Oracle

Getting Started with SQL For Oracle NoSQL Database

12c Release 1

Library Version 12.1.4.3

ORACLE®

NOSQL DATABASE

Legal Notice

Published 17-Feb-2017

Table of Contents

Preface vi
Conventions Used in This Book vi
1. The SQL for Oracle NoSQL Database Data Model 1
Example Data 1
Data Types and Values 2
Wildcard Types and JSON Data 4
JSON Data 5
Timestamp 5
Type Hierarchy 6
Subtype-Substitution Rule Exception 8
SQL for Oracle NoSQL Database Sequences 8
2. SQL for Oracle NoSQL Database Queries 10
Select-From-Where (SFW) Expressions 10
SELECT Clause 10
SELECT Clause Hints 11
FROM Clause 12
WHERE Clause 13
ORDER BY Clause 13
Comparison Rules 13
OFFSET Clause 14
LIMIT Clause 14
3. Constructors 15
Array Constructors 15
Map Constructors 15
4. Operators 17
Value Comparison Operators 17
Logical Operators 18
Sequence Comparison Operators 19
Exists Operator 19
Is-Of-Type Operator 19
5. Expressions 21
Cast Expressions 21
Column Reference Expression 22
Constant Expressions 23
Path Expressions 23
Field Step Expressions 24
Map Filter Step Expressions 25
Array Filter Step Expressions 25
Array Slice Step Expressions 26
Searched Case Expressions 27
Variable Reference Expression 28
6. Built-in Functions 29
size 29
7. Simple Select-From-Where Queries 31
SQLBasicExamples Script 31
Running the SQL Shell 32

Selecting columns 32
Renaming columns 33
Computing new columns 33
Identifying tables and their columns 34
Filtering results 35
Ordering Results 36
Limiting and Offsetting Results 38
Using External Variables 39
8. Working with complex data 40
SQLAdvancedExamples Script 40
Working with Timestamps 43
Working With Records 43
Using ORDER BY to Sort Results 45
Working With Arrays 46
Working With Maps 51
9. Working With Indexes 54
10. Working with JSON 63
SQLJSONExamples Script 63
Basic Queries 66
Using Exists with JSON 67
Seeking NULLS in Arrays 68
Examining Data Types JSON Columns 69
Using Map Steps with JSON Data 72
Casting Datatypes 74
Using Searched Case 75
A. Introduction to the SQL for Oracle NoSQL Database Shell 81
Running the shell 81
Configuring the shell 82
Shell Utility Commands 83
connect 83
consistency 83
describe 84
durability 84
exit 84
help 84
history 84
import 84
load 85
mode 85
output 89
page 89
show faults 89
show query 89
show tables 90
show users 90
show users 90
timeout 90
timer 90
verbose 91

version 91

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.

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

Welcome to SQL for Oracle NoSQL Database. This language provides a SQL-like interface to Oracle NoSQL Database. The first portion of this book provides an introduction to the language. If you prefer a more example-oriented introduction to the language, skip to the following chapters:

- Simple Select-From-Where Queries (page 31)
- Working with complex data (page 40)
- Working With Indexes (page 54)
- Working with JSON (page 63)

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

Example Data

The language definition portion of this document frequently provides examples to illustrate the concepts. The following table definition is used by those examples:

```
CREATE TABLE Users (
  id INTEGER,
  firstName STRING,
  lastName STRING,
  otherNames ARRAY(RECORD(first STRING, last STRING)),
  age INTEGER,
  income INTEGER,
  address JSON,
  connections ARRAY(INTEGER),
  expenses MAP(INTEGER),
  moveDate timestamp(4),
  PRIMARY KEY (id)
)
```

The rows of the Users table defined above represent information about users. For each such user, the "connections" field is an array containing ids of other users that this user is connected with. The ids in the array are sorted by some measure of the strength of the connection.

The "expenses" column is a maps expense categories (like "housing", clothes", "books", etc) to the amount spent in the associated category. The set of categories may not be known in advance, or it may differ significantly from user to user, or may need to be frequently updated by adding or removing categories for each user. As a result, using a map type for "expenses", instead of a record type, is the right choice.

Finally, the "address" column has type JSON. A typical value for "address" will be a map representing a json document.

Typical row data for this table will look like this:

```
{
  "id":1,
  "firstname":"David",
  "lastname":"Morrison",
  "otherNames" : [{"first" : "Dave",
                   "last" : "Morrison"}],
  "age":25,
  "income":100000,
  "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},
  "moveDate" : "2016-10-29T18:43:59.8319"
}
```

Data Types and Values

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

  4-byte long integer number.

- Long

  8-byte long integer number.

- Float

  4-byte long real number.

- Double

  8-byte long real number.

- String

  Sequence of unicode characters.

- Boolean

  Values are either `true` or `false`.

- Binaries

  An uninterpreted sequence of zero or more bytes.

- Enums

  Values are symbolic identifiers (tokens). Enums are stored as strings, but are not considered to be strings.

- Timestamp

  Represents a point in time as a date and, optionally, a time. See Timestamp (page 5) for details.

- JSON null value (JNULL)

  Special value that indicates the absence of an actual value within a JSON datatype such as an object, map, or array.

- SQL NULL

  Special value that is used to indicate the absence of an actual value, or the fact that a value is unknown or inapplicable.

SQL for Oracle NoSQL Database also supports the following complex types:

- Array

  An array is an ordered collection of zero or more items. Normally all elements of an array have the same type. Also, normally arrays cannot contain NULL items. However, arrays of type JSON can contain a mix of JSON datatypes, as well as NULL items.

- Map

  An unordered collection of zero or more key-item pairs, where all keys are strings and all the items normally have the same type. Also, normally Maps cannot contain NULL items. However, if the map is of type JSON, then it can contain a mix of datatypes for the items, as well as NULL items.

- Record

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. Also, record items may be NULL.

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).

Wildcard Types and JSON Data

The Oracle NoSQL data model includes the following wildcard types:

- Any

  All possible values.

- AnyAtomic

  All possible atomic values.

- AnyJsonAtomic

  All atomic values that are valid JSON values. This is the union of all numeric values, all string values, the 2 boolean values, and the JNULL value.

- JSON

  All possible JSON values. The domain set is defined recursively as follows:

  1. All AnyJsonAtomic values.
  2. All arrays whose elements are included in AnyJsonAtomic values.
  3. All maps whose field values are those described in (1) and (2) of this list.

- AnyRecord

  All possible record values.

With the exception of JNULL items (which pair the JNULL value with the JSON type), no item can have a wildcard type as its type. Wildcard types should be viewed as abstract types. However, items may have an imprecise type. For example, an item may have `Map(JSON)` as its type, indicating that its value is a map that can store field values of different types, as long as all of these values belong to the JSON type. In fact, `Map(JSON)` is the type that represents all JSON objects (JSON documents), and `Array(JSON)` is the type that represents all JSON arrays.

Note

A type is called precise if it is not one of the wildcard types and, in case of complex types, all of its constituent types are also precise. Items that have precise types are said to be strongly typed.

JSON Data

To load JSON data into a table, input is accepted as strings or streams containing JSON text. Oracle NoSQL Database parses the input text internally and maps its constituent pieces to the values and types of the data model described here.

Specifically, 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. For example, even if the array contains integers only, the array item that will be created will have type `Array(JSON)`. The reason that the array is not created with type `Array(Integer)` is that this would mean that we could never update the array by putting something other than integers.

For the same reason, when a JSON object is encountered in the input text, a map item is created whose type is `Map(JSON)`, unconditionally. 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). Finally, strings in the input text are mapped to string items, boolean values are mapped to boolean items, and JSON nulls to JSON null items.

Timestamp

Represents a point in time as a date and, optionally, a time value.

Timestamp values have a precision in fractional seconds that range from 0 to 9. For example, a precision of 0 means that no fractional seconds are stored, 3 means that the timestamp stores milliseconds, and 9 means a precision of nanoseconds. 0 is the minimum precision, and 9 is the maximum.

There is no timezone information stored in timestamp; they are all assumed to be in the UTC timezone.

The number of bytes used to store a timestamp depends on its precision (the on-disk storage varies between 5 and 9 bytes).

This datatype is specified as a string in the following format:

```
 "<yyyy>-<mm>-<dd>[T<HH>:<mm>:<ss>[.<SS>]]"
```

where:

- <yyyy> is the four-digit year value (for example, "2016").
- <mm> is the two-digit month value (for example, "08").
- <dd> is the two-digit day value (for example, "01").
- <HH> is the two-digit hour value (for example, "18").
- <mm> is the two-digit minute value.

<ss> is the two-digit seconds value.

- <SS> is the fractional seconds value. If the value specified here exceeds the precision declared when the table column was defined, then the value specified is rounded off.

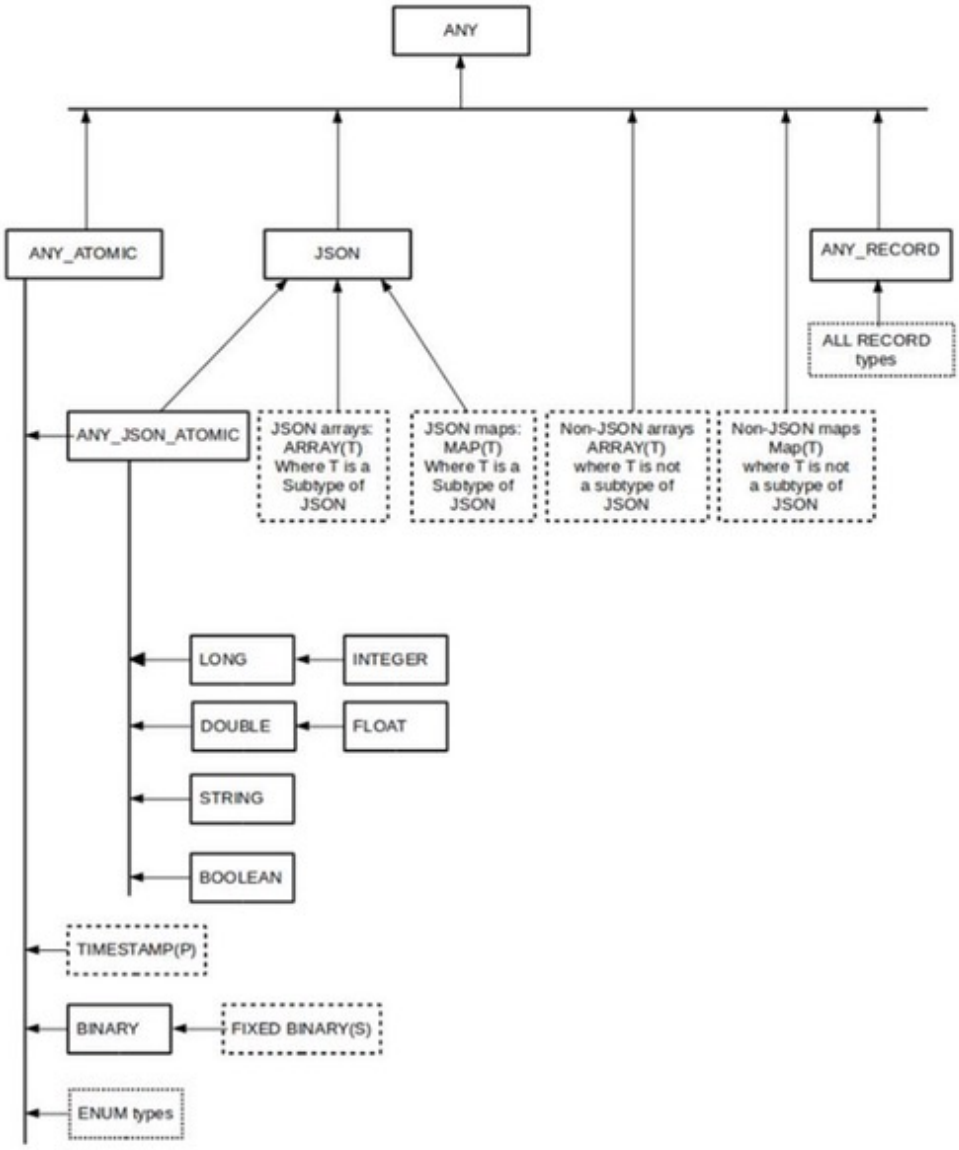
Type Hierarchy

SQL for Oracle NoSQL Database defines a subtype-supertype relationship among the types such that types are arranged in a hierarchy. For example, every type is a subtype of ANY. Any atomic type is a subtype of AnyAtomic. Integer is a subtype of Long. An array is a subtype of JSON if its element type is JSON or another subtype of JSON.

A data item is an instance of a type (T) if the data item's type is (T) or a subtype of (T).

This relationship is important because the usual subtype-substitution rule is supported by SQL for Oracle NoSQL Database. If an operation expects input items of type (T), or produces items of type (T), then it can also operate on or produce items of type (S) if (S) is a subtype of (T). (There is an exception to this rule. See Subtype-Substitution Rule Exception (page 8)).

The following figure illustrates this data hierarchy. Dotted boxes in the figure represent collections of types.



In addition to the subtype relationships described here, the following relationships are also defined:

- Every type is a subtype of itself. A type (T) is a proper subtype of another type (S) if (T) is a subtype of (S) and (T) is not equal to (S).
- An enum type is a subtype of another enum type if both types contain the same tokens and in the same order, in which case the types are actually considered equal.

- Timestamp(p1) is a subtype of Timestamp(p2) if p1 <= p2.
- A record type (S) is a subtype of another record type (T) if:
  1. both types contain the same field names in the same order; and
  2. for each field, its value type in (s) is a subtype of its value type in (T); and
  3. nullable fields in (S), are also nullable in (T).
- (Array(S)) is a subtype of (Array(T)) if (S) is a subtype of (T).
- (Map(S)) is a subtype of (Map(T)) if (S) is a subtype of (T).

Subtype-Substitution Rule Exception

Ordinarily, if an operation expects input items of type (T), or produces items of type (T), then it can also operate on or produce items of type (S) if (S) is a subtype of (T). However, there is an exception to this rule.

Items whose type is a proper subtype of Array(JSON) or Map(JSON) cannot be used as:

- record/map field values if the field type is JSON, Array(JSON) or Map(JSON); or
- elements of arrays whose element type is JSON, Array(JSON) or Map(JSON).

This is in order to disallow strongly type data to be inserted into JSON data.

For example, consider a JSON document which is a map value whose associated type is Map(JSON). The document may contain an array whose values contain only integers. However, the type associated with the array cannot be Array(integer), it must be Array(JSON). If the array had type Array(integer), the user would not be able to add any non-integer values to it.

SQL for Oracle NoSQL Database Sequences

A *sequence* is an important concept in SQL for Oracle NoSQL Database. It is used wherever expressions and operators are discussed.

A sequence is the result of any expression that returns zero or more items. Sequences are not containers; they cannot be nested.

Note that an array is not a sequence; rather it is a single item that contains other items in it.

Sequences have a type. A sequence type specifies the type of items that may appear in a sequence. For example, a sequence could contain atomic elements that are of type integer. In this case, the sequence type would be integer.

Sequence types have a cardinality. The cardinality indicates constraints on how many items can or must appear in the sequence:

- *

indicates a sequence of zero or more items.

- +

  indicates a sequence of one or more items.

- ?

  indicates a sequence of zero or one items.

- The absence of a quantifier indicates a sequence of exactly one item.

When we say that the result of an expression must have a sequence of a certain type, what we mean is the sequence must have that type or any subtype of that type.

Sequence subtypes are defined as follows:

- The empty sequence is a subtype of all sequence types whose quantifier is * or ?.
- A sequence type (s1) is a subtype of another sequence type (s2) if:
  - s1's item type is a subtype of s2's item type; and
  - s1's sequence quantifier is a sub-quantifier for s2's quantifier.

The following table shows the subtype relationship for the various quantifiers:

S2 \ S1	one	?	+	*
one	TRUE	FALSE	FALSE	FALSE
?	TRUE	TRUE	FALSE	FALSE
+	TRUE	FALSE	TRUE	FALSE
*	TRUE	TRUE	TRUE	TRUE

Chapter 2. SQL for Oracle NoSQL Database Queries

This chapter describes the Select-From-Where (SFW) expression, which is the core expression used to form SQL queries. For examples of using SFW expressions, see these extended examples:

- Simple Select-From-Where Queries (page 31)
- Working with complex data (page 40)
- Working With Indexes (page 54)
- Working with JSON (page 63)

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

Note

The examples shown in this chapter rely on the sample data shown in Example Data (page 1).

Select-From-Where (SFW) Expressions

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), SQL for Oracle NoSQL Database provides extensions to traditional SQL through novel kinds of expressions, such as path expressions.

The semantics of the SFW expression are similar to those in standard SQL. Processing starts with the FROM clause, followed by the WHERE clause (if any), followed by the ORDER BY clause (if any), followed by the OFFSET and LIMIT clauses, and finishing with the SELECT clause. Each clause is described below. A query must contain exactly one SFW expression, which is also the top-level expression of the query. Subqueries are not supported yet.

```
SELECT <expression>
FROM <table name>
[WHERE <expression>]
[ORDER BY <expression> [<sort order>]]
[OFFSET <number>]
[LIMIT <number>];
```

Each of the SFW clauses are introduced in the following sections. For details on each clause, see the *SQL for Oracle NoSQL Database Specification*.

SELECT Clause

SELECT clauses come in two forms. In the first form, it contains only a single star (*) symbol. This form simply returns all rows.

```
SELECT * FROM Users;
```

In the second form, the SELECT clause contains a comma-separated list of field expressions, where each expression is optionally associated with a name. In the simplest case, each expression is simply the name of a column in the table from which data is being selected.

```
SELECT id, firstname, lastname FROM Users;
```

The AS keyword can also be used:

```
SELECT id, firstname AS Name, lastname AS Surname FROM Users;
```

SELECT clauses can contain many different kinds of expressions. For more information, see Expressions (page 21).

The SELECT clause always returns a record. Normally, the record has one field for each field expression, and the fields are arranged in the same order as the field expressions. Each field value is the value computed by the corresponding field expression and its name is the name associated with the field expression. If no field name is provided explicitly (using the AS keyword), one is automatically generated for you.

To create valid records, the field names must be unique, and they must return at most one item. If a field expression returns more than one result, the result is returned in an array.

If the result of a field expression is empty, NULL is used as the value of the corresponding field in the record returned by SELECT.

Note

If the SELECT clause contains only one field expression with no associated name, then just the value returned by the clause is returned. If this value is already a record, then this is returned. If this value is not a record, then it is wrapped in a record before being returned.

SELECT Clause Hints

The SELECT clause can contain one or more hints which are used to help choose an index to use for the query. A hint is a comment that begin with a + symbol:

```
/*+ <hint> */
```

Each hint takes the form:

```
<hint type> (<table path> [<index name>]) [comment string]
```

The following hint types are supported:

- FORCE_INDEX

  Specifies a single index, which is used without considering any of other indexes. This is true even if there are no index predicates for the forced index. However, if the query has an ORDER BY clause, and the forced index is not the sorting index, an error is thrown.

This index hint requires you to specify an <index name>.

- PREFER_INDEXES

  The PREFER_INDEXES hint specifies one or more indexes. The query processor may or may not use one of the preferred indexes.

  This index hint requires you to specify at least one <index name>.

- FORCE_PRIMARY_INDEX

  Requires the query to use the table's primary index.

  You do not specify an <index name> when you use this type of hint.

- PREFER_PRIMARY_INDEX

  Specifies that you prefer to use the primary index for the query. This index may or may not be used.

  You do not specify an <index name> when you use this type of hint.

For more information on indexes, see Working With Indexes (page 54).

FROM Clause

The FROM clause is very simple: it can include only a single table. The table is specified by its name, which may be a composite (dot-separated) name in the case of child tables. The table name may be followed by a table alias.

For example, to select a table named `Users`:

```
<select expression> FROM Users <other clauses>;
```

To select a table named `People`, which is a child of a table named `Organizations`:

```
<select expression> FROM Organizations.People <other clauses>;
```

To select a table named `People` and give it the alias `u`:

```
<select expression> FROM Users u <other clauses>;
```

The result of the FROM clause is a sequence containing the rows of the referenced table. The FROM clause creates a nested scope, which exists for the rest of the SFW expression.

The SELECT, WHERE, and ORDER BY clauses operate on the rows produced by the FROM clause, processing one row at a time. The row currently being processed is called the context row. The context row can be referenced in expressions by either the table name, or the table alias.

If the table alias starts with a dollar sign ($), then it serves as a variable declaration whose name is the alias. This variable is bound to the context row and can be referenced within the SFW expression anywhere an expression returning a single record may be used. If this variable has the same name as an external variable, it hides the external variable. Because table alias are essentially variables, the like all other variables their names are case-sensitive.

WHERE Clause

The WHERE clause returns a subset of the rows coming from the FROM clause. Specifically, for each context row, the expression in the WHERE clause is evaluated. The result of this expression must have type BOOLEAN?. If the result is false, or empty, or NULL, the row is skipped; otherwise the row is passed on to the next clause.

For example, to limit the rows selected to just those where the column `firstname` contains `John`:

```
<select statement> <from statement> WHERE firstname = "John";
```

ORDER BY Clause

The ORDER BY clause reorders the sequence of rows it receives as input. The relative order between any two input rows is determined by evaluating, for each row, the expressions listed in the order-by clause and comparing the resulting values. Each order-by expression must have type AnyAtomic?.

Note

It is possible to perform ordering only if there is an index that already sorts the rows in the desired order.

For detailed information on how comparison is performed for order-by expressions, see the *SQL for Oracle NoSQL Database Specification*.

For example, to order a query result by age.

```
<select statement> <from statement> WHERE firstname = "John"
ORDER BY age;
```

It is possible to specify a sorting order: `ASC` (ascending) or `DESC` (descending). Ascending is the default sorting order. To present these results in descending order:

```
<select statement> <from statement> WHERE firstname = "John"
ORDER BY age DESC;
```

You can also specify whether NULLS should come first or last in the sorting order. For example:

```
<select statement> <from statement> WHERE firstname = "John"
ORDER BY age DESC NULLS FIRST;
```

Remember that ordering is only possible if there is an index that sorts the rows in the desired order. Be aware that, in the current implementation, NULLs are always sorted last in the index. The specified handling for NULLs must match the index so, currently, if the sort order is ascending then NULL LAST must be used, and if the sort order is descending then NULL FIRST must be used.

Comparison Rules

This section describes the sorting rules used when query results are sorted.

First, consider the case where only one ORDER BY clause is used in the query.

Two rows are considered equal if both rows contain the same number of elements, and for equivalent positions in each row, the atomic values are identical. So if you have two rows, R1 and R2, then they are equal if R1[0] = R2[0] and R1[1] = R1[1]. In this context, NULLs are considered equal only to other NULLs.

Assuming that the number of elements in R1 and R2 are equal, then R1 is less than R2 if any of the following is true:

- No NULLs appear in either row and sorting is in ascending order. In this case, R1 is less than R2 if there are a positionally-equivalent pair of atomic elements (as evaluated from lowest to highest) where the R1 element is less than the R2 element. That is, if R1[1] < R2[2] then R1 is less than R2.
- No NULLs appear in either row and sorting is in descending order. In this case, R1 is less than R2 if there are a positionally-equivalent pair of atomic elements (as evaluated from lowest to highest) where the R1 element is greater than the R2 element. That is, if R1[1] > R2[2] then R1 is less than R2.
- A NULL appears in R2, but not in R1, and sorting is in ascending order with NULLS LAST.
- A NULL appears in R2, but not in R1, and sorting is in descending order with NULLS FIRST.

If multiple ORDER BY statements are offered, then atomic values are returned for comparison purposes by evaluating the statements from left to right.

Be aware that if an expression returns an empty sequence, then the return value is NULL.

If no sorting order is provided to the query, then by default ascending order with NULLS LAST is used. If only the sort order is specified, then NULLs sort last if the order is ascending. Otherwise, they sort first.

OFFSET Clause

Specifies the number of initial query results that should be skipped; that is, they are not returned. This clause accepts a single non-negative integer as its argument. This argument may be a single integer literal, or a single external variable, or any expression which is built from literals and external variables.

Although it is possible to use this clause without an ORDER BY clause, it does not make sense to do so. Without an ORDER BY clause, results are returned in random order, so the subset of results skipped will be different each time the query is run.

LIMIT Clause

Specifies the maximum number of results to return. This clause accepts a single non-negative integer as its argument. This argument may be a single integer literal, or a single external variable, or any expression which is built from literals and external variables.

Although it is possible to use this clause without an ORDER BY clause, it does not make sense to do so. Without an ORDER BY clause, results are returned in random order, so the subset of results returned will be different each time the query is run.

Chapter 3. Constructors

SQL for Oracle NoSQL Database offers two constructors that you can use: Array and Map constructors.

Note

The examples shown in this chapter rely on the sample data shown in Example Data (page 1).

Array Constructors

```
[ <expression>, <expression>, ... ]
```

An array constructor constructs a new array out of the items returned by the expressions inside the square brackets. These expressions are computed left to right and the produced items are appended to the array. Any NULLs produced by the input expressions are skipped (arrays cannot contain NULLs).

The type of the constructed array is determined during query compilation, based on the types of the input expressions and the usage of the constructor expression. Specifically, if a constructed array can be inserted in another constructed array and this "parent" array has type ARRAY(JSON), then the "child" array will also have type ARRAY(JSON). This is because "typed" data is not allowed inside JSON data.

For example, the use of the explicit array constructor here means that the field will exist in all returned rows, even if the path inside the array constructor returns empty. Without this constructor, a NULL can be returned for the field. With it, an empty result returns an empty array.

```
select firstname, lastname,
[u.expenses.keys($value > 1000)] AS Expenses
from Users u;
```

Map Constructors

```
{ <expression>:<expression>,
<expression:<expression>, ... }
```

A map constructor constructs a new map out of the items returned by the expressions inside the curly brackets. These expressions come in pairs: each pair computes one field.

The first expression in a pair must return at most one string, which serves as the field's name. If the field name expression returns the empty sequence, no field is constructed.

The second expression returns the field value. If this expression returns more than one item, an array is implicitly constructed to store the items, and that array becomes the field value. If the field value expression returns the empty sequence, no field is constructed.

If the computed name or value for a field is NULL the field is skipped (maps cannot contain NULLs).

The type of the constructed map is determined during query compilation, based on the types of the input expressions and the usage of the constructor expression. Specifically, if a constructed map can be inserted in another constructed map and this "parent" map has type MAP(JSON), then the "child" map will also have type MAP(JSON). This is because "typed" data is not allowed inside JSON data.

For example, construct a map consisting of user's last name and their phone numbers:

```
select {
    “last_name” : u.lastName,
    “phones” : u.address.phones
}
from Users u;
```

Chapter 4. Operators

This chapter describes the various operators you can use with your SQL expressions.

Note

The examples shown in this chapter rely on the sample data shown in Example Data (page 1).

Value Comparison Operators

Value comparison operators are primarily used to compare 2 values, one produced by the left operand and another from the right operand. The available value comparison operators are:

```
=
!=
>
>=
<
<=
```

If any operand returns more than one item, an error is raised. If both operands return the empty sequence, the operands are considered equal (so true will be returned if the operator is =, <=, or >=). If only one of the operands returns empty, the result of the comparison is false unless the operator is !=.

Among atomic items, if the types of the items are not comparable, false is returned. The following rules defined what atomic types are comparable and how the comparison is done in each case.

- A numeric item is comparable with any other numeric item. If an integer/long value is compared to a float/double value, the integer/long will first be cast to float/double.
- A string item is comparable to another string item.

  A string item is also comparable to an enum item. In this case, before the comparison the string is cast to an enum item in the type of the other enum item. Such a cast is possible only if the enum type contains a token whose string value is equal to the source string. If the cast is successful, the two enum items are then compared as explained in the next bullet; otherwise, the two items are incomparable and false is returned.
- Two enum items are comparable only if they belong to the same type. If so, the comparison is done on the ordinal numbers of the two enums (not their string values).
- Binary and fixed binary items are comparable with each other for equality only. The 2 values are equal if their byte sequences have the same length and are equal byte-per-byte.
- A boolean item is comparable with another boolean item.
- A timestamp item is comparable to another timestamp item, even if their precisions are different.

- JNULL (JSON null) is comparable with JNULL. If the comparison operator is !=, JNULL is also comparable with every other kind of item, and the result of such a comparison is always true, except when the other item is also JNULL.

The semantics of comparisons among complex items is:

- A record is comparable with another record for equality only, and only if they contain comparable values. To be equal, the 2 records must have equal sizes (number of fields) and for each field in the first record, there must exist a field in the other record such that the two fields are at the same position within their containing records, have equal field names, and equal values.
- A map is comparable with another map for equality only, and only if they contain comparable values. To be equal, the 2 maps must have equal sizes (number of fields) and for each field in the first map, there must exist a field in the other map such that the two fields have equal names and equal values.
- An array is comparable to another array, if the elements of the 2 arrays are comparable pair-wise. Comparison between 2 arrays is done lexicographically. That is, the arrays are compared like strings, with the array elements playing the role of the "characters" to compare.

As with atomic items, if two complex items are not comparable according to the above rules, false is returned. Comparisons between atomic and complex items return false always.

Note

The reason for returning false for incomparable items, instead of raising an error, is to handle schemaless applications where different table rows may contain very different data, or differently shaped data. As a result, even the writer of the query may not know what kind of items an operand may return and an operand may indeed return different kinds of items from different rows. Nevertheless, when the query writer compares "something" with, say, an integer, they expect that the "something" will be an integer and they would like to see results from the table rows that fulfill that expectation, instead of the whole query being rejected because some rows do not fulfill the expectation.

Logical Operators

The binary AND and OR operators and the unary NOT operator have the usual semantics. Their operands are conditional expressions, which must have type BOOLEAN.

An empty result from an operand is treated as the false value. If an operand returns NULL then:

- The AND operator returns false if the other operand returns false; otherwise, it returns NULL.
- The OR operator returns false if the other operand returns false; otherwise, it returns NULL.
- The NOT operator returns NULL.

Sequence Comparison Operators

Comparisons between two sequences is done using the following operators:

=any
!=any
>any
>=any;
<any
<=any

The result of an any operator on two input sequences S1 and S2 is true only if:

1. There is a pair of items, i1 and i2; and
2. i1 belongs to S1, and i2 belongs to S2; and
3. i1 and i2 compare true using the corresponding value-comparison operator.

Otherwise, if any of the input sequences contains NULL, the result is NULL.

Otherwise, the result is false.

Exists Operator

The exists operator checks whether a sequence is empty. True is returned if the sequence is not empty.

Note that this operator returns true even if the sequence contains null data. So for strongly type data (that is, non-JSON data columns), this operator will always return true because on data import the column will be always at minimum populated with null.

For an examples of using the exists operator, see Using Exists with JSON (page 67).

Is-Of-Type Operator

Returns true if the input sequence matches the identified type.

```
<sequence> IS OF TYPE (<type>*|+|?,<type>*|+|?,...])
```

or

```
<sequence> IS NOT OF TYPE (<type>*|+|?,<type>*|+|?,...)
```

The is-of-type operator checks the input sequence type against one or more target sequence types. It returns true if both of the following conditions are true:

- The cardinality of the input sequence matches the quantifier of the target type:
  - If the quantifier is *, the input sequence may have any number of items.
  - If the quantifier is +, the input sequence must have at least one item.

  - If the quantifier is ?, the input sequence must have at most one item.
  - If there is no quantifier, the input sequence must have exactly one item.
- All of the items in the input sequence are instance of the specified type(s). For the purpose of this check, a NULL is not considered to be and instance of any type.

  SQL for Oracle NoSQL Database's subtype-supertype relationship model is relevant to the usage of this operator. See Type Hierarchy (page 6) for more information.

If the cardinality requirement is met and the input sequence contains a NULL, this operator returns NULL. In all other cases, the result is false.

Note

If the number of the target types is greater than one, the expression is equivalent to OR-ing that number of is-of-type expressions, each having one target type.

For an example of using is-of-type, see Examining Data Types JSON Columns (page 69).

Chapter 5. 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. 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.

Note

The examples shown in this chapter rely on the sample data shown in Example Data (page 1).

Cast Expressions

The cast expression creates, if possible, new items of a given target type from the items of its input sequence.

```
CAST (<input_sequence> AS <target_type><quantifier>)
```

Cast expressions are evaluated as follows:

1. A cardinality check is performed based on the <quantifier>. If <quantifier> is:

   - *

     then <input_sequence> may have any number of items.

   - +

     then <input_sequence> must have at least one item.

   - ?

     then <input_sequence> must have at most one item.

   - No quantifier

     then <input_sequence> must exactly one item.

   If this check fails, an error is raised.

2. Each input item is cast to the <target_type>:

   - If the type of the input item is equal to the target item type, the cast is a noop: the input item itself is returned.

   - If the target type is a wildcard type, the cast is a noop if the type of the input item is a subtype of the wildcard type; otherwise an error is raised.

- An error is also raised if the input item can not be cast to the target type.

The following rules specify when the various data types can be cast (see Data Types and Values (page 2) for a description of the types discussed below):

- Every atomic item can be cast to the String type.
- Every numeric item can be cast to every other numeric type.
- String items may be cast to all other atomic types. Whether the cast succeeds depends on whether the actual string value can be parsed into a value that belongs to the domain of the target type.
- Timestamp items can be cast to all the timestamp types. If the target type has a smaller precision that the input item, the resulting timestamp is the one closest to the input timestamp in the target precision. For example, consider the following 2 timestamps with precision 3: 2016-11- 01T10:00:00.236 and 2016-11-01T10:00:00.267. The result of casting these timestamps to precision 1 is: 2016-11-01T10:00:00.2 and 2016-11-01T10:00:00.3, respectively.
- Array items can be cast to array types only. A new array is created whose type is the target type, and each element in the input array is cast to the element type of the target array. If this cast is successful, the new element is added to the new array.
- Map items can be cast to map types only. A new map is created whose type is the target type, and each field value in the input map is cast to the value type of the target map. If this cast is successful, the new field value and the associated field name are added to the new map.
- Record items may be cast to record types only. The target type must have the same fields and in the same order as the type of the input item. A new record is created whose type is the target type, and each field value in the input record is cast to the value type of the corresponding field in the target type. If this cast is successful, the new field value and the associated field name are added to the new record.

For example the following selects the last name of users who moved to their current address in 2015 or later. Because there is no literal for Timestamp values, to create such a value a string has to cast to a Timestamp type:

```
select u.lastName
from Users u
where cast (u.moveDate as Timestamp(0)) >=
   cast (“2015-01-01T00:00:00” as Timestamp(0));
```

For more examples of using the cast expression, see Casting Datatypes (page 74).

Column Reference Expression

A column-reference expression returns the item stored in the specified column within the context row (the row that a WHERE, ORDER BY, or SELECT clause is currently working on).

A column-reference expression consists of one identifier, or two identifiers separated by a dot. If there are two ids, the first is considered to be a table name or /alias, and the second a column in that table. A single id refers to a column in the table referenced inside the FROM clause.

Notice that child tables in Oracle NoSQL Database have composite names using dot as a separator among multiple ids. As a result, a child-table name cannot be used in a column-reference expression; instead, a table alias must be used to access a child table column using the two-id format. For example, if "Address" is a child table of Persons, then:

```
SELECT id, p.Address.state FROM Persons p;
```

Constant Expressions

There are five kinds of constants available:

- Strings.

  Sequences of unicode characters enclosed in double or single quotes. String literals are translated into String items.

- Integer numbers

  Sequences of one or more digits. Integer literals are translated into Integer items, if their value fits in 4 bytes, otherwise into Long items.

- Real numbers

  Representation of real numbers using "dot notation" and/or exponent. Real literals are translated into Double items.

- The boolean values `true` and `false`.

- The JSON null value.

Path Expressions

To navigate inside complex values and select their nested values, SQL for Oracle NoSQL Database supports path expressions. A path expression has an input expression followed by one or more steps.

```
<primary_expressions>.<step>*
```

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).

There are three kinds of path expression 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.

All steps iterate over their input sequence, producing zero or more items for each input item. If the input sequence is empty, the result of the step is also empty. Otherwise, the overall result of the step is the concatenation of the results produced for each input item. The input item that a step is currently operating on is called the *context item*, and it is available within the step expression using the dollar sign ($) variable. This context-item variable exists in the scope created by the step expression.

For all steps, if the context item is NULL, it is just added into the output sequence with no further processing.

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.

Field Step Expressions

A field step selects the value of a field from a record or map. The field to select is specified by its field name, which is either given explicitly as an identifier, or is computed by a name expression. The name expression must be of type string.

```
<primary_expression>.<id> | <string> | <var_ref> |
<parenthesized_expr> | <func_call>*
```

As a simple example, the field step expression `u.address.city`:

```
SELECT id, u.address.city FROM Users u;
```

Retrieves the field "city" from the "address" column in the Users ("u") table.

A field step processes each context item as follows:

1. If the context item is an atomic item, it is skipped (the result is empty).
2. The name expression is evaluated. If the name expression returns the empty sequence or NULL, the context item is skipped. Otherwise, the evaluated name expression is passed to the next step.
3. If the context item:
   - Is a record

     and if that record contains a field identical to the evaluated name expression, then that field is returned. Otherwise, an error is raised.
   - Is a map

     and if that map contains a field identical to the evaluated name expression, then that field is returned. Otherwise, an empty result is returned.

If the context item ($) is an array, then the field step is applied to each element of the array with the context item being set to the current array element. If the context item is an atomic item, it is skipped (the result is empty).

Map Filter Step Expressions

A map filter step is used with records and maps to select either the field name (keys) or the field values of the fields that satisfy a given condition. This condition is specified as a predicate expression inside parentheses. If the predicate expression is missing, it is assumed to be `true` — all the field names or values are returned.

```
<primary_expression>.keys | values (<predicate>)
```

where `keys` references the record's or map's field name, and `values` references the record's or map's field values.

In addition to the context-item variable ($), the predicate expression may reference the following two variables: $key is bound to the name of the context field — that is, the current field in $, and $value is bound to the value of the context field. The predicate expression must be boolean.

A simple example is `u.expenses.keys($value > 1000)`, which selects all the expenses greater than $1000. Combined with this query:

```
SELECT id, u.expenses.keys($value > 1000) FROM Users u;
```

all the user IDs and expense fields are returned where more than 1000 was spent.

A map filter step processes each context item as follows:

1. If the context item is an atomic item, it is skipped (the result is empty).
2. If the context item is a record or map, the step iterates over its fields. For each field, the predicate expression is evaluated. A NULL or an empty result from the predicate expression is treated as a false value. If the predicate result is true, the context field is selected and either its name or its value is returned; otherwise the context field is skipped.

Note

If the context item ($) is an array, then the map filter step is applied to each element of the array with the context item being set to the current array element.

Array Filter Step Expressions

An array filter step is used with arrays to select elements of arrays by evaluating a predicate expression for each element. Elements are selected or rejected depending on the results of the predicate expression. If the predicate expression is missing, it is assumed to be `true` — all the array elements are returned.

```
[<primary_expression>[<predicate_expression>]]
```

Notice in the syntax that the entire expression is enclosed in square brackets ([]). This is the array constructor. Use of the array constructor is frequently required in order to obtain the desired result, and so we show it here. The use of the explicit array constructor guarantees that the records in the result set will always have an array as their second field. For example:

```
SELECT lastName,
[ u.address.phones[$element.area = 650].number ] AS phoneNumbers
FROM Users u;
```

Assume that `u.address.phones` references one or more phone numbers. Without the array constructor, the result records would contain an array for users with more than one phone (because the information would be held in an array in the store anyway), but just a single integer for users with just one phone. For users with just one phone, the phones field might not be an array (containing a single phone object), but just a single phone object. If such a single phone object has area code 650, its number will be selected, as expected.

In addition to the context-item variable ($), the predicate expression may reference the following two variables: $element is bound to the current element in $, and $pos is bound to the position of the context element within $. Positions are counted starting with 0.

An array filter step processes each context item as follows:

1. If the context item is not an array, an array is created and the context item is added to that array. Then the array filter is applied to this single-item array.

2. If the context item is an array, the step iterates over the array elements and computes the predicate expression on each element.

   The predicate expression must return a boolean item, or a numeric item, or the empty sequence, or NULL. A NULL or an empty result from the predicate expression is treated as a false value. If the predicate result is true/false, the context element is selected/skipped, respectively. If the predicate result is a number, the context element is selected only if $pos equals that number. This means that if the predicate result is a negative number, or greater or equal to the array size, the context element is skipped.

Array Slice Step Expressions

An array slice step is used with arrays to select elements of arrays based on element position. The elements to select are identified by specifying boundary positions which identify "low" position and "high" positions. Each boundary expression must return at most one item of type LONG or INTEGER, or NULL. The low and/or the high expression may be missing. The context-item variable ($) is available during the computation of the boundary expressions.

```
<primary_expression>[<low>:<high>]
```

For example, assume an array of connects ordered from the strongest connect (position 0) to the weakest, select the strongest connection for the user with id 10:

```
select connections[0] as strongestConnection from Users
where id = 10;
```

Select user 10's five strongest connections, and return the array (notice the use of the array constructor):

```
select [ connections[0:4] ] as strongConnections from Users
where id = 10;
```

Select user 10's five weakest connections:

```
select [ connections[size($) - 5 : ] ] as weakConnections from Users
where id = 10;
```

An array slice step processes each context item as follows:

1. If the context item is not an array, an array is created and the context item is added to that array. Then the array filter is applied to this single-item array.

2. If the context item is an array, the boundary expressions (if any) are evaluated.

   If any boundary expression returns NULL or an empty result, the context item is skipped.

   Otherwise, if the low expression is absent, or if it evaluates to less than 0, the lower boundary is set to 0. If the high expression is absent, or if it evaluates to higher than the array_size -1, it is set to array_size - 1.

3. After the low and high positions are determined, the step selects all the elements positions, inclusively, between those two boundaries. If the low position is greater than the high position, then no elements are selected.

Searched Case Expressions

```
CASE
    WHEN <expr> THEN <expr>
    (WHEN <expr> THEN <expr>)*
    (ELSE <expr>)?
    END;
```

The searched case expression consists of a number of when-then pairs, followed by an optional `else` clause at the end. Each when expression is a condition that must return boolean. The `then` expressions as well as the `else` expression may return any sequence of items.

The case expression is evaluated by:

1. Evaluating the `when` expressions from top to bottom until the first one is discovered that returns true.

2. The `then` expression for the previously identified `when` is evaluated. This result is returned as the result for the entire case expression.

3. If no `when` expression returns true, but there is an `else` expression, then that expression is evaluated and its result is the result of the entire case expression.

4. Otherwise, the result of the entire case expression is the empty sequence.

For example, construct a map using the map constructor ({}) in which the `phones`: element is either the contents of the `phones` column, or a string to indicate nothing was found in that column:

```
select {
    “last_name” : u.lastName,
```

```
    “phones” : case
               when exists u.address.phones then u.address.phones
               else “Phone info absent at the expected place”
               end,
    “high_expenses” : [ u.expenses.keys($value > 5000) ]
  }
from Users u;
```

For more examples of using searched case expressions, see Using Searched Case (page 75).

Variable Reference Expression

A variable reference expression is: `$<variablename>`.

A variable-reference expression returns the item that the specified variable is currently bound to. Syntactically, a variable-reference expression is just the name of the variable.

Chapter 6. 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 one function is included in the current version: size.

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:

```
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
```

Chapter 7. 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, get the `Examples` download from http://www.oracle.com/technetwork/database/database-technologies/nosqldb/downloads/index.html and run the `SQLBasicExamples` script found in the Examples folder. This creates the table and imports the data used.

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 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,
}
```

You run the SQLBasicExamples script using the load (page 85) command:

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

Running the SQL Shell

You can run SQL queries using the SQL shell. This is described in Introduction to the SQL for Oracle NoSQL Database Shell (page 81). But, briefly, to run the queries shown in this document, start the shell:

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

To learn more about the shell and its available commands, see Introduction to the SQL for Oracle NoSQL Database Shell (page 81).

Note

This book shows examples which are displayed in COLUMN mode. Be aware that the default output type is JSON. Use the `mode` command to switch between COLUMN and JSON (or JSON pretty) output.

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.

Be aware that the FROM clause can name only one table. If you want to retrieve data from a child table, use dot notation (that is, "parent.child").

To select all the columns of the table Users, use the short-hand "star" notation:

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

The result as shown by the shell is:

```
 +----+-----------+----------+-----+--------+---------------------+
 | id | firstname | lastname | age | income |      lastLogin      |
 +----+-----------+----------+-----+--------+---------------------+
 |  3 | John      | Morgan   |  38 | 200000 | 2016-11-29T08:21:35 |
 |  4 | Peter     | Smith    |  38 |  80000 | 2016-10-19T09:18:05 |
 |  2 | John      | Anderson |  35 | 100000 | 2016-11-28T13:01:11 |
 |  5 | Dana      | Scully   |  47 | 400000 | 2016-11-08T09:16:47 |
 |  1 | David     | Morrison |  25 | 100000 | 2016-10-29T18:43:59 |
 +----+-----------+----------+-----+--------+---------------------+

5 rows returned
```

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

```
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
```

Renaming columns

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

```
sql-> 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`:

```
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   | 200000 |         16666 |
 +----+----------+--------+---------------+

5 rows returned
```

To select the income column and perform an addition operation which calculates `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   | 200000 |          205000 |
 |  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 3 ways of writing 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 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:

```
sql-> 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:

```
sql-> 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:

```
sql-> 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:

```
sql-> 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 using the ORDER BY clause.

Note

You can use ordering only if you are selecting by the table's primary key, or if there is an index that sorts the table's rows in the desired order.

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

```
sql-> 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:

```
sql-> create index idx1 on Users(lastname);
Statement completed successfully
sql-> 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:

```
sql-> create index idx2 on Users(age, income);
Statement completed successfully
sql-> 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).

```
sql-> 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 54).

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

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

5 rows returned
```

Limiting and Offsetting Results

Use the `LIMIT` clause to limit the number of results returned in a SQL statement. Suppose you had 10000 rows in the Users table, but you only wanted to return the first 4 rows. Do this:

```
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 | 200000 |
 |  4 | Peter     | Smith    |  38 |  80000 |
 +----+-----------+----------+-----+--------+

4 rows returned
```

Suppose you only wanted to show results 3 and 4 out of the 10000 rows. To do this, first use the `LIMIT` clause to limit the results to two. Then, use the `OFFSET` clause to skip 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 | 200000 |
 |  4 | Peter     | Smith    |  38 |  80000 |
 +----+-----------+----------+-----+--------+

2 rows returned
```

Note

It is recommended to use LIMIT and OFFSET with an ORDER BY clause. Otherwise, the results will be returned in a random order, producing unexpected results.

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 |
+-----------+----------+-----+
```

Chapter 8. Working with complex data

In this chapter, we walk you through query examples that use complex types (arrays, maps, records). If you want to follow along the examples, get the `Examples` download from http://www.oracle.com/technetwork/database/database-technologies/nosqldb/downloads/index.html and run the `SQLAdvancedExamples` script found in the Examples folder. This 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 (page 85) command:

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

Note

The Persons table models people that might be connected to other people in the same table. These connections are stored in the "connections" column, which is an array holding the ids of other people 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.

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
```

Of course, queries against Timestamps will frequently involve a range of time:

```
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
```

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 (see Path Expressions (page 23)) `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 (see Sequence Comparison Operators (page 19)). 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.

Using ORDER BY to Sort Results

If you want to sort output using a field that is not the table's primary key, then specify an index. For example, if we want to query based on a Timestamp and sort the results in descending order by the timestamp:

```
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 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:

```
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 (page 54).

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:

```
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 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:

```
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 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.

```
sql-> 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.

```
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 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:

```
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. 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:

```
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
```

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 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 a map field value, use `.values()`. To select a map field key, use `.keys()`. Empty expressions for either of these steps returns all values or all keys. If the steps 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 $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 all the id and expense categories where the user 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
```

Chapter 9. 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:

This chapter builds on the examples that you began in Working with complex data (page 40).

```
sql-> create index idx_income on Persons (income);
Statement completed successfully
sql-> create index idx_age on Persons (age);
Statement completed successfully
sql-> mode line
Query output mode is LINE
sql-> 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:

```
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                    |
 +-------------+------------------------------+
 | 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 immediately 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.

```
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                             |
 +-------------+-----------------------------------+
 | 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). 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.

The next two examples demonstrate 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.

```
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                         |
 +-------------+--------------------------------+
 | 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 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                         |
+-------------+--------------------------------+
| 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 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.

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

Chapter 10. Working with JSON

This chapter provides examples on working with JSON data. If you want to follow along the examples, get the `Examples` download from http://www.oracle.com/technetwork/database/database-technologies/nosqldb/downloads/index.html and run the `SQLJSONExamples` script found in the Examples 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 native 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. It uses the SQLJSONExamples script, which can be found in your examples directory, to illustrate its examples.

Note

JSON data cannot be indexed.

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
  }
}
```

Basic Queries

Because JSON is parsed and stored in native data formats internally with Oracle NoSQL Database, querying JSON data is not any different from querying data in other column types. See Simple Select-From-Where Queries (page 31) and Working with complex data (page 40) 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, which is mapped internally into a Map(JSON) type. You can therefore query information in this column in the same way as you would if you were operating on 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       |
 +----+---------------------+------------+

6 rows returned
```

Notice that the last row in the result set contains all NULLs. This is because that row was populated using a JSON object that looks nothing like the objects used to populate the rest of our table. This is both a strength and a weakness of using JSON. You can modify your schema easily, but if you are not careful you can end up with tables containing dissimilar data in both big 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             |
 +----+---------------------+
 |  1 | NULL                |
 +----+---------------------+

6 rows returned
```

Here, id 1 is NULL because that user had no expenses greater than $1000, while id 6 is NULL because it has no `j.person.expenses` field.

Using 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, there is a row in our table that does not contain an address field. To see all the rows that do contain an address field:

For example, some of the JSON persons have a zip code entered for their address, and other do not. 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 examining data for inconsistencies, it might be more useful to see all the rows where information is missing by using `where not exists`:

```
sql-> SELECT * FROM JSONPersons j WHERE NOT EXISTS j.person.lastname;
 +----+-------------------+
 | id |      person       |
 +----+-------------------+
 |  6 | 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 caste one data type to another using the `cast` expression. See Cast Expressions (page 21) for rules on how casting works.

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 (page 43) 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
```

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/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]
```

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

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 find and introduction to `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 84).

If you want to run a script file, you can use the "load" command. For more information see load (page 85).

For a complete list of the utility commands accessed through "`java -jar` <kvhome>/lib/sql.jar <command>" see Shell Utility Commands (page 83)

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> (,<field_name>)*)?
 | index <index_name> on <table_name>
```

Provides a JSON description of a table or index.

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/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 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 tables

```
show [as json] tables | table <table_name>
```

Shows either all tables currently existing in the store, or the named table.

show users

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

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

show users

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

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

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:

```
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.