside map filters (because map entries do not have fixed positions).

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

```
onql-> SELECT id, p.expenses
FROM Persons p
WHERE p.expenses[] >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:

```
onql-> SELECT id FROM Persons p
WHERE p.expenses[$key != "food"] >any 2000;
 +----+
 | id |
 +----+
 |  2 |
 +----+

1 row returned
```

Built-in Functions

You can use function-call expressions to invoke functions, which in the current version can only be built-in (system) functions. Only two functions are included in the current version: keys and size.

keys

The function keys can be used to return the keys (field names) of a given record or map. An empty map has no keys, but a record has always at least one key.

To return the id and the expenses categories for each person:

```
onql-> SELECT id, [keys(p.expenses)]
AS properties FROM Persons p;
 +----+------------+
 | id | properties |
 +----+------------+
 |  4 | books      |
 |    | clothes    |
 |    | food       |
 |    | shoes      |
 +----+------------+
 |  3 | food       |
 |    | gas        |
 |    | travel     |
 +----+------------+
 |  5 | clothes    |
 |    | food       |
 |    | shoes      |
 +----+------------+
 |  2 | books      |
```

```
 |    | food       |
 |    | travel     |
 +----+------------+
 |  1 | food       |
 |    | gas        |
 +----+------------+

5 rows returned
```

size

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:

```
onql-> 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:

```
onql-> 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: has:

```
onql-> SELECT id, size([p.expenses[$element > 2000]])
AS expensiveCategories FROM Persons p;
 +----+---------------------+
 | id | expensiveCategories |
```

```
 +----+--------------------+
 |  4 |                  1 |
 |  2 |                  1 |
 |  5 |                  0 |
 |  3 |                  0 |
 |  1 |                  0 |
 +----+--------------------+

5 rows returned
```

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. Sometimes more than one index is applicable to a query. In the current implementation only one of the applicable indexes can be used, and the processor uses a simple heuristic to choose what seems the best index. Because the heuristic may not always choose the best index, SQL for Oracle NoSQL Database allows users to force the use of a particular index via index hints, which are written inside the query itself. Here is an example:

```
onql-> create index idx_income on Persons (income);
Statement completed successfully
onql-> create index idx_age on Persons (age);
Statement completed successfully
onql-> mode line
Query output mode is LINE
onql-> SELECT * from Persons
WHERE income > 10000000 and age < 40;
> Row 0
 +-------------+--------------------------------+
 | id          | 3                              |
 +-------------+--------------------------------+
 | firstname   | John                           |
 +-------------+--------------------------------+
 | lastname    | Morgan                         |
 +-------------+--------------------------------+
 | age         | 38                             |
 +-------------+--------------------------------+
 | income      | 100000000                      |
 +-------------+--------------------------------+
 | 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              |
 |              | gas          | 10                |
 |              | travel       | 700               |
 +--------------+--------------------------------+

1 row returned
```

In the above query, 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. In its current implementation, 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 this example, it is the idx_age index. To choose the idx_income index instead, the query should be written with an index hint:

```
onql-> 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                      |
 +--------------+--------------------------------+
 | 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              |
 |             | 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 immediatelly after (without any space) the opening "/*".

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

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

 > Row 0
 +-------------+-----------------------------------+
 | id          | 4                                 |
 +-------------+-----------------------------------+
 | firstname   | Peter                             |
 +-------------+-----------------------------------+
 | lastname    | Smith                             |
 +-------------+-----------------------------------+
 | age         | 38                                |
 +-------------+-----------------------------------+
 | income      | 80000                             |
 +-------------+-----------------------------------+
 | 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    | 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). Furthermore, 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 will be retrieved from the table.

The next two examples demostrate the use of multi-key indexes, that is indexes that index 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.

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

 > Row 0
 +-------------+---------------------------------+
 | id          | 2                               |
 +-------------+---------------------------------+
 | firstname   | John                            |
 +-------------+---------------------------------+
 | lastname    | Anderson                        |
 +-------------+---------------------------------+
 | age         | 35                              |
 +-------------+---------------------------------+
 | income      | 100000                          |
 +-------------+---------------------------------+
 | address     | street       | 187 Hill Street |
 |             | city         | Beloit          |
 |             | state        | WI              |
 |             | 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                             |
+-------------+-----------------------------------+
| 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    | books        | 240                |
```

```
 |             | clothes      | 2000               |
 |             | food         | 6000               |
 |             | shoes        | 1200               |
 +-------------+----------------------------------+

 > Row 2
 +-------------+----------------------------------+
 | id          | 5                                |
 +-------------+----------------------------------+
 | firstname   | Dana                             |
 +-------------+----------------------------------+
 | lastname    | Scully                           |
 +-------------+----------------------------------+
 | age         | 47                               |
 +-------------+----------------------------------+
 | income      | 400000                           |
 +-------------+----------------------------------+
 | 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    | 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: the key 339 will be searched for in the index and all the associated table rows will be retrieved.

```
onql-> create index idx_expenses on
Persons (keyof(expenses), elementof(expenses));
Statement completed successfully
onql-> SELECT * FROM Persons p WHERE p.expenses.food > 1000;
 > Row 0
 +-------------+--------------------------------+
 | id          | 2                              |
 +-------------+--------------------------------+
 | firstname   | John                           |
 +-------------+--------------------------------+
 | lastname    | Anderson                       |
 +-------------+--------------------------------+
 | age         | 35                             |
 +-------------+--------------------------------+
 | income      | 100000                         |
 +-------------+--------------------------------+
 | address     | street       | 187 Hill Street |
 |             | city         | Beloit          |
 |             | state        | WI              |
 |             | 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                      |
 +-------------+--------------------------------+
 | 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             |
  |             | gas          | 10               |
  |             | travel       | 700              |
  +-------------+---------------------------------+

  > Row 2
  +-------------+-----------------------------------+
  | id          | 4                                 |
  +-------------+-----------------------------------+
  | firstname   | Peter                             |
  +-------------+-----------------------------------+
  | lastname    | Smith                             |
  +-------------+-----------------------------------+
  | age         | 38                                |
  +-------------+-----------------------------------+
  | income      | 80000                             |
  +-------------+-----------------------------------+
  | 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    | 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: 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.

For a more detailed description of index creation and usage, see the see the *SQL for Oracle NoSQL Database Specification*.

Appendix 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. Then, the available shell commands are described.

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

Running the shell

The shell is run interactively or used to run single commands. The general usage to start the shell is:

```
java -jar KVHOME/lib/onql.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]
```

where:

- `-consistency`

  Configures the read consistency used for this session.

- `-durability`

  Configures the write durability used for this session.

- `-helper-hosts`

  Specifies a comma-separated list of hosts and ports.

- `-store`

  Specifies the name of the store.

- `-timeout`

  Configures the request timeout used for this session.

- `-username`

  Specifies the username to login as.

For example, you can start the shell like this:

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

The above command assumes that a store "kvstore" is running at port 5000. You can now execute queries. In the next part of the book, you will learn all about `SQL for Oracle NoSQL Database` and how to create these 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 (page 34).

If you want to run a script file, you can use the "load" command. For more information see load (page 34).

For a complete list of the utility commands accessed through "`java -jar` <kvhome>`/lib/onql.jar` <command>" see Shell Utility Commands (page 33)

Configuring the shell

You can also pass the shell start-up arguments by modifying the configuration file `.kvclirc` found in the user 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. `[onql]` 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:

```
[onql]
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/onql.jar <command>".

The interactive prompt for the shell is:

```
onql->
```

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:
   ```
   onql-> command [arguments]
   ```
2. All arguments are specified using flags which 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.

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 <name> -file <name> [JSON | CSV]
```

Imports records from the specified file into the named table. The records can be in either JSON or CSV format. If the format is not specified JSON is assumed.

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

For example, a file containing the records of 2 tables "users" and "email":

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

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/onql.jar -helper-hosts node01:5000 \
-store kvstore
onql-> load -file ./test.onql
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 column.

For example, a table shown in COLUMN mode:

```
onql-> mode column;
onql-> 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.

```
onql-> mode column;
onql-> 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:

```
onql-> 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:

```
onql-> mode line;
```

```
onql-> 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
```

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

```
onql-> mode line;
onql-> 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:

```
onql-> mode json;
onql-> 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 "".

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

1 row returned
```

Finally, a table shown in CSV mode:

```
onql-> mode csv;
onql-> 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 "".

```
onql-> mode csv;
onql-> 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

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

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

timeout

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

Configures or displays the request timeout for this session. If not specified, it shows the current value of request timeout.

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:

```
onql-> timer on
onql-> 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.