Oracle® NoSQL Database

SQL Reference Guide

Release 20.2

F14605-05

August 2020

Oracle NoSQL Database SQL Reference Guide, Release 20.2

F14605-05

Contents

1

Introduction to SQL for Oracle NoSQL Database

Structured Query Language (SQL) is the set of statements with which all programs and users access data in the Oracle NoSQL Database. This book provides information on SQL as used by Oracle NoSQL Database. Application programs and Oracle tools often allow users access to the Oracle NoSQL Database without using SQL directly.

This chapter contains the following topics:

Note:

No prior knowledge of SQL is required for reading this document.

SQL Program

The data model of Oracle NoSQL Database supports (a) flat relational data, (b) hierarchical typed (schema-full) data, and (c) schema-less JSON data. SQL for Oracle NoSQL Database is designed to handle all such data in a seamless fashion, without any *impedance mismatch* among the different sub models.

In the current version, an SQL program consists of a single statement, which can be a non-updating query (read-only DML statement), an updating query (updating DML statement), a data definition command (DDL statement), a user management and security statement, or an informational statement. This is illustrated in the following syntax, which lists all the statements supported by the current SQL version.

```
program::=
(
    query |
    insert_statement |
    delete_statement |
    update_statement |
    create_namespace_statement |
    show_namespaces_statement |
    drop_namespace_statement |
    create_region_statement |
    show_regions_statement |
    drop_region_statement |
    create_table_statement |
    show_tables_statement |
    describe_table_statement |
    alter_table_statement |
    drop_table_statement |
    create_index_statement |
    show_indexes_statement |
    describe_index_statement |
    drop_index_statement |
    create_text_index_statement |
    create_user_statement |
    create_role_statement |
    drop_role_statement |
    drop_user_statement |
    alter_user_statement |
    grant_statement |
    revoke_statement |
) EOF
```

This document is concerned with the first 19 statements in the above list, that is, with read-only queries, insert/delete/update statements, namespace statements, and DDL statements, excluding text indexes. The document describes the syntax and semantics for each statement, and supplies examples. The programmatic APIs available to compile and execute SQL statements and process their results are described in the *Java Direct Driver Developer's Guide*.

EBNF Syntax

This specification uses EBNF meta-syntax to specify the grammar of SQL for Oracle NoSQL Database. The following EBNF notations apply:

- **Upper-case words** are used to represent keywords, punctuation characters, operator symbols, and other syntactical entities that are recognized by EBNF as terminals (aka tokens) in the query text. For example, `SELECT` stands for the "select" keyword in the query text. Notice that keywords are case-insensitive; "select" and "sELEct" are both the same keyword, represented by the SELECT terminal.
- **Lower-case words** are used for non-terminals. For example, `array_step : [expression]` means that array_step is an expression enclosed in square brackets.
- **" "** Anything enclosed in quotes is also considered a terminal. For example, the following production rule defines the value-comparison operators as one of the =, >= symbols: For example, `val_comp : "=" | ">="` .
- ***** means 0 or more of whatever precedes it. For example, `field_name*` means 0 or more field names.
- **+** means 1 or more of whatever precedes it. For example, `field_name+` means 1 or more field names.

- **[]** means optional, i.e., zero or 1 of whatever contained in it. For example, `[field_name]` means zero or one field names.
- **|** means this or that. For example, `INT | STRING` means an integer, or a string.
- **()** Parentheses are used to group EBNF sub-expressions together. For example, `(INT | STRING) [comment]` means an integer, or a string, followed by a comment, or just an integer, or a string, followed by nothing.

Comments

The language supports comments in both DML and DDL statements. Such comments have the same semantics as comments in a regular programming language, that is, they are not stored anywhere, and have no effect to the execution of the statements. The following comment constructs are recognized:

/* comment */
Potentially multi line comment.

Note:

However, if a '+' character appears immediately after the opening "/*", and the comment is next to a SELECT keyword, the comment is actually not a comment but a hint for the query processor. See Choosing the Best Applicable Index.

// comment
Single line comment.

# comment
Single line comment.

As we will see, DDL statements may also contain comment clauses, which are stored persistently as properties of the created data entities. Comment clauses start with the COMMENT keyword, followed by a string literal, which is the content of the comment.

Syntax

```
comment ::= COMMENT string
```

Identifiers

An identifier is a sequence of characters conforming to the following rules:

- It starts with a latin alphabet character (characters 'a' to 'z' and 'A' to 'Z').
- The characters after the first one may be any combination of latin alphabet characters, decimal digits ('0' to '9'), or the underscore character ('_').
- It is not one of the reserved words. The only reserved words are the literals TRUE, FALSE, and NULL.

ID is the terminal that represents the identifiers. However, in the grammar rules presented in this document we will use the non-terminal symbol id to denote identifiers.

Syntax

```
ID ::= ALPHABET (ALPHABET | DIGIT | '_')*
ALPHABET ::= 'a'..'z'|'A'..'Z'
DIGIT ::= '0'..'9'

id ::=
ID |
ADD | ALTER | ANCESTORS | AND |
ANY | ANYATOMIC | ANYJSONATOMIC | ANYRECORD |
ARRAY | AS | ASC |
BINARY | BOOLEAN | BY |
CASCADE | CASE | CAST | COMMENT | COUNT | CREATE |
DAYS | DECLARE | DEFAULT | DESC | DESCENDANTS | DISTINCT | DOUBLE | DROP
|
ELSE | END | ENUM | EXISTS | EXTRACT | FIRST | FLOAT | FROM |
GEOMETRY | GROUP | HOURS | IF | IN | INDEX | INTEGER | IS |
JSON | KEY | KEYS |
LAST | LIMIT | LONG | MAP |
NAMESPACE | NESTED | NO | NOT | NULLS |
OF | OFFSET | ON | OR | ORDER |
POINT | PRIMARY | RECORD |
SELECT | SHARD | STRING |
TABLE | TABLES | THEN | TTL | TYPE |
USING | VALUES |
WHEN | WHERE | WITH
```

Literals

A literal (a.k.a constant value) is a fixed value appearing in the query text. There are four kinds of literals: numbers, strings, boolean values, and the JSON NULL value. The following production rules are used to recognize literals in the query text. The Constant Expressions section describes how the tokens listed below are translated into instances of the data model.

Syntax

```
INT_CONSTANT ::= DIGIT+
FLOAT_CONSTANT ::=
   (DIGIT* '.' DIGIT+ [(E|e) [+|-]  DIGIT+]) |
   (DIGIT+ (E|e) [+|-] DIGIT+)
NUMBER_CONSTANT ::= (FLOAT_CONSTANT | INT_CONSTANT) (N|n)
STRING_CONSTANT ::= '\'' [(ESC|.)*] '\''
DSTRING_CONSTANT ::= '"' [(ESC|.)*] '"'
ESC ::= '\\' ([\'\\/bfnrt]|UNICODE)
DSTR_ESC ::= '\\' ([\"\\/bfnrt]|UNICODE)
UNICODE ::= 'u'HEX HEX HEX HEX
TRUE ::= (T|t)(R|r)(U|u)(E|e)
```

```
FALSE ::= (F|f)(A|a)(L|l)(S|s)(E|e)
NULL ::= (N|n)(U|u)(L|l)(L|l)
```

Note:

The literals TRUE, FALSE, and NULL are reserved words.

Operator Precedence

The relative precedence among the various operators and expressions in SQL for Oracle NoSQL Database is defined implicitly by the order in which the grammar rules for these operators and expressions are listed in the grammar specification. Specifically, the earlier a grammar rule appears, the lower its precedence. For example, consider the following 3 rules that define the syntax for the OR, AND, and NOT operators. Because or_expr appears before and_expr and not expr, OR has lower precedence than AND and NOT. And AND has lower precedence than NOT, because and_expr appears before not_expr. As a result, an expression like a = 10 and not b > 5 or c < 20 and c > 1 is equivalent to (a = 10 and (not b > 5)) or (c < 20 and c > 1). See Logical Operators: AND, OR, and NOT for more details.

```
or_expression ::= and_expression | (or_expression OR and_expression)
and_expression ::= not_expression | (and_expression AND not_expression)
not_expression ::= [NOT] is_null_expression
```

Reserved Words

Reserved words are words that look like identifiers, but cannot be used as identifiers (i.e., in places where identifiers are expected). SQL for Oracle NoSQL Database has a short list of reserved words. Currently, this list consists of the following (case-insensitive) words: **TRUE**, **FALSE**, and **NULL**.

2

Oracle NoSQL Database Data Model

In Oracle NoSQL Database, data is modeled as typed items. A **typed item** (or simply **item**) is a pair consisting of a data type and a value. A **type** is a set of possible values that may be stored in a database field (e.g. a table column) or be returned by some computation (e.g. during the evaluation of a query). In any item, the item value must be an instance of the item type. The values in this set are called the instances of the type. An item is said to be an instance of a type T if its value is an instance of T.

The Oracle NoSQL Database data model consists of various data types that allow for the storage and manipulation of hierarchical data. The Oracle NoSQL Database data types can be broadly classified into **atomic types**, **complex types** and **wildcard types**. Instances of atomic data types are single, indivisible units of data. Instances of complex data types contain multiple items and provide access to their nested items. Wildcard types are similar to abstract supertypes in object-oriented programming languages. They combine instances of other data types and they are used to support data that do not have a fixed structure. This chapter describes each of these data types in detail.

Note:

Data types describe the kind of data that can be stored in an Oracle NoSQL Database, as well as the kind of data generated during the evaluation of a query.

This chapter contains the following topics:

- Atomic Data Types
- Complex Data Types
- Wildcard Data Types
- Data Type Hierarchy
- Data Type Definitions

Atomic Data Types

An instance of an atomic data type is a single, indivisible unit of data. The following table lists the atomic types currently available. For each type, a description of its instances is given.

Table 2-1 Atomic Data Types

Data Type	Description	Example
INTEGER	An integer between -2^31 to 2^31-1.	2147483647

Table 2-1 (Cont.) Atomic Data Types

Data Type	Description	Example
LONG	An integer between -2^63 to 2^63-1.	9223372036854775807
FLOAT	A single precision IEEE 754 floating point number.	100.12345
DOUBLE	A double precision IEEE 754 floating point number.	100.12345678901234
NUMBER	An arbitrary-precision signed decimal number (equivalent to the Java BigDecimal type).	100.123456789
STRING	A sequence of zero or more unicode characters.	"Oracle"
BOOLEAN	Has only two possible values. TRUE and FALSE.	TRUE
BINARY	An uninterpreted sequence of zero or more bytes.	Type: BINARY Type Instance: "0x34 0xF5"
FIXED BINARY (S)	An uninterpreted sequence of S bytes.	Type: BINARY(3) Type Instance: "0x34 0xF5 0xAB"
ENUM (T1, T2, …, Tn)	One of the symbolic tokens (T1, T2, …, Tn) explicitly listed in the ENUM type. The order in which the tokens are listed is important. For example, ENUM(a, b) and ENUM(b, a) are two distinct types.	Type: ENUM(Chennai, Bangalore, Boston) Type Instance: Boston
TIMESTAMP (P)	A value representing a point in time as a date (year, month, day), time (hour, minute, second), and number of fractions of a second. The scale at which fractional seconds are counted is called precision P of a timestamp. The minimum precision is 0 and maximum is 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. There is no timezone information stored in timestamp; they are all assumed to be in the UTC timezone.	Type: timestamp(3) Type Instance : '2020-01-20T12:15:054'

In addition to the kind of atomic values described above, the Oracle NoSQL Database data model includes the following 2 atomic values:

Table 2-2 Atomic Values

Data Type	Description
JSON NULL	This is considered to be an instance of the JSON data type. For more information about JSON, see Wildcard Data Types.
SQL NULL	This is a special value that is used to indicate the fact that a value is unknown or inapplicable. NULL is assumed to be an instance of every data type.

Note:

Although an instance of a numeric type may be semantically equivalent to an instance of another numeric type, the 2 instances are distinct. For example, there is a single number 3 in the universe, but there are 5 different instances of 3 in the data model, one for each of the numeric types.

Complex Data Types

An instance of a complex data type contains multiple values and provides access to its nested values. Currently, Oracle NoSQL Database supports the following kinds of complex values:

Table 2-3 Complex Data Types

Data Type	Description	Example
ARRAY (T)	In general, an array is an ordered collection of zero or more items. The items of an array are called elements. Arrays cannot contain any NULL values. An instance of ARRAY (T) is an array whose elements are all instances of type T. T is called element type of the array.	Type: ARRAY (INTEGER) Type Instance: [600004,560076,01803]
MAP (T)	In general, a map is an unordered collection of zero or more key-item pairs, where all keys are strings. The keys in a map must be unique. The key-item pairs are called fields. The keys are called fields names, and the associated items are called field values. Maps cannot contain any NULL field value. An instance of MAP (T) is a map whose field values are all instance of type T. T is called the value type of the map.	Type: MAP(INTEGER) Type Instance: { "Chennai":600004, "Bangalore":560076, "Boston":01803 }
RECORD (k1 T1 n1, k2 T2 n2, ……., kn Tn nn)	In general, a record is an ordered collection of one or more key-item pairs, where all keys are strings. The keys in a record must be unique. The key-item pairs are called fields. The keys are called fields names, and the associated items are called field values. Records may contain NULL as field value. An instance of RECORD (k1 T1 n1, k2 T2 n2, ……., kn Tn nn) is a record of exactly n fields, where for each field i (a) the field name is ki, (b) the field value is an instance of type Ti, and (c) the field conforms to the nullability property ni, which specifies whether the field value may be NULL or not. Contrary to maps and arrays, it is not possible to add or remove fields from a record. This is because the number of fields and their field names are part of the record type definition associated with a record value.	Type: RECORD(country STRING, zipcode INTEGER, state STRING, street STRING) Type Instance: { "country":"US", "zipcode":600004, "state":"Arizona", "street":"4th Block" }

Example 2-1 Complex Data Type

The following examples illustrate the difference between the way data get stored in various complex data types.

To store the zip codes of multiple cities when the number of zip codes is not known in advance, you can use arrays.

```
Declaration:
ARRAY(INTEGER)


Example:
[600004,560076,01803]
```

To store the names of multiple cities along with their zip codes and the number of zip codes are not known, you can use maps.

```
Declaration:
MAP(INTEGER)


Example:
{
"Chennai":600004,
"Bangalore":560076,
"Boston":01803
}
```

Records are used for an ordered collection. If you want to store a zip code as part of a bigger data set, you can use records. In this example, only a zip code is stored in a record.

```
Declaration:
RECORD(zipcode INTEGER)


Example:
{
"zipcode":600004
}
```

You can combine multiple complex data types so that more complex data can be stored. For the same zipcode data, the following example combines two complex data types and stores the information.

```
Declaration:
ARRAY(RECORD(area STRING, zipcode INTEGER))


Example:
[
    {"area":"Chennai","zipcode":600004},
    {"area":"Bangalore","zipcode":560076},
    {"area":"Boston","zipcode":01803}
]
```

Example 2-2 Complex Data Type

This example illustrate the differences in the way a map and a record should be declared for storing complex data.

Let us consider the following data.

```
{
"name":"oracle",
"city":"Redwood City",
"zipcode":94065,
"offices":["Chennai","Bangalore","Boston"]
}
```

For the above data, you declare a map and a record as shown below.

```
Record
(
name STRING,
city STRING,
zipcode INTEGER,
offices Array(STRING)
)


Map(ANY)
```

Wildcard Data Types

The Oracle NoSQL Database data model includes the following wildcard data types:

Table 2-4 Wildcard Data Types

Data Type	Description	Examples
ANY	Any instance of any NoSQL type is an instance of the ANY type as well.	{ "city" : "Santa Cruz", "zip" : 95008, "phones" : [{ "area" : 408, "number" : 4538955, "kind" : "work" }, { "area" : 831, "number" : 7533341, "kind" : "home" }] } "Santa Cruz" 95008 TRUE '0x34 0xF5' '2020-01-20T12:15:05 4' [12, "foo", { "city":"Santa Cruz"}, [2, 3]]
ANYATOMIC	Any instance of any other atomic type is an instance of the ANYATOMIC type as well. The json null value is also an instance of ANYATOMIC.	"Santa Cruz" 95008 TRUE '0x34 0xF5' '2020-01-20T12:15:05 4'
ANYJSONATOMIC	Any instance of a numeric type, the string type, and the boolean type is an instance of the ANYJSONATOMIC type as well. The json null value is also an instance of ANYJSONATOMIC.	"Santa Cruz" 95008 true
JSON	The JSON type represents all valid json values. Specifically, an instance of JSON can be 1. an instance of ANYJSONATOMIC, 2. or an array whose elements are all instances of JSON, 3. or a map whose field values are all instances of JSON.	{ "city" : "Santa Cruz", "zip" : 95008, "phones" : [{ "area" : 408, "number" : 4538955, "kind" : "work" }, { "area" : 831, "number" : 7533341, "kind" : "home" }] } "Santa Cruz" 95008 true [12, "foo", { "city":"Santa Cruz"}, [2, 3]]
ANYRECORD	Any instance of any other RECORD type is an instance of the ANYRECORD type as well.	{ "city" : "Santa Cruz", "zip" : 95008 }

A data 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**.

Wildcard types are abstract, which means that no item can have a wildcard type as its type. 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.

To load json data into a table, Oracle NoSQL Database offers programmatic APIs that accept input json as strings or streams containing json text. Oracle NoSQL Database will parse the input text internally and map 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). When a JSON object is encountered in the input text, a map item is created whose type is MAP(JSON). When numbers are encountered, they are converted to integer, long, double, or number items, depending on the actual value of the number. 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. In general, the end result of this parsing is a tree of maps, arrays, and atomic values. For persistent storage, the tree is serialized into a binary format.

JSON data is schemaless, in the sense that a field of type JSON can have very different kind of values in different table rows. For example, if "info" is a top-level table column of type JSON, in one row the value of "info" may be an integer, in another row an array containing a mix of doubles and strings, and in a third row a map containing a mix of other maps, arrays, and atomic values. Furthermore, the data stored in a JSON column or field, can be updated in any way that produces a still valid JSON instance. As a result, each JSON tree (either in main memory or as a serialized byte array on disk) is selfdescribing with regard to its contents.

Data Type Hierarchy

The Oracle NoSQL Database data model also defines a **subtype-supertype relationship** among the types presented above. The relationship can be expressed as an is_subtype(T, S) function that returns true if type T is a subtype of type S and false otherwise. is_subtype(T, S) returns true in the following cases:

- T and S are the same type. So, every type is a subtype of itself. We say that 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.
- S is the ANY type. So, every type is a subtype of ANY.
- S is the ANYATOMIC type and T is an atomic type.
- S is ANYJSONATOMIC and T is one of the numeric types or the STRING type, or the BOOLEAN type.
- S is NUMBER and T is one of the other numeric types.
- S is LONG and T is INTEGER.
- S is DOUBLE and T is FLOAT.
- S is TIMESTAMP(p2), T is TIMESTAMP(p1) and p1 <= p2.
- S is BINARY and T is FIXED_BINARY.
- S is ARRAY(T2), T is ARRAY(T1) and T1 is a subtype of T2.
- S is MAP(T2), T is MAP(T1) and T1 is a subtype of T2.

- S and T are both record types and (a) both types contain the same field names and in the same order, (b) for each field, its type in T is a subtype of its type in S, and (c) if the field is nullable in T, it is also nullable in S.
- S is JSON and T is (a) an array whose element type is a subtype of JSON, or (b) a map whose value type is a subtype of JSON, or (c) ANYJSONATOMIC or any of its subtypes.

Note:

The is_subtype relationship is transitive, that is, if type A is a subtype of type B and B is a subtype of C, then A is a subtype of C.

The is_subtype 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 then it can also operate on items of type S, where S is a subtype of T. However, there are two exceptions to this rule:

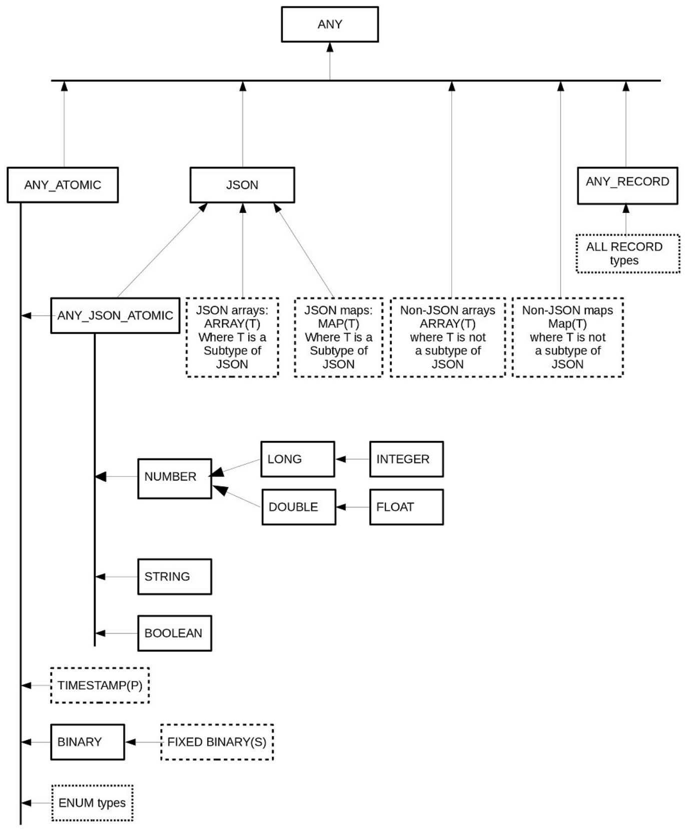
1. DOUBLE and FLOAT are subtypes of NUMBER. However, DOUBLE and FLOAT include three special values in their domain:
   a. NaN (not a number)
   b. Positive infinity
   c. Negative infinity

   These three values are not in the domain of NUMBER. You can provide DOUBLE/ FLOAT types to NUMBER type as long as these are not one of the three special values; otherwise, an error will be raised.
2. Items whose type is a proper subtype of ARRAY (JSON) or MAP (JSON) cannot be used as:
   a. RECORD/MAP field values if the field type is JSON, ARRAY (JSON) or MAP (JSON)
   b. Elements of ARRAY whose element type is JSON, ARRAY (JSON) or MAP (JSON)

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

   For example, consider a JSON document M, i.e., a MAP value whose associated type is a MAP (JSON). M may contain an ARRAY value A that contains only INTEGERs. However, the type associated with A cannot be ARRAY (INTEGER), it must be ARRAY (JSON). If A had type ARRAY (INTEGER), the user would not be able to add any non-INTEGER values to A, i.e., the user would not be able to update the JSON document in a way that would still keep it a JSON document.

Figure 2-1 SQL Type Hierarchy


ANY
ANY_ATOMIC
JSON
ANY_RECORD
ALL RECORD types
ANY_JSON_ATOMIC
JSON arrays: ARRAY(T) Where T is a Subtype of JSON
JSON maps: MAP(T) Where T is a Subtype of JSON
Non-JSON arrays ARRAY(T) where T is not a subtype of JSON
Non-JSON maps Map(T) where T is not a subtype of JSON
NUMBER
LONG
INTEGER
DOUBLE
FLOAT
STRING
BOOLEAN
TIMESTAMP(P)
BINARY
FIXED BINARY(S)
ENUM types

Data Type Definitions

The Oracle NoSQL Database data model types inside SQL statements are referred to using type_definition syntax. This syntax is used both in data definition language (DDL) statements and data manipulation language (DML) statements.

Syntax

```
type_definition ::=
   INTEGER |
   LONG |
   FLOAT |
   DOUBLE |
   NUMBER |
   STRING |
   BOOLEAN |
   ANY |
   JSON |
   ANYRECORD |
   ANYATOMIC |
   ANYJSONATOMIC |
   array_definition |
   map_definition |
   binary_definition |
   timestamp_definition |
   enum_definition |
   record_definition

array_definition ::= ARRAY "(" type_definition ")"
map_definition ::= MAP "(" type_definition ")"
binary_definition ::= BINARY ["(" INT_CONSTANT ")"]
timestamp_definition ::= TIMESTAMP ["(" INT_CONSTANT ")"]

enum_definition ::= ENUM "(" id_list ")"
id_list ::= id ["," id]

record_definition ::= RECORD "(" field_definition (","
field_definition)* ")"
field_definition ::= id type_definition [default_definition] [comment]
default_definition ::=
   (default_value [NOT NULL]) | (NOT NULL [default_value])
default_value ::= DEFAULT (number | string | TRUE | FALSE | id)
```

Semantics

type_definition
When the type_def grammar rule is used in any DDL statement, the only wildcard type that is allowed is the JSON type. So, for example, it is possible to create a table with a column whose type is JSON, but not a column whose type is ANY.

timestamp_definition
The precision is optional while specifying a TIMESTAMP type. If omitted, the default precision is 9 (nanoseconds). This implies that the type TIMESTAMP (with no precision specified) is a supertype of all other TIMESTAMP types (with a specified precision). However, in the context of a CREATE TABLE statement, a precision must be explicitly specified. This restriction is to prevent users from inadvertently creating TIMESTAMP values with precision 9 (which takes more space) when in reality they don't need that high precision.

record_definition
Field default values and descriptions do not affect the value of a RECORD type, i.e., two RECORD types created according to the above syntax and differing only in their default values and/or field descriptions have the same value (they are essentially the same type).

field_definition
The field_definition rule defines a field of a RECORD type. It specifies the field name, its type, and optionally, a default value and a comment. The comment, if present, is stored persistently as the field's description.

default_definition
By default, all RECORD fields are nullable. The default_definition rule can be used to declare a field not-nullable or to specify a default value for the field. When a record is created, if no value is assigned to a field, the default value is assigned by Oracle NoSQL Database, if a default value has been declared for that field. If not, the field must be nullable, in which case the null value is assigned. Currently, default values are supported only for numeric types, STRING, BOOLEAN, and ENUM.

3 Namespace Management

A namespace defines a group of tables, within which all of the table names must be uniquely identified. This chapter describes namespaces and how to create and manage the namespaces in Oracle NoSQL Database.

Namespaces permit you to do table privilege management as a group operation. You can grant authorization permissions to a namespace to determine who can access both the namespace and the tables within it.

Namespaces permit tables with the same name to exist in your database store. To access such tables, you can use a fully qualified table name. A fully qualified table name is a table name preceded by its namespaces, followed with a colon (:), such as `ns1:table1`.

All tables are part of some namespace. There is a default Oracle NoSQL Database namespace, called `sysdefault`. All tables are assigned to the default `sysdefault` namespace, until or unless you create other namespaces, and create new tables within them. You cannot change an existing table's namespace. Tables in `sysdefault` namespace do not require a fully qualified name and can work with just the table name. For example, to access a table in `sysdefault` namespace, you can just specify `table1` instead of `sysdefault:table1`.

Note:

In a store that was created new or was upgraded from a version prior to 18.3, all the tables will be part of `sysdefault` namespace.

This chapter contains the following topics:

CREATE NAMESPACE Statement

You can add a new namespace by using the CREATE NAMESPACE statement.

Syntax

```
create_namespace_statement ::=
   CREATE NAMESPACE [IF NOT EXISTS] namespace_name

namespace_name ::= name_path
```

```
name_path ::= field_name ("." field_name)*
field_name ::= id | DSTRING
```

Semantics

IF NOT EXISTS: This is an optional clause. If you specify this clause, and if a namespace with the same name exists, then this is a noop and no error is generated. If you don't specify this clause, and if a namespace with the same name exists, an error is generated indicating that the namespace already exists.

Note:

Namespace names starting with *sys* are reserved. You cannot use the prefix *sys* for any namespaces.

Example 3-1 Create Namespace Statement

The following statement defines a namespace named *ns1*.

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

SHOW NAMESPACES Statement

Syntax

```
show_namespaces_statement ::= SHOW [AS JSON] NAMESPACES
```

Semantics

The show namespaces statement provides the list of namespaces in the system.

AS JSON can be specified if you want the output to be in JSON format.

Example 3-2 Show Namespaces

The following statement lists the namespaces present in the system.

```
SHOW NAMESPACES;

namespaces
  ns1
  sysdefault
```

Example 3-3 Show Namespaces

The following statement lists the namespaces present in the system in JSON format.

```
SHOW AS JSON NAMESPACES;

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

DROP NAMESPACE Statement

You can remove a namespace by using the DROP NAMESPACE statement.

Syntax

```
drop_namespace_statement ::=
   DROP NAMESPACE [IF EXISTS] namespace_name [CASCADE]
```

Semantics

IF EXISTS: This is an optional clause. If you specify this clause, and if a namespace with the same name does not exist, no error is generated. If you don't specify this clause, and if a namespace with the same name does not exist, an error is generated indicating that the namespace does not exist.

CASCADE: This is an optional clause that enables you to specify whether to drop the tables and their indexes in this namespace. If you specify this clause, and if the namespace contains any tables, then the namespace together with all the tables in this namespace will be deleted. If you don't specify this clause, and if the namespace contains any tables, then an error is generated indicating that the namespace is not empty.

Note:

You cannot drop the default namespace, *sysdefault*.

Example 3-4 Drop Namespace Statement

The following statement removes the namespace named *ns1*.

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

Namespace Resolution

To resolve a table from a table_name that appears in an SQL statement, the following rules apply:

- if the table_name contains a namespace name, no resolution is needed, because a qualified table name uniquely identifies a table.
- if you don't specify a namespace name explicitly, the namespace used is the one contained in the ExecuteOptions instance that is given as input to the executeSync(), execute(), or prepare() methods of TableAPI. See *Java Direct Driver Developer's Guide*.
- if ExecuteOptions doesn't specify a namespace, the default sysdefault namespace is used.

Using different namespaces in ExecuteOptions allows executing the same queries on separate but similar tables.

Namespace Scoped Privileges

You can add one or more namespaces to your store, create tables within them, and grant permission for users to access namespaces and tables. You can grant the following permissions to users. See Grant Roles or Privileges in the *Security Guide*.

System-scoped Privileges:

- CREATE_ANY_NAMESPACE
- DROP_ANY_NAMESPACE

Namespace-scoped privileges:

- CREATE_TABLE_IN_NAMESPACE
- DROP_TABLE_IN_NAMESPACE
- EVOLVE_TABLE_IN_NAMESPACE
- CREATE_INDEX_IN_NAMESPACE
- DROP_INDEX_IN_NAMESPACE

Note:

The label MODIFY_IN_NAMESPACE can be used as a helper to specify all five namespace-scoped privileges.

Example 3-5 Namespace Scoped Privileges

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

4 Region Management

Oracle NoSQL Database supports Multi-Region Architecture in which you can create tables in multiple KVStores, and still maintain consistent data across these clusters. Each KVStore cluster in a Multi-Region NoSQL Database setup is called a Region.This chapter describes creating and managing regions in Oracle NoSQL Database.

This chapter contains the following topics:

- CREATE REGION Statement
- SHOW REGIONS Statement
- DROP REGION Statement

CREATE REGION Statement

In a Multi-Region Oracle NoSQL Database setup, you must define all the remote regions for each local region. For example, if there are three regions in a Multi-Region setup, you must define the other two regions from each participating region. You use the create region statement to define remote regions in the Multi-Region Oracle NoSQL Database.

Syntax

```
create_region_statement ::= CREATE REGION region_name

region_name ::= id | DSTRING
```

Semantics

region_name
The name of the region that is different from the local region where the command is executed.

Example 4-1 Create Region

The following create region statement creates a remote region named *my_region1*.

```
CREATE REGION my_region1;
```

SHOW REGIONS Statement

Syntax

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

Semantics

The show regions statement provides the list of regions present in the Multi-Region Oracle NoSQL Database.

AS JSON can be specified if you want the output to be in JSON format.

Example 4-2 Show Regions

The following statement lists all the existing regions.

```
SHOW REGIONS;

regions
  my_region1 (remote, active)
  my_region2 (remote, active)
```

Example 4-3 Show Regions

The following statement lists all the existing regions in JSON format.

```
SHOW AS JSON REGIONS;

{"regions" : [
    {"name" : "my_region1", "type" : "remote", "state" : "active"},
    {"name" : "my_region2", "type" : "remote", "state" : "active"}
]}
```

DROP REGION Statement

In a Multi-Region Oracle NoSQL Database environment, the drop region statement removes the specified remote region from the local region.

Syntax

```
drop_region_statement :: DROP REGION region_name
```

Semantics

region_name
The name of the region that you want to drop. This region must be different from the local region where the command is executed.

Example 4-4 Drop Region

The following drop region statement removes a remote region named *my_region1*.

```
DROP REGION my_region1;
```

5

Table Management

In Oracle NoSQL Database, data is stored and organized in tables. This chapter describes tables and creating and managing tables in Oracle NoSQL Database.

A table is an unordered collection of record items, all of which have the same record type. We call this record type the table schema. The table schema is defined by the CREATE TABLE statement. The records of a table are called rows and the record fields are called columns. Therefore, an Oracle NoSQL Database table is a generalization of the (normalized) relational tables found in more traditional RDBMSs.

Although table rows are records, records are not rows. This is because, rows have some additional properties that are not part of the table schema (i.e., they are not stored as top-level columns or nested fields). To extract the values of such properties, the functions listed in the Functions on Rows section must be used.

This chapter contains the following topics:

- CREATE TABLE Statement
- SHOW TABLES Statement
- DESCRIBE TABLE Statement
- Table Hierarchies
- Using the IDENTITY Column
- Sequence Generator
- DROP TABLE Statement
- ALTER TABLE Statement
- Altering an IDENTITY Column

CREATE TABLE Statement

The table is the basic structure to hold user data. You use the create table statement to create a new table in the Oracle NoSQL Database.

Syntax

```
create_table_statement ::=
   CREATE TABLE [IF NOT EXISTS] table_name [comment]
   "(" table_definition ")" [ttl_definition]

table_name ::= [namespace_name ":"] name_path
name_path ::= field_name ("." field_name)*
field_name ::= id | DSTRING

table_definition ::=
   (column_definition | key_definition)
   ("," (column_definition | key_definition))*
```

```
column_definition ::=
   id type_definition
   [default_definition | identity_definition]
   [comment]
key_definition ::=
   PRIMARY KEY
   "(" [shard_key_definition [","]] [id_list_with_size] ")"
   [ttl_definition]
id_list_with_size ::= id_with_size ("," id_with_size)*
id_with_size ::= id [storage_size]
storage_size ::= "(" INT_CONSTANT ")"
shard_key_definition ::= SHARD "(" id_list_with_size ")"
ttl_definition ::= USING TTL INT_CONSTANT (HOURS | DAYS)
```

Semantics

table_name
The table name is specified as an optional namespace_name and a local_name. The local name is a name_path because, in the case of child tables, it will consist of a list of dot-separated ids. Child tables are described in the Table Hierarchies section.
A table_name that includes a namespace_name is called a ***qualified table name***. When an SQL statement (DDL or DML) references a table by its local name only, the local name is resolved internally to a qualified name with a specific namespace name. See the Namespace Management chapter.

IF NOT EXISTS
This is an optional clause. If this clause is specified and if a table with the same qualified name exists (or is being created) and if that existing table has the same structure as in the statement, no error is generated. In all other cases and if a table with the same qualified name exists, the create table statement generates an error indicating that the table exists.

ttl_definition
The Time-To-Live (TTL) value is used in computing the expiration time of a row. Expired rows are not included in query results and are eventually removed from the table automatically by Oracle NoSQL Database. If you specify a TTL value while creating the table, it applies as the default TTL for every row inserted into this table. However, you can override the table level TTL by specifying a TTL value via the table insertion API.
The expiration time of a row is computed by adding the TTL value to the current timestamp. To be more specific, for a TTL value of N hours/days, the expiration time is the current time (in UTC) plus N hours/days, rounded up to the next full hour/day. For example, if the current timestamp is 2020-06-23T10:01:36.096 and the TTL is 4 days, the expiration time will be 2020-06-28T00:00:00.000. You can use zero as a special value to indicate that a rows should never expire. If the CREATE TABLE statement has no TTL specification, the default table TTL is zero.
In case of MR Tables with TTL value defined, the rows replicated to other regions carry the expiration time when the row was written. This can be either the default table level TTL value or a row level override that is set by your application. Therefore, this row will expire in all the regions at the same time, irrespective of when they were replicated. However, if a row is updated in one of the regions and it expires in the local region even before it is replicated to one of the remote region(s), then this row will expire as soon as it is replicated and committed in that remote region.

table_definition
The table_definition part of the statement must include at least one field definition, and exactly one primary key definition (Although the syntax allows for multiple key_definitions, the query processor enforces the one key_definition rule. The syntax is this way to allow for the key definition to appear anywhere among the field definitions).

column_definition
The syntax for a column definition is similar to the field_definition grammar rule that defines the fields of a record type. See Data Type Definitions section. It specifies the name of the column, its data type, whether the column is nullable or not, an optional default value or whether the column is an IDENTITY column or not, and an optional comment. As mentioned in Table Management section, tables are containers of records, and the table_definitions acts as an implicit definition of a record type (the table schema), whose fields are defined by the listed column_definitions. However, when the type_definition grammar rule is used in any DDL statement, the only wildcard type that is allowed is the JSON type. So, for example, it is possible to create a table with a column whose type is JSON, but not a column whose type is ANY.

identity_definition
The identity_definition specifies the name of the identity column. There can only be one identity column per table. See Using the IDENTITY Column section.

key_definition
The syntax for the primary key specification (key_definition) specifies the primary key columns of the table as an ordered list of field names. The column names must be among the ones appearing in the field_definitions, and their associated type must be one of the following: a numeric type, string, enum, or timestamp. The usual definition of a primary key applies: two rows of the same table cannot have the same values on all of their primary key columns.

shard_key_definition
A key_definition specifies the table's shard key columns as well, as the first N primary-key columns, where 0 < N <= M and M is the number of primary-key columns. Specification of a shard key is optional. By default, for a root table (a table without a parent) the shard key is the whole primary key. Semantically, the shard key is used to distribute table rows across the multiple servers and processes that comprise an Oracle NoSQL Database store. Briefly, two rows having the same shard key, i.e., the same values on their shardkey columns, will always be located in the same server and managed by the same process. Further details about the distribution of data in Oracle NoSQL Database can be found in the Primary and Shard Key Design section.

storage_size
An additional property of INTEGER-typed primary-key fields is their storage size. This is specified as an integer number between 1 and 5 (the syntax allows any integer, but the query processor enforces the restriction). The storage size specifies the maximum number of bytes that may be used to store in serialized form a value of the associated primary key column. If a value cannot be serialized into the specified number of bytes (or less), an error will be thrown. An internal encoding is used to store INTEGER (and LONG) primary-key values, so that such values are sortable as strings (this is because primary key values are always stored as keys of the "primary" Btree index). The following table shows the range of positive values that can be stored for each byte-size (the ranges are the same for negative values). Users can save storage space by specifying a storage size less than 5, if they know that the key values will

be less or equal to the upper bound of the range associated with the chosen storage size.

comment
Comments are included at table-level and they become part of the table's metadata as uninterpreted text. Comments are displayed in the output of the describe statement.

Example 5-1 Create Table

The following create table statement defines a *users* table that hold information about the users.

```
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),
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. We assume that the ids in the array are sorted by some measure of the strength of the connection. The "expenses" column is a map mapping 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 that looks like this:

```
{
    "street" : "Pacific Ave",
    "number" : 101,
    "city"   : "Santa Cruz",
    "state"  : "CA",
    "zip"    : 95008,
    "phones" : [
            { "area" : 408, "number" : 4538955, "kind" : "work" },
            { "area" : 831, "number" : 7533341, "kind" : "home" }
    ]
}
```

However, any other valid JSON value may be stored there. For example, some addresses may have additional fields, or missing fields, or fields spelled differently. Or, the "phones" field may not be an array of JSON objects but a single such object. Or the whole address maybe just one string, or number, or JNULL.

SHOW TABLES Statement

Syntax

```
show_tables_statement ::=
    SHOW [AS JSON] (TABLES | TABLE table_name)
```

Semantics

The show tables statement provides the list of tables present in the system. If you want to know the details of a specific table, then you can use show table statement. If the named table does not exist then this statement fails.

Example 5-2 Show Tables

The following statement lists all the tables in the system.

```
SHOW TABLES;

tables
  SYS$IndexStatsLease
  SYS$PartitionStatsLease
  SYS$SGAttributesTable
  SYS$StreamRequest
  SYS$StreamResponse
  SYS$TableStatsIndex
  SYS$TableStatsPartition
  Users2
  users
```

Example 5-3 Show Tables

The following statement lists all the tables in the system in JSON format.

```
SHOW AS JSON TABLES;

{"tables" : [
    "SYS$IndexStatsLease",
    "SYS$PartitionStatsLease",
    "SYS$SGAttributesTable",
    "SYS$StreamRequest",
    "SYS$StreamResponse",
    "SYS$TableStatsIndex",
    "SYS$TableStatsPartition",
    "Users2",
    "users"
]}
```

Example 5-4 Show Tables

The following statement lists a specific table in the system.

```
SHOW TABLE users;

tableHierarchy
  users
```

DESCRIBE TABLE Statement

Syntax

```
describe_table_statement ::=
    (DESCRIBE | DESC) [AS JSON] TABLE table_name
    [ "(" field_name [","" field_name] ")"]
```

Semantics

The description for tables contains the following information:

- Name of the table.
- Time-To-Live value of the table.
- Owner of the table.
- Whether the table is a system table.
- Name of parent tables.
- Name of children tables.
- List of indexes present on the table.
- Desciption of the table.

The description for fields contains the following information:

- Id of the field.
- Name of the field.
- Datatype of fields, for example, INTEGER, STRING, Map(INTEGER), etc.
- Whether the field is nullable. If the field is nullable then 'Y' is displayed, otherwise 'N' is displayed.
- Default value of the field.
- Whether the field is a shard key. If the field is a shard key then 'Y' is displayed, otherwise 'N' is displayed.
- Whether the field is a primary key. If the field is a primary key then 'Y' is displayed, otherwise 'N' is displayed.
- Whether the field is an identity field. If the field is an identity field then 'Y' is displayed, otherwise 'N' is displayed.

AS JSON can be specified if you want the output to be in JSON format.

Example 5-5 Describe Table

AS JSON can be specified if you want the output to be in JSON format.

```
DESCRIBE TABLE users;

 === Information ===
 +-------+-----+-------+----------+----------+--------+----------
+---------+-------------+
 | name  | ttl | owner | sysTable | r2compat | parent | children |
indexes | description |
 +-------+-----+-------+----------+----------+--------+----------
+---------+-------------+
 | users |     |       | N        | N        |        |
|         |             |
 +-------+-----+-------+----------+----------+--------+----------
+---------+-------------+

 === Fields ===
 +----+-------------+---------------------+----------+-----------
+----------+------------+----------+
 | id |    name     |        type         | nullable |  default  |
shardKey | primaryKey | identity |
 +----+-------------+---------------------+----------+-----------
+----------+------------+----------+
 |  1 | id          | Integer             | N        | NullValue |
Y        | Y          |          |
 +----+-------------+---------------------+----------+-----------
+----------+------------+----------+
 |  2 | firstName   | String              | Y        | NullValue
|          |            |          |
 +----+-------------+---------------------+----------+-----------
+----------+------------+----------+
 |  3 | lastName    | String              | Y        | NullValue
|          |            |          |
 +----+-------------+---------------------+----------+-----------
+----------+------------+----------+
 |  4 | otherNames  | Array(              | Y        | NullValue
|          |            |          |
 |    |             |   RECORD(           |          |
|          |            |          |
 |    |             |     first : String, |          |
|          |            |          |
 |    |             |     last : String   |          |
|          |            |          |
 |    |             |   ))                |          |
|          |            |          |
 +----+-------------+---------------------+----------+-----------
+----------+------------+----------+
 |  5 | age         | Integer             | Y        | NullValue
|          |            |          |
 +----+-------------+---------------------+----------+-----------
+----------+------------+----------+
 |  6 | income      | Integer             | Y        | NullValue
|          |            |          |
 +----+-------------+--------------------+----------+-----------
+----------+------------+----------+
 |  7 | address     | Json               | Y        | NullValue
|          |            |          |
 +----+-------------+--------------------+----------+-----------
+----------+------------+----------+
 |  8 | connections | Array(Integer)     | Y        | NullValue
|          |            |          |
 +----+-------------+--------------------+----------+-----------
+----------+------------+----------+
 |  9 | expenses    | Map(Integer)       | Y        | NullValue
|          |            |          |
 +----+-------------+--------------------+----------+-----------
+----------+------------+----------+
```

Example 5-6   Describe Table

The following statement provides information about the users table and its fields in JSON format.

```
DESC AS JSON TABLE users;

{
  "json_version" : 1,
  "type" : "table",
  "name" : "users",
  "shardKey" : [ "id" ],
  "primaryKey" : [ "id" ],
  "fields" : [ {
    "name" : "id",
    "type" : "INTEGER",
    "nullable" : false,
    "default" : null
  }, {
    "name" : "firstName",
    "type" : "STRING",
    "nullable" : true,
    "default" : null
  }, {
    "name" : "lastName",
    "type" : "STRING",
    "nullable" : true,
    "default" : null
  }, {
    "name" : "otherNames",
    "type" : "ARRAY",
    "collection" : {
      "name" : "RECORD_gen",
      "type" : "RECORD",
      "fields" : [ {
        "name" : "first",
        "type" : "STRING",
        "nullable" : true,
        "default" : null
      }, {
        "name" : "last",
        "type" : "STRING",
        "nullable" : true,
        "default" : null
      } ]
    },
    "nullable" : true,
    "default" : null
  }, {
    "name" : "age",
    "type" : "INTEGER",
    "nullable" : true,
    "default" : null
  }, {
    "name" : "income",
    "type" : "INTEGER",
    "nullable" : true,
    "default" : null
  }, {
    "name" : "address",
    "type" : "JSON",
    "nullable" : true,
    "default" : null
  }, {
    "name" : "connections",
    "type" : "ARRAY",
    "collection" : {
      "type" : "INTEGER"
    },
    "nullable" : true,
    "default" : null
  }, {
    "name" : "expenses",
    "type" : "MAP",
    "collection" : {
      "type" : "INTEGER"
    },
    "nullable" : true,
    "default" : null
  } ]
}
```

Example 5-7   Describe Table

The following statement provides information about a specific field in the users table.

```
DESCRIBE TABLE users (income);

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

Table Hierarchies

The Oracle NoSQL Database enables tables to exist in a parent-child relationship. This is known as table hierarchies.

The create table statement allows for a table to be created as a child of another table, which then becomes the parent of the new table. This is done by using a composite name (a name_path) for the child table. A composite name consists of a number N (N > 1) of identifiers separated by dots. The last identifier is the *local* name of the child table and the first N-1 identifiers are the name of the parent.

Semantics

The semantic implications of a parent-child relationship are the following:

- A child table inherits the primary key columns of its parent table. This is done implicitly, without including the parent columns in the create table statement of the child. For example, in the following Example 5-8 example, table A.B has an extra column, called ida, and its primary key columns are ida and idb. Similarly, table A.B.C has 2 extra columns, ida and idb, and its primary key columns are ida, idb, and idc. The inherited columns are placed first in the schema of a child table.
- All tables in the hierarchy have the same shard key columns, which are specified in the create table statement of the root table. So, in our example, the common shard key is column ida. Trying to include a shard key clause in the create table statement of a non-root table will raise an error.
- A parent table cannot be dropped before its children are dropped.
- When two rows RC and RP from a child table C and its parent table P, respectively, have the same values on their common primary key columns, we say that RP and RC match, or that RP contains RC. In this case, RP and RC will also be co-located physically, because they have the same shard key. Given that a child table always has more primary key columns than its parent, a parent row may contain multiple child rows, but a child row will match with at most one parent row.

Note:

Oracle NoSQL Database does not require that all the rows in a child table have a matching row in the parent table. In other words, a referential integrity constraint is not enforced.

Given that the Oracle NoSQL Database model includes arrays and maps, one may wonder why are child tables needed? After all, for each parent row, its matching child rows could be stored in the parent row itself inside an array or map. However, doing so could lead to very large parent rows, resulting in bad performance. This is especially true given the append-only architecture of the Oracle NoSQL Database store, which implies that a new version of the whole row is created every time the row is updated. So, child tables should be considered when each parent row contains a lot of child rows and/or the child rows are large. If, in addition, the child rows are not accessed very often or if they are updated very frequently, using child tables becomes even more appealing.

Example 5-8 Table Hierarchy

The following statements create a table hierarchy, that is a tree of tables connected by parent-child relationships. A is the root table, A.B and A.G are children of A, and A.B.C is a child of A.B (and a grandchild of A).

```
CREATE TABLE A (
       ida INTEGER, a1 STRING, a2 INTEGER, PRIMARY KEY(ida));
CREATE TABLE A.B (
        idb INTEGER, b1 STRING, a2 STRING, PRIMARY KEY(idb));
CREATE TABLE A.B.C (
        idc INTEGER, b1 STRING, c2 STRING, PRIMARY KEY(idc));
CREATE TABLE A.G (
        idg INTEGER, g1 STRING, g2 DOUBLE, PRIMARY KEY(idg));
```

Using the IDENTITY Column

Declare a column as identity to have Oracle NoSQL Database automatically assign values to it, where the values are generated from a sequence generator. See Sequence Generator.

Syntax

```
identity_definition ::=
GENERATED (ALWAYS | (BY DEFAULT [ON NULL])) AS IDENTITY
["(" sequence_generator_attributes+ ")"]
```

Semantics

An INTEGER, LONG, or NUMBER column in a table can be defined as an identity column. The system can automatically generate values for the identity column using a sequence generator. See Sequence Generator section. A value for an identity column is generated during an INSERT, UPSERT, or UPDATE statement.

An identity column can be defined either as GENERATED ALWAYS or GENERATED BY DEFAULT.

GENERATED ALWAYS
The system always generates a value for the identity column. An exception is raised if the user supplies a value for the identity column.

GENERATED BY DEFAULT
The system generates a value for the identity column only if the user does not supply a value for it. If ON NULL is specified for GENERATED BY DEFAULT, the system will generate a value when the user supplies a NULL value or the value evaluates to a NULL.

Identity Column Characteristics

- There can be only one identity column per table.

- The identity column of a table can be part of the primary key or the shard key.
- Secondary indexes can be created on an identity column.
- The set of values that may be assigned to an identity column is defined by its data type and the attributes of the sequence generator attached to it. The values are always integer numbers. Both negative and positive INTEGER are possible. If you want only positive values, then set the START WITH attribute to 1 and specify a positive INCREMENT BY attribute. When you specify CYCLE, numbers will be regenerated from the MINVALUE. In this case, if you want positive values you must also set MINVALUE to be a positive number.
- The system generates unique values for an identity column that is defined as GENERATED ALWAYS and has the sequence generator attribute NO CYCLE set. Otherwise, duplicate identity values can occur in the following scenarios:
  - The identity column is defined as GENERATED BY DEFAULT and the user supplies a value during an insert or update statement that already exists in the table for the identity column.
  - The CYCLE option is set for an identity column that is defined as GENERATED BY DEFAULT or GENERATED ALWAYS and the sequence generator reaches the end of the cycle and then recycles through the sequence generator to generate values that were generated in the previous cycle.
  - If the identity column properties are altered using the alter table statement so that during an insert or update operation the user can supply a value that already exists.
- Sequence generator attributes can be altered using the alter table statement.
- Holes in the sequence can occur when:
  - The application caches identity values and shuts down or crashes before using all of the cached values for inserting rows.
  - Identity values are assigned during a transaction that is rolled back.
- For example on inserting rows with an identity column see, Inserting Rows with an IDENTITY Column section.

Example 5-9 Identity Column using GENERATED ALWAYS

```
CREATE TABLE T1 (
    id INTEGER GENERATED ALWAYS AS IDENTITY
    (START WITH 2 INCREMENT BY 2 MAXVALUE 200 NO CYCLE),
    name STRING,
    PRIMARY KEY (id)
);
```

In the above example, the `INTEGER` column id is defined as a GENERATED ALWAYS AS IDENTITY column and is the primary key for table T. The system will start to generate values 2 through 200 incrementing by 2. So values for the id column will be 2,4,6,8,…200. Since the NO CYCLE option is defined, the system will raise an exception after the number 200 is generated saying it has reached the end of the sequence generator.

Example 5-10 Identity Column using GENERATED BY DEFAULT

```
CREATE TABLE T2 (
    id LONG GENERATED BY DEFAULT AS IDENTITY
    (START WITH 1 INCREMENT BY 1 CYCLE CACHE 200),
    account_id INTEGER,
    name STRING,
    PRIMARY KEY (account_id)
);
```

In the above example, the creation of a table with an identity column on id column is shown. The id column is of type LONG, is defined as GENERATED BY DEFAULT, and it is not a primary key column. This example also demonstrates how to specify a CYCLE and CACHE sequence generator attributes. The system will only generate a value during INSERT/UPSERT/UPDATE if the user did not supply a value. It starts off generating values 1, 2, 3,... up to the maximum value of the LONG datatype, and once it exhausts all the sequence generator values, it will cycle through and re-start from the MINVALUE value of the sequence generator, which in this case, is the minimum value of the LONG datatype. The CACHE value of 200 means that every time a client uses up the values in the cache and asks for the next value, the system will give it 200 values to fill up the cache. In this example, the system will give values 1 through 200 when a client asks for a value for the first time. Another client operating on the same table may get values 201-300, so on and so forth.

Sequence Generator

The sequence generator is a service that generates a sequence of integer numbers.

Syntax

```
sequence_generator_attributes ::=
   (START WITH signed_int) |
   (INCREMENT BY signed_int) |
   (MAXVALUE signed_int) | (NO MAXVALUE) |
   (MINVALUE signed_int) | (NO MINVALUE) |
   (CACHE INT) | (NO CACHE) |
   CYCLE | (NO CYCLE)
```

Semantics

Oracle NoSQL Database only supports sequence generators that are attached to identity columns. See Using the IDENTITY Column.The numbers in the generated sequence depend on the attributes of the sequence generator attributes. The following attributes are supported:

START WITH
The first number in the sequence. The default is 1.

INCREMENT BY
The next number in the sequence is generated by adding INCREMENT BY value to the current value of the sequence. The increment value can be a positive number or a negative number. Sequence generator having a positive increment is ascending and a

sequence generator having a negative increment is descending. The default is 1. The value 0 is invalid.

MINVALUE or NO MINVALUE
The lower bound of the sequence range. The default is -2^31 which is the minimum value of the integer datatype.

MAXVALUE or NO MAXVALUE
The upper bound of the sequence range. The default 2^31-1 which is the maximum value of the integer datatype.

CYCLE or NO CYCLE
Specify this to indicate that the sequence continues to generate values after reaching either its maximum or minimum value. After an ascending sequence reaches its maximum value, it generates its minimum value. After a descending sequence reaches its minimum, it generates its maximum value.

CACHE
The value of this attribute is the number of sequence numbers that will be generated every time a request is made to the sequence generator. These requests originate at the Oracle NoSQL Database clients and are serviced by the sequence generator, which "lives" at the server. Specifically, the numbers generated in each request are sent back to the client and are cached there. Whenever a client needs to assign a value to an identity column, the next sequence number from the cache is consumed. When cache empties, a request for another batch of CACHE sequence numbers is sent to the sequence generator. The default CACHE size is 1000.

DROP TABLE Statement

The drop table statement removes the specified table and all its associated indexes from the database.

Syntax

```
drop_table_statement ::= DROP TABLE [IF EXISTS] name_path
```

Semantics

IF EXISTS
By default, if the named table does not exist then this statement fails. If the optional IF EXISTS is specified and the table does not exist then no error is reported.

IDENTITY
When a table with an identity column is dropped, the associated sequence generator is also removed.

Example 5-11   Drop Table

```
CREATE TABLE DROPTEST (id INTEGER, name STRING, PRIMARY KEY(id));

DROP TABLE DROPTEST;
```

ALTER TABLE Statement

You can add, remove or modify a schema field from the table schema using the ALTER TABLE statement. You can also change the default Time-To-Live value of the table.

Syntax

```
alter_table_statement ::=
   ALTER TABLE name_path (alter_field_statements | ttl_definition)

alter_field_statements ::=
   "(" alter_field_statement ("," alter_field_statement)* ")"

alter_field_statement ::=
   add_field_statement | drop_field_statement | modify_field_statement |
alter_regions_statement

add_field_statement ::=
   ADD schema_path type_definition
   [default_definition | identity_definition]
   [comment]

drop_field_statement ::= DROP schema_path

modify_field_statement ::=
   (MODIFY schema_path identity_definition) |
   (DROP IDENTITY)

alter_regions_statement ::= add_regions_statement |
drop_regions_statement

add_regions_statement ::= ADD REGIONS region_names

drop_regions_statement ::= DROP REGIONS region_names

region_names ::= region_name ["," region_name]*

schema_path ::= init_schema_path_step ("." schema_path_step)*
init_schema_path_step ::= id ("[" "]")*
    schema_path_step ::= id ("[" "]")* | VALUES "(" ")"
```

Semantics

modify_field_statement
You can use the MODIFY keyword to modify only an identity column.

add_field_statement
Adding a field does not affect the existing rows in the table. If a field is added, its default value or NULL will be used as the value of this field in existing rows that do not contain it. The field to add may be a top-level field (i.e. A table column) or it may be deeply nested inside a hierarchical table schema. As a result, the field is specified via

a path. The path syntax is a subset of the one used in queries and is described in the Path Expressions section.

drop_field_statement
Dropping a field does not affect the existing rows in the table. If a field is dropped, it will become invisible inside existing rows that do contain the field. The field to drop may be a top-level field (i.e. A table column) or it may be deeply nested inside a hierarchical table schema. As a result, the field is specified via a path. The path syntax is a subset of the one used in queries and is described in the Path Expressions section.

add_regions_statement
The add regions clause lets you link an existing MR Table with new regions in a multi-region Oracle NoSQL Database environment. This clause is used in expanding MR Tables to new regions. See Use Case 2: Expand a Multi-Region Table in the *Administrator's Guide*.

drop_regions_statement
The drop regions clause lets you disconnect an existing MR Table from a participating region in a multi-region Oracle NoSQL Database environment. This clause is used in contracting MR Tables to fewer regions. See Use Case 3: Contract a Multi-Region Table Use Case 3: Contract a Multi-Region Table in the *Administrator's Guide*.

Example 5-12 Alter Table

The following example adds a middle name into the names stored in other_names.

```
ALTER TABLE users (ADD otherNames[].middle STRING);
```

Altering an IDENTITY Column

An identity column can be modified using the alter table statement.

There are two ways to alter an identity column:

- The attributes of the sequence generator associated with an identity column can be altered. The sequence generator is modified immediately with the new attributes, however, a client will see the effects of the new attributes on the sequence numbers generated on subsequent requests by the client to the sequence generator, which will happen when the cache is used up or the attributes stored at the client time out.

  Note:

  Each client has a time-based cache to store the sequence generator attributes. The client will connect to the server to refresh this cache once it expires. The default timeout is 5 mins and it can be changed by setting sgAttrsCacheTimeout in KVStoreConfig.

  The Following example shows how to alter the property of the identity column id from GENERATED ALWAYS to GENERATED BY DEFAULT and altering sequence generator attributes START WITH, INCREMENT BY, MAXVALUE and CACHE.

- The identity property of an existing identity column can be dropped. The sequence generator attached to that identity column is also removed. The system will no longer generate a value for that column.

Example 5-13   Alter Identity Column

```
CREATE TABLE AlterTableExample (
    id INTEGER GENERATED ALWAYS AS IDENTITY
    (START WITH 1
    INCREMENT BY 2
    MAXVALUE 100
    CACHE 1
    CYCLE),
    name STRING,
PRIMARY KEY(id)
);
INSERT INTO AlterTableExample VALUES (DEFAULT, "John" );
SELECT * FROM AlterTableExample;

ALTER TABLE AlterTableExample ( MODIFY
    id GENERATED BY DEFAULT AS IDENTITY
    (START WITH 1000
    INCREMENT BY 3
    MAXVALUE 5000
    CACHE 1
    CYCLE)
);

INSERT INTO AlterTableExample VALUES (DEFAULT, "Peter" );
INSERT INTO AlterTableExample VALUES (DEFAULT, "Mary" );
SELECT * FROM AlterTableExample;
```

6
SQL Query Management

You can use a query to retrieve data from one or more tables. A query is a statement that consists of zero or more variable declarations followed by single SELECT expression. The result of a query is always a sequence of records having the same record type.

> **Note:**
>
> Subqueries are not supported in Oracle NoSQL Database.

Syntax

```
query ::= [variable_declaration] select_expression
```

Variable declarations and expressions will be defined later in this chapter. Before doing so, a few general concepts and related terminology must be established first.

Expressions

An **expression** represents a set of operations to be executed in order to produce a result. The various kinds of expressions supported by Oracle NoSQL Database are described later in this chapter.

Expressions are built by combining other expressions and sub-expressions via operators, function calls, or other gramatical constructs. The simplest kind of expressions are constants and references to variables or identifiers.

Sequences

A **sequence** is a set of zero or more items. All expressions operate on zero or more input sequences and produce an ouput sequence as their result.

A sequence is just a collection set of zero or more items (including NULLs). A sequence is not an item itself (so no nested sequences) nor is it a container: there is neither a persistent data structure nor a java class at the public API level (or internally) that represents a sequence. Expressions usually operate on sequences by iterating over their items.

> **Note:**
>
> An array is not a sequence of items. Instead, it is a single item, albeit one that contains other items in it. So, arrays are containers of items.

Although, in general, Oracle NoSQL Database expressions work on sequences and produce sequences, many of them place restrictions on the cardinality of the sequences they accept and/or produce. For example, several expressions are scalar: they require that their input sequence(s) contain no more than one item and they never produce a sequence of more than one item. Notice that a single item is considered equivalent to a sequence containing only that single item.

Boxing and Unboxing Sequence

A sequence produced by an expression E can be converted to an array by wrapping E with an array constructor : [E]. See Array and Map Constructors section. This is called **boxing** the sequence. Conversely, there are expressions that **unbox** an array: they select all or a subset of the items contained in the array and return these items as a sequence. There is no implicit unboxing of arrays; an expression must always be applied to do the unboxing. In most cases, sequence boxing must also be done explicitly, that is, the query writer must use an array constructor. There are, however, a couple of cases where boxing is done implicitly, that is, an expression (which is not an array constructor) will convert an input sequence to an array.

Note:

In standard SQL the term "expression" means "scalar expression", i.e., an expression that returns exactly one (atomic) item. The only operations that can produce more than one items (or zero items) are query blocks (either as top-level queries or subqueries) and the set operators like union, intersection, etc (in these cases, the items are tuples). In Oracle NoSQL Database too, most expressions are scalar. Like the query blocks of standard SQL, the select-form-where expression of Oracle NoSQL Database returns a sequence of items. However, to navigate and extract information from complex, hierarchical data, Oracle NoSQL Database includes path expressions as well. See Path Expressions section. Path expressions are the other main source of multi-item sequences in Oracle NoSQL Database. However, if path expressions are viewed as subqueries, the Oracle NoSQL Database model is not that different from standard SQL.

Sequence Types

A sequence type specifies the type of items that may appear in a sequence, as well as an indication about the cardinality of the sequence.

Syntax

```
sequence_type ::= type_definition [quantifier]

quantifier := "*" | "+" | "?"
```

Semantics

quantifier
The quantifier is one of the following:

- ***** 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.

subtype relationship

A subtype relationship exists among sequence types as well. It is defined as follows:

- The empty sequence is a subtype of all sequence types whose quantifier is * or ?
- A sequence type SUB is a subtype of another sequence type SUP (supertype) if SUB's item type is a subtype of SUP's item type, and SUB's quantifier is a subquantifier of SUP's quantifier, where the subquantifier relationship is defined by the following matrix.

The following matrix illustrates the subquantifier relationship between the quantifiers. The column heading indicate the supertype(SUP) of the quantifier. The row heading indicate the subtype (SUB) of the quantifier.

Sup Q1 | Sub Q2	**one**	**?**	**+**	*****
one	true	false	false	false
?	true	true	false	false
+	true	false	true	false
*****	true	true	true	true

For example, as per the above table, **?** is a superquantifier of **one** and **?**, but is not a superquantifier of **+** and *****. Similarly, ***** is a superquantifier of all other quantifiers.

Note:

In the following sections, when we say that an expression must have (sequence) type T, what we mean is that when the expression is evaluated, the result sequence must have type T or any subtype of T. Similarly, the usual subtype-substitution rules apply to input sequences: if an expression expects as input a sequence of type T, any subtype of T may actually be used as input.

Variable Declaration

Syntax

```
variable_declaration ::= DECLARE (variable_name type_definition ";")+

    variable_name ::= "$" id
```

External Variables

A query may start with a variable declaration section. The variables declared here are called external variables. The value of an external variable is global and constant. The values of external variables are not known in advance when the query is formulated or compiled. Instead, the external variables must be bound to their actual values before the query is executed. This is done via programmatic APIs. See *Java Direct Driver Developer's Guide*.

The type of the item bound to an external variable must be equal to or a subtype of the variable's declared type. The use of external variables allows the same query to be compiled once and then executed multiple times, with different values for the external variables each time. All the external variables that appear in a query must be declared in the declaration section. This is because knowing the type of each external variable in advance is important for query optimization.

Note:

External variables play the role of the global constant variables found in traditional programming languages (e.g. final static variables in java, or const static variables in c++).

Internal Variables

Oracle NoSQL Database allows implicit declaration of internal variables as well. Internal variables are bound to their values during the execution of the expressions that declare them.

Variables (internal and external) can be referenced in other expressions by their name. In fact, variable references themselves are expressions, and together with literals, are the starting building blocks for forming more complex expressions.

Scope

Each variable is visible (i.e., can be referenced) within a scope. The query as a whole defines the global scope, and external variables exist within this global scope. Certain expressions create sub-scopes. As a result, scopes may be nested. A variable declared in an inner scope hides another variable with the same name that is declared in an outer scope. Otherwise, within any given scope, all variable names must be unique.

Note:

The names of variables are case-sensitive.

Note:

The following variable names cannot be used as names for external variables: $key, $value, $element, and $pos.

Example 6-1 Variable Declaration

The following query selects the first and last names of all users whose age is greater than the value assigned to the $age variable when the query is actually executed.

```
DECLARE $age INTEGER;

SELECT firstName, lastName
FROM Users
WHERE age > $age;
```

SELECT Expression

You can query data from the tables using the SELECT expression. Multiple clauses can be used with the SELECT expression. The clauses that can be used in the SELECT expression are given in the syntax below.

Syntax

```
select_expression ::=
   SELECT Clause
   from_clause
   [where_clause]
   [groupby_clause]
   [orderby_clause]
   [limit_clause]
   [offset_clause]
```

Semantics

The SELECT clause and the FROM clause are mandatory.

The processing of the query starts with the FROM clause, followed by the WHERE clause (if any), followed by the GROUP BY clause (if any), followed by the ORDER BY clause (if any), followed by the SELECT clause and finishing with the OFFSET and LIMIT clauses (if any). Each clause produces a set of records, which is processed by the next clause. Each clause is described in the following sections.

FROM Clause

The FROM clause is used to retrieve rows from the referenced table(s).

Syntax

```
from_clause ::= FROM (single_from_table | nested_tables)

single_from_table ::= aliased_table_name

aliased_table_name ::=
   (table_name | SYSTEM_TABLE_NAME) [[AS] table_alias]

table_alias ::= [$] id
```

Semantics

As shown in the syntax, the FROM clause can either reference a single table or include a nested tables clause. For nested tables, see the Joining Tables in the Same Table Hierarchy section.

single_from_table
In a simple FROM clause, the table is specified by its name, which may be a composite (dot-separated) name in the case of child tables. The result of the simple FROM clause is a sequence containing the rows of the referenced table.

aliased_table_name
The table name may be followed by a table alias. Table aliases are essentially variables ranging over the rows of the specified table. If no alias is specified, one is created internally, using the name of the table as it is spelled in the query, but with dot chars replaced with '_' in the case of child tables. See Table Hierarchies.

Note:

Table aliases are case-sensitive, like variable names.

The other clauses of the SELECT expression 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 columns of the context row can be referenced in expressions either directly by their names or by the table alias followed by a dot char and the column name. See the Column References section. If the table alias starts with a dollar sign ($), then it actually serves as a variable declaration for a variable whose name is the alias. This variable is bound to the context row as a whole and can be referenced within sub expressions of the SELECT expression. For example, it can be passed as an argument to the expiration_time function to get the expiration time of the context row. See the timestamp(0) expiration_time(AnyRecord) function. In other words, a table alias like $foo is an expression by itself, whereas foo is not. Notice that if this variable has the same name as an external variable, it hides the external variable. This is because the FROM clause creates a nested scope, which exists for the rest of the SELECT expression.

Example 6-2 From Clause

```
SELECT * FROM users;
```

The above example selects all information for all users.

WHERE Clause

The WHERE clause filters the rows coming from the FROM clause, returning the rows satisfying a given condition.

Syntax

```
where_clause ::= WHERE expression
```

Semantics

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.

Example 6-3 WHERE Clause

```
SELECT * FROM users WHERE firstname ="John";
```

The above example selects all information for users whose first name is "John".

GROUP BY Clause

The GROUP BY clause is used in a SELECT statement to collect data across multiple rows and group the result by one or more columns or expressions. The GROUP BY clause is often used with aggregate functions. Oracle NoSQL Database applies the aggregate functions to each group of rows and returns a single row for each group.

Syntax

```
groupby_clause ::= GROUP BY expression ("," expression)*
```

Semantics

Each (grouping) expression must return at most one atomic value. If a grouping expression returns an empty result on an input row, that row is skipped. Equality among grouping values is defined according to the semantics of the "=" operator, with the exception that two NULL values are considered equal. See Value Comparison Operators section. Then, for each group, a single record is constructed and returned by the GROUP BY clause. If the clause has N grouping expressions, the first N fields of the returned record store the values of the grouping expressions. The remaining M fields (M >= 0) store the result of zero or more aggregate functions. In general, aggregate functions iterate over the rows of a group, evaluate an expression for each such row, and aggregate the returned values into a single value per group. Oracle NoSQL Database supports many aggregate functions as described in the Using Aggregate Functions section.

Syntactically, aggregate functions are not actually listed in the GROUP BY clause, but appear in the SELECT clause instead. In fact, aggregate functions can appear only in the SELECT or ORDER BY clauses, and they cannot be nested. Semantically, however, every aggregate function that appears in the SELECT or ORDER BY list is actually evaluated by the GROUP BY clause. If the SELECT clause contains any aggregate functions, but the SELECT expression does not contain a GROUP BY clause, the whole set of rows produced by the FROM or the WHERE clauses is considered as one group and the aggregate functions are evaluated over this single group.

The implementation of the GROUP BY clause may be index-based or generic. Index-based grouping is possible only if an index exists that sorts the rows by the values of the grouping expressions, see the Simple Indexes section. More precisely, let e*1*, e*2*, …, e*N* where e*i* is the *i*th expression (i is a number in the range 1,2,3, ...N) be the grouping expressions as they appear in the GROUP BY clause (from left to right). Then, for index-based grouping, there must exist an index (which may be the primary-key index or one of the existing secondary indexes) such that for each *i* in 1,2,...,N, e*i* matches the definition of the *i*-th index field. If such an index does not exist or is not selected by the query optimizer, the GROUP BY will be generic. A generic GROUP BY uses a hash table to find rows belonging to the same group and stores all groups before returning any results to the application. The hash table is stored in the client driver memory (local hash tables, of limited size, may be used at the servers as well). As a result, a generic GROUP BY may consume a large amount of driver memory. In contrast, index-based grouping exploits the row sorting provided by the index to avoid the materialization and caching of any intermediate results. It is therefore recommended to create appropriate indexes for use in GROUP BY queries. See Using Indexes for Query Optimization. Finally, notice that when you use index-based grouping, the results of a grouping SELECT expression are ordered by the grouping expressions.

Example 6-4 GROUP BY Clause

This query groups users by their age and for each group returns the associated age and the average income of the users in the group. Grouping is possible only if there is a secondary index on the age column (or more generally, a multi-column index whose first column is the age column).

```
SELECT
age,
count(*) AS count,
avg(income) AS income
FROM users
GROUP BY age;
```

Using Aggregate Functions

You can use built in aggregate functions to find information such as a count, a sum, an average, a minimum, or a maximum.

The following functions are called SQL aggregate functions, because their semantics are similar to those in standard SQL: they work in conjunction with grouping and they aggregate values across the rows of a group.

- long count(*)
- long count(any*)
- number sum(any*)
- number avg(any*)
- any_atomic min(any*)
- any_atomic max(any*)

long count(*)

The count star function returns the number of rows in a group.

long count(any*)

The count function computes its input expression on each row in a group and counts all the non-NULL values returned by these evaluations of the input expression.

number sum(any*)

The sum function computes its input expression on each row in a group and sums up all the numeric values returned by these evaluations of the input expression. Any non-numeric values are skipped. However, if it can be determined at compile time that the input expression does not return any numeric values, an error is thrown. The resulting value has type long, double, or number, depending on the type of the input items: if there is at least one input item of type number, the result will be a number, otherwise if there is at least one item of type double or float, the result will be double, otherwise the result will be a long. Finally, if no numeric values are returned by sum's input, the result is NULL.

number avg(any*)

The avg (average) function computes its input expression on each row in a group and sums up as well as counts all the numeric values returned by these evaluations of the input expression. Any nonnumeric values are skipped. However, if it can be determined at compile time that the input expression does not return any numeric values, an error is thrown. The resulting value is the division of the sum by the count. This value has type double, or number, depending on the type of the input items: if there is at least one input item of type number, the result will be a number, otherwise the result will be double. Finally, if no numeric values are returned by avg's input, the result is NULL.

any_atomic min(any*)

The min function returns the minimum value among all the values returned by the evaluations of the input expression on each row in a group. More specifically: If it can be determined at compile time that the values returned by the input expression belong to a type for which an order comparison is not defined (i.e., RECORD, MAP, BINARY, or FIXED_BINARY), an error is thrown. Otherwise, the min value for each group is initialized to NULL. Next, let M be the current minimum value and N be the next input value. If N is a record, map, array, binary or fixed binary value, NULL, or the json null, it is skipped. If M is NULL, M is set to N. Else, if N is less than M, M is set to N. In comparing the two values M and N, the rules of the value comparison operator are used when the values are comparable according to the same rules. See Value Comparison Operators. When the values are not comparable, the following order is used: numeric values < timestamps < strings and enums < booleans. Notice that according to these rules, the min function will return NULL if and only if all the input values in a group are records, maps, arrays, binary or fixed binary values, NULL, or the json null.

any_atomic max(any*)

The max function returns the maximum value in all the sequences returned by the evaluations of the input expression on each row in a group. The specific rules are the same as for the min function, except that the current max value M will be replaced by the next input value N if N is not skipped and is greater than M.

Example 6-5 Aggregate Function

```
CREATE INDEX idx11 ON users (age);

SELECT
age, count(*) AS count, avg(income) AS income
```

```
FROM users
GROUP BY age;
```

The above query groups users by their age, and for each age, return the number of users having that age and their average income.

Sequence Aggregate Functions

Sequence aggregate functions simply aggregate the items in their input sequence, using the same rules as their corresponding SQL aggregate function. Sequence aggregate functions can be used to aggregate the values of an array or map inside the current context row, without any grouping across rows.

For example, seq_sum() will skip any non-numeric items in the input sequence and it will determine the actual type of the return value (long, double, or number) the same way as the SQL sum(). The only exception is seq_count(), which contrary to the SQL count(), will return NULL if any of its input items is NULL. Furthermore, there are no restrictions on where sequence aggregate functions can appear (for example, they can be used in the WHERE and/or the SELECT clause). An example using seq_sum and seq_max is given in Example 6-30.

The following functions are the sequence aggregate functions.

- long seq_count(any*)
- number seq_sum(any*)
- number seq_avg(any*)
- any_atomic seq_min(any*)
- any_atomic seq_max(any*)

long seq_count(any*)

Returns the number of items in the input sequence. See long count(any*) for details.

number seq_sum(any*)

Returns the sum of the numeric items in the input sequence. See number sum(any*) for details.

number seq_avg(any*)

Returns the average of the numeric items in the input sequence. See number avg(any*) for details.

any_atomic seq_min(any*)

Returns the minimum of the items in the input sequence. See any_atomic min(any*) for details.

any_atomic seq_max(any*)

Returns the minimum of the items in the input sequence. See any_atomic max(any*) for details.

Example 6-6 Sequence Aggregate Function

```
SELECT id, seq_sum(u.expenses.values()) FROM users u;
```

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, taking into account the sort_spec associated with each ORDER BY expression.

Syntax

```
orderby_clause ::= ORDER BY
   expression sort_spec
   ("," expression sort_spec)*

sort_spec ::= [ASC|DESC] [NULLS (FIRST|LAST)]
```

Semantics

Each ordering expression must return at most one atomic value. If an ordering expression returns an empty sequence, the special value EMPTY is used as the returned value. If the SELECT expression includes GROUP BY as well, then the expressions in the ORDER BY must be the grouping expressions (in the GROUP BY clause, if any), or aggregate functions, or expressions that are built on top of grouping expression and/or aggregate functions.

sort_spec : A sort_spec specifies the "direction" of the sort (ascending or descending) and how to compare the special values NULL, JNULL, and EMPTY with the non-special values.

- If NULLS LAST is specified, the special values will appear after all the non-special values.
- If NULLS FIRST is specified, the special values will appear before all the non-special values.

The relative ordering among the 3 special values themselves is fixed:

- if the direction is ASC, the ordering is EMPTY < JNULL < NULL;
- otherwise the ordering is reversed.

Notice that in the grammar, sort_specs are optional.

- If no sort_spec is given, the default is ASC order and NULLS LAST.
- If only the sort order is specified, then NULLS LAST is used if the order is ASC, otherwise NULLS FIRST.
- If the sort order is not specified, ASC is used.

Taking into account the above rules, the relative order between any two input rows is determined as follows. Let *N* be the number of order-by expressions and let V*i1*, V*i2*, … V*iN* be the atomic values (including EMPTY) returned by evaluating these expressions, from left to right, on a row R*i*. Two rows R*i*, R*j* are considered equal if V*ik* is equal to V*jk* for each *k* in 1, 2, …, N. In this context, NULLs are considered to be equal only to themselves. Otherwise, R*i* is considered less than R*j* if there is a pair V*im*, V*jm* such that:

- *m* is 1, or V*ik* is equal to V*jk* for each *k* in 1, 2, …, (*m*-1), and
- V*im* is not equal to V*jm*, and
- the *m*-th sort_spec specifies ascending order and V*im* is less than V*jm*, or
- the *m*-th sort_spec specifies descending order and V*im* is greater than V*jm*

In the above rules, comparison of any two values V*ik* and V*jk*, when neither of them is special and they are comparable to each other, is done according to the rules of the value-comparison operators defined in the Value Comparison Operators section.

If V*ik* and V*jk* do not have comparable types (which, for example, - can arise when sorting by json fields), the following rule applies:

- If the direction is ASC, the ordering is numeric items < timestamps < strings and enums < booleans.
- Otherwise the ordering is reversed.

As with grouping, sorting can be index-based or generic. Index-based sorting is possible only if there is an index that sorts the rows in the desired order. More precisely, let e*1*, e*2*, …, e*N* by the ORDER BY expressions as they appear in the ORDER BY clause (from left to right). Then, there must exist an index (which may be the primary-key index or one of the existing secondary indexes) such that for each *i* in 1,2,...,N, ei matches the definition of the *i*-th index field. Furthermore, all the sort_specs must specify the same ordering direction and for each sort_spec, the desired ordering with respect to the special values must match the way these values are sorted by the index. In the current implementation, the special values are always sorted last in an index. So, if the sort order is ASC, all sort_specs must specify NULL LAST, and if the sort order is DESC, all sort_specs must specify NULLS FIRST.

Note:

If no appropriate index exists or is not selected by the query optimizer, the sorting will be generic. This implies that all query results must be fetched into the driver memory and cached there before they can be sorted. So, as with grouping, generic sorting can consume a lot of driver memory, and is therefore best avoided.

For both generic ORDER BY and GROUP BY, applications can programmatically specify how much memory such operations are allowed to consume at the client driver. We have specific methods for this functionality in each of the available language drivers as given below.

Table 6-1 APIS for Memory Consumption

Language Driver	Get maximum memory consumption	Set maximum memory consumption
Java	getMaxMemoryConsumption()	setMaxMemoryConsumption(long v)
Python	get_max_memory_consumption()	set_max_memory_consumption (memory_consumption)

Table 6-1 (Cont.) APIS for Memory Consumption

Language Driver	Get maximum memory consumption	Set maximum memory consumption
Node.js	maxMemoryMB	maxMemoryMB
Go	GetMaxMemoryConsumption()	MaxMemoryConsumption

Example 6-7 ORDER BY Clause

This example selects the id and the last name for users whose age is greater than 30, returning the results sorted by id. Sorting is possible in this case because id is the primary key of the users table.

```
SELECT id, lastName
FROM users
WHERE age > 30
ORDER BY id;
```

Example 6-8 ORDER BY Clause

This example selects the id and the last name for users whose age is greater than 30, returning the results sorted by age. Sorting is possible only if there is a secondary index on the age column (or more generally, a multi-column index whose first column is the age column).

```
SELECT id, lastName
FROM users
WHERE age > 30
ORDER BY age;
```

Example 6-9 ORDER BY Clause

The following example returns all the rows sorted by the first name.

```
SELECT id, firstName, lastName
    FROM users
    ORDER BY firstName;
 +----+-----------+----------+
 | id | firstName | lastName |
 +----+-----------+----------+
 | 10 | John      | Smith    |
 | 20 | Mary      | Ann      |
 | 30 | Peter     | Paul     |
 +----+-----------+----------+
3 rows returned
```

Example 6-10   ORDER BY Clause

The following example returns the firstName, lastName and income sorted by the income from highest to lowest.

```
SELECT firstName, lastName, income
    from users
    ORDER BY income DESC;
 +-----------+----------+--------+
 | firstName | lastName | income |
 +-----------+----------+--------+
 | Mary      | Ann      |  90000 |
 | Peter     | Paul     |  53000 |
 | John      | Smith    |  45000 |
 +-----------+----------+--------+
3 rows returned
```

Example 6-11   ORDER BY Clause

The following example groups the data by age and returns the number of users having that age and their average income ordered by their average income.

```
SELECT
    age, count(*), avg(income)
    FROM users
    GROUP BY age
    ORDER BY avg(income);
 +-----+----------+----------+
 | age | Column_2 | Column_3 |
 +-----+----------+----------+
 |  22 |        1 |  45000.0 |
 |  25 |        1 |  53000.0 |
 |  43 |        1 |  90000.0 |
 +-----+----------+----------+
3 rows returned
```

Example 6-12   ORDER BY Clause

In the following example, Query 1 returns the state and income sorted by income. However, if we want to group Query 1 by state, then we can use the GROUP BY clause. However, when a SELECT expression includes grouping, expressions in the SELECT and ORDER BY clauses must reference grouping expressions, aggregate functions or external variable only. So, to get the desired result, we need to rewrite Query 1 as given in Query 2.

```
Query 1:

SELECT
    u.address.state, u.income
    FROM users u
    ORDER BY u.income;
 +---------------+--------+
 |     state     | income |
 +---------------+--------+
 | NV            |  45000 |
 +---------------+--------+
 | CA            |  53000 |
 +---------------+--------+
 | CA            |  90000 |
 +---------------+--------+
3 rows returned

Query 2:

SELECT
    u.address.state, max(u.income)
    FROM users u
    GROUP BY u.address.state
    ORDER BY max(u.income);
 +---------------+----------+
 |     state     | Column_2 |
 +---------------+----------+
 | NV            |    45000 |
 +---------------+----------+
 | CA            |    90000 |
 +---------------+----------+
2 rows returned
```

Example 6-13   ORDER BY Clause

In the following example, the Query 1 returns the income and state of all the rows in the users table. The Query 2 gets the average income for each state.

```
Query 1:

SELECT
    u.address.state, u.income
    FROM users u;
 +---------------+--------+
 |     state     | income |
 +---------------+--------+
 | CA            |  53000 |
 +---------------+--------+
 | NV            |  45000 |
 +---------------+--------+
 | CA            |  90000 |
 +---------------+--------+
3 rows returned

Query 2:

SELECT
    u.address.state, avg(u.income)
    FROM users u
    GROUP BY u.address.state
    ORDER BY avg(u.income);
 +---------------+----------+
 |     state     | Column_2 |
 +---------------+----------+
 | NV            |  45000.0 |
 +---------------+----------+
 | CA            |  71500.0 |
 +---------------+----------+
2 rows returned
```

SELECT Clause

The SELECT clause transforms each input row to a new record that will appear in the query result. The SELECT clause comes in two forms: "select star" form and "projection" form.

select star form
In select star form the SELECT clause contains a single star symbol (*). In this the SELECT clause is a no-op; it simply returns its input sequence of rows.

projection form
In the projection form the SELECT clause contains a list of expressions, where each expression is optionally associated with a name. In this the listed expressions and their associated names are refered as field expressions and field names respectively.

Syntax

```
select_clause ::= SELECT [DISTINCT] select_list

select_list ::= [hints]
   (STAR | (expression AS id ("," expression AS id)*))
```

Semantics

In projection form, the SELECT clause creates a new record for each input row. In this the record constructed by the SELECT clause has one field for each field expression and the fields are arranged in the same order as the field expressions. For each field, its value is the value computed by the corresponding field expression and its name is the name specified by the AS keyword, or if no field name is provided explicitly (via the AS keyword), one is generated internally during query compilation. To create valid records, the field names must be unique. Furthermore, each field value must be exactly one item. To achieve this, the following two implicit conversions are employed:

1. If the result of a field expression is empty, NULL is used as the value of the corresponding field in the created record.
2. If the compiler determines that a field expression may return more than one item, it wraps the field expression with a conditional array constructor. See the Array and Map Constructors section. During runtime, an array will be constructed only if the field expression does actually return more than one item; if so, the returned items will be inserted into the constructed array, which will then be used as the value of the corresponding field in the created record.

The above semantics imply that all records generated by a SELECT clause have the same number of fields and the same field names. As a result, a record type can be created during compilation time that includes all the records in the result set of a query. This record type is the type associated with each created record, and is available programmatically to the application.

The SELECT clause can contain an optional DISTINCT keyword. If the SELECT clause contains the DISTINCT keyword, then the database will return only one copy of each set of duplicate rows selected. Duplicate rows are those with matching values in the SELECT list. The query uses the combination of values in all specified columns in the SELECT list to evaluate the uniqueness. See the Example 6-19 example. Equality between values is checked using the semantics of the "=" operator. See the Value Comparison Operators section.

If the SELECT expression is a grouping one, then the expressions in the SELECT list must be the grouping expressions (in the GROUP BY clause, if any), or aggregate functions, or expressions that are built on top of grouping expression and/or aggregate functions.

Note:

The SELECT clause may also contain one or more hints, that help the query processor choose an index to use for the query. See the Choosing the Best Applicable Index section.

Example 6-14 SELECT Clause

```
SELECT * FROM users;
```

Example 6-15 SELECT Clause

Select the id and the last name for users whose age is greater than 30. We show 4 different ways of writing this query, illustrating the different ways that the top-level columns of a table may be accessed.

```
SELECT id, lastName FROM users WHERE age > 30;

SELECT users.id, lastName FROM users WHERE users.age > 30;

SELECT $u.id, lastName FROM users $u WHERE $u.age > 30;

SELECT u.id, lastName FROM users u WHERE users.age > 30;
```

Example 6-16 SELECT Clause

Select the id and the last name for users whose age is greater than 30, returning the results sorted by id. Sorting is possible in this case because id is the primary key of the users table.

```
SELECT id, lastName FROM users WHERE age > 30 ORDER BY id;
```

Example 6-17 SELECT Clause

Select the list of distinct age of the users.

```
SELECT DISTINCT age FROM users;

{"age":25}
{"age":43}
{"age":22}
```

Example 6-18 SELECT Clause

Select the list of othernames of the users. Notice that the output of the SELECT command is compared and any duplicates are removed from the final output.

```
SELECT otherNames FROM Users;

{"otherNames":null}
{"otherNames":null}
{"otherNames":[{"first":"Johny","last":"BeGood"}]}

SELECT DISTINCT otherNames FROM Users;

{"otherNames":null}
{"otherNames":[{"first":"Johny","last":"BeGood"}]}
```

Example 6-19 SELECT Clause

Select the list of firstname and othernames of the users. Notice that the query uses the combination of values in all specified columns in the SELECT list to evaluate the uniqueness.

```
SELECT firstName, otherNames FROM Users;

{"firstName":"Peter","otherNames":null}
{"firstName":"Mary","otherNames":null}
{"firstName":"John","otherNames":[{"first":"Johny","last":"BeGood"}]}

SELECT DISTINCT firstName, otherNames FROM Users;

{"firstName":"Peter","otherNames":null}
{"firstName":"Mary","otherNames":null}
{"firstName":"John","otherNames":[{"first":"Johny","last":"BeGood"}]}
```

LIMIT Clause

The LIMIT clause is used to specify the maximum number M of results to return to the application. M is computed by an expression that may be a single integer literal, or a single external variable, or any expression which is built from literals and external variables and returns a single non-negative integer.

Syntax

```
limit_clause ::= LIMIT add_expression
```

Semantics

Although it's possible to use limit without an order-by clause, it does not make much sense to do so. This is because without an order-by, results are returned in a random order, so the subset of results returned will be different each time the query is run.

Example 6-20 LIMIT Clause

```
SELECT * FROM users
WHERE age > 30
ORDER BY age
LIMIT 5;
```

OFFSET Clause

The OFFSET clause is used to specify a number N of initial query results that should be skipped (not returned to the application). N is computed by an expression that may be a single integer literal, or a single external variable, or any expression which is built from literals and external variables and returns a single non-negative integer.

Syntax

```
offset_clause ::= OFFSET add_expression
```

Semantics

Although it's possible to use offset without an order-by clause, it does not make much sense to do so. This is because without an order-by, results are returned in a random order, so the subset of results skipped will be different each time the query is run.

Example 6-21 OFFSET Clause

```
SELECT * FROM users
WHERE age > 30
ORDER BY age
OFFSET 10;
```

Path Expressions

Syntax

```
path_expression ::= primary_expression (map_step | array_step)*

map_step ::= "." (map_filter_step | map_field_step)

array_step ::= array_filter_step | array_slice_step
```

Semantics

Path expressions are used to navigate inside hierarchically structured data. As shown in the syntax, a path expression has an input expression (which is one of the primary expressions described in the Primary Expressions section, followed by one or more steps. The input expression may return any sequence of items. Each step is actually an expression by itself; It takes as input a sequence of items and produces zero or more items, which serve as the input to the next step, if any. Each step creates a nested scope, which covers just the step itself.

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 via an implicitly-declared variable, whose name is a single dollar sign ($). This context-item variable exists in the scope created by the step expression.

There are several kinds of steps. For all of them, if the context item is NULL, it is just added into the output sequence with no further processing. Otherwise, the following subsections describe the operation performed by each kind of step on each non-NULL context item.

Field Step Expressions

Syntax

```
map_field_step ::=
   id | string | variable_reference |
   parenthesized_expression | function_call
```

Semantics

The main use of a field step is to select 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 have type STRING?.

A field step processes each context item as follows:

- If the context item is an atomic item, it is skipped (the result is empty).
- The name expression is computed. The name expression may reference the context item via the $ variable. If the name expression returns the empty sequence or NULL, the context item is skipped. Otherwise, let K be the result of of the name expression (if an identifier is used instead of a name expression, K is the string with the same characters as the identifier).
- If the context item is a record, then if that record contains a field whose name is equal to K, the value of that field is returned, otherwise, an error is raised.
- If the context item is a map, then if that map contains a field whose name is equal to K, the value of that field is returned, otherwise, an empty result is returned.
- If the content item is an array, the field step is applied recursively to each element of the array (with the context item being set to the current array element).

Example 6-22   Field Step Expression

Select the id and the city of all users.

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

Notice that if the input to a path expressions is a table column, a table alias must be used together with the column name. Otherwise, as explained in the

Variable References section, an expression like address.city would be interpreted as a reference to the city column of a table called address, which is of course not correct.

Recall that address is a column of type JSON. For most (if not all) users, its value will be a json document, i.e.. a map containing other json values. If it is a document and it has a field called city, its value will be returned. For address documents with no city field, the path expression u.address.city returns the empty sequence, which gets converted to NULL by the SELECT clause. The same is true for addresses that are atomic values (e.g. flat strings). Finally, a user may have many addresses stored as an array in the address column. For such a user, all of his/her cities will be returned inside an array.

The record items constructed and returned by the above query will all have type RECORD(id INTEGER, city JSON). The city field of this record type has type JSON, because the address column has type JSON and as a result, any nested field in an address can have any valid JSON value. However, each actual record value in the result will have a city field whose field value has a more specific type (most likely STRING).

Note:

The query processor could be constructing on-the-fly a precise RECORD type for each individual record constructed by the query, but it does not do so for performance reasons. Instead it constructs a common type for all returned record items.

Example 6-23 Field Step Expression

Select the id and amount spent on books for all users who live in California.

```
SELECT id, u.expenses.books
FROM users u
WHERE u.address.state = "CA";
```

In this case, "expenses" is a "typed" map: all of its values have INTEGER as their type. As a result, the record items constructed and returned by the above query will all have type RECORD(id INTEGER, books INTEGER).

Example 6-24 Field Step Expression

For each user, select their id and a field from his/her address. The field to select is specified via an external variable.

```
DECLARE $fieldName STRING;

SELECT u.id, u.address.$fieldName
FROM users u;
```

Example 6-25 Field Step Expression

For each user select all their last names. In this query the otherName column is an array, and the .last step is applied to each element of the array.

```
SELECT lastName, u.otherNames.last
FROM users u;
```

Example 6-26 Field Step Expression

For each user select their id and all of their phone numbers (without the area code). This query will work as expected independently of whether phones is an array of phone objects or a single such phone object. However, if phones is, for example, a single integer or a json object without a number field, the path expression will return the empty sequence, which will be converted to NULL by the SELECT clause.

```
SELECT id, u.address.phones.number
FROM users u;
```

Example 6-27 Field Step Expression

For each state, find how much people in that state spent on books.

```
SELECT u.address.state, sum(u.expenses.books)
FROM users u
GROUP BY u.address.state;
```

For the above query to work, an index must exist whose first field is u.address.state.

Map-Filter Step Expressions

Syntax

```
map_filter_step ::= (KEYS | VALUES) "(" [expression] ")"
```

Semantics

Like field steps, map-filter steps are meant to be used primarily with records and maps. Map-filter steps select either the field names (keys) or the field values of the map/record fields that satisfy a given condition (specified as a predicate expression inside parentheses). If the predicate expression is missing, it is assumed to be the constant true (in which case all the field names or all of the field values will be returned).

A map filter step processes each context item as follows:

- If the context item is an atomic item, it is skipped (the result is empty).
- If the context item is a record or map, the step iterates over its fields. For each field, the predicate expression is computed. In addition to the context-item variable ($), the predicate expression may reference the following two implicitly-declared variables: $key is bound to the name of the context field, i.e., the current field in $, and $value is bound to the value of the context field. The predicate expression must be BOOLEAN?. 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.

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

Example 6-28 Map-Filter Step Expressions

For each user select their id and the expense categories in which the user spent more than $1000.

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

Example 6-29 Map-Filter Step Expressions

For each user select their id and the expense categories in which they spent more than they spent on clothes. In this query, the context-item variable ($) appearing in the filter step expression [$value > $.clothes] refers to the context item of that filter step, i.e., to an expenses map as a whole.

```
SELECT id, u.expenses.keys($value > $.clothes)
FROM users u;
```

Example 6-30 Map-Filter Step Expressions

For each user select their id, the sum of their expenses in all categories except housing, and the maximum of these expenses.

```
SELECT id,
seq_sum(u.expenses.values($key != housing)) AS sum,
seq_max(u.expenses.values($key != housing)) AS max
FROM users u;
```

Example 6-31 Map-Filter Step Expressions

Notice that field steps are actually a special case of map-filter steps. For example the query

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

is equivalent to

```
SELECT id, u.address.values($key = "city")
FROM users u;
```

However, the field step version is the preferred one, for performance reasons.

Array-Filter Step Expressions

Syntax

```
array_filter_step ::= "[" [expression] "]"
```

Semantics

An array filter is similar to a map filter, but it is meant to be used primarily for arrays. An array filter step selects elements of arrays by computing a predicate expression for each element and selecting or rejecting the element depending on the predicate result. The result of the filter step is a sequence containing all selected items. If the predicate expression is missing, it is assumed to be the constant true (in which case all the array elements will be returned).

An array filter step processes each context item as follows:

- 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 as described below.
- If the context item is an array, the step iterates over the array elements and computes the predicate expression on each element. In addition to the context-item variable ($), the predicate expression may reference the following two implicitly-declared variables: $element is bound to the context element, i.e., the current element in $, and $pos is bound to the position of the context element within the array (positions are counted starting with 0). 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 P, the context element is selected only if the condition $pos = P is true. Notice that this implies that if P is negative or greater or equal to the array size, the context element is skipped.

Example 6-32 Array-Filter Step Expression

For each user, select their last name and his/her phone numbers with area code 650.

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

Notice the the path expression in the select clause is enclosed in square brackets, which is the syntax used for arrayconstructor expressions as described in the Array and Map Constructors section. The use of the explicit array constructor guarantees that the records in the result set will always have an array as their second field. Otherwise, the result records would contain an array for users with more than one phones, but a single integer for users with just one phone. Notice also that for users with just one phone, the phones field in address may 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.

Example 6-33 Array-Filter Step Expression

For each user, select their last name and phone numbers having the same area code as the first phone number of that user.

```
SELECT lastName,
[ u.address.phones[$element.area = $[0].area].number ]
FROM users u;
```

Example 6-34 Array-Filter Step Expression

Among the 10 strongest connections of each user, select the ones with id > 100. (Recall that the connections array is assumed to be sorted by the strength of the connections, with the stronger connections appearing first).

```
SELECT [ connections[$element > 100 AND $pos < 10] ]
AS interestingConnections
FROM users;
```

Example 6-35 Array-Filter Step Expression

Count the total number of phones numbers with areacode 650.

```
SELECT count(u.address.phones[$element.area = 650])
FROM users u;
```

Example 6-36 Array-Filter Step Expression

To count the total number of people with at least one phone in the 650 areacode, a case expression (see Case Expressions) and the exists operator (see Exists Operator) must be used.

```
SELECT count(CASE
WHEN EXISTS u.address.phones[$element.area = 650] THEN 1
ELSE 0
END)
FROM users u;
```

Array-Slice Step Expressions

Syntax

```
array_slice_step ::= "[" [expression] ":" [expression] "]"
```

Semantics

Array slice steps are meant to be used primarily with arrays. In general, an array slice step selects elements of arrays based only on the element positions. The elements to select are the ones whose positions are within a range between a "low" position and a "high" position. The low and high positions are computed by two boundary expressions: a "low" expression for the low position and a "high" expression for the high position. 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.

An array filter step processes each context item as follows:

- 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 as described below.
- If the context item is an array, the boundary expressions are computed, if present. If any boundary expression returns NULL or an empty result, the context item is skipped. Otherwise, let L and H be the values returned by the low and high expressions, respectively. If the low expression is absent, L is set to 0. If the high expression is absent, H is set to the size of the array - 1. If L is < 0, L is set to 0. If H > array_size - 1, H is set to array_size - 1. After L and H are computed, the step selects all the elements between positions L and H (L and H included). If L > H no elements are selected.

Notice that based on the above rules, slice steps are actually a special case of filter steps. For example, a slice step with both boundary expressions present, is equivalent to <input expr>[<low expr> <= $pos and $pos <= <high expr>]. Slice steps are provided for convenience (and better performance).

Example 6-37 Array-Slice Step Expression

Select the strongest connection of the user with id 10.

```
SELECT connections[0] AS strongestConnection
FROM users
WHERE id = 10;
```

Example 6-38 Array-Slice Step Expression

For user 10, select his/her 5 strongest connections (i.e. the first 5 ids in the "connections" array).

```
SELECT [ connections[0:4] ] AS strongConnections
FROM users
WHERE id = 10;
```

Notice that the slice expression will return at most 5 ids; if user 10 has fewer that 5 connections, all of his/her connections will be returned.

Example 6-39 Array-Slice Step Expression

For user 10, select his/her 5 weakest connections (i.e. the last 5 ids in the "connections" array).

```
SELECT [ connections[size($) - 5 : ] ] AS weakConnections
FROM users
WHERE id = 10;
```

In this example, size() is a function that returns the size of a given array, and $ is the context array, i.e., the array from which the 5 weakest connections are to be selected.

Comparision Expressions

This section describes various comparision expressions in Oracle NoSQL Database.

Logical Operators: AND, OR, and NOT

Syntax

```
expression ::= or_expression

or_expression ::= and_expression | (or_expression OR and_expression)

and_expression ::= not_expression | (and_expression AND not_expression)

not_expression ::= [NOT] is_null_expression
```

Semantics

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 true if the other operand returns true; otherwise it returns NULL.
- The NOT operator returns NULL.

Example 6-40 Logical Operators

Select the id and the last name for users whose age is between 30 and 40 or their income is greater than 100K.

```
SELECT id, lastName FROM users
WHERE 30 <= age
    AND age <= 40
    OR income > 100000;
```

IS NULL and IS NOT NULL Operators

Syntax

```
is_null_expression ::= condition_expression [IS [NOT] NULL]

condition_expression ::=
   comparison_expression | exists_expression
     | is_of_type_expression | in_expression
```

Semantics

The IS NULL operator tests whether the result of its input expression is NULL. If the input expression returns more than one item, an error is raised. If the result of the input expression is empty, IS NULL returns false. Otherwise, IS NULL returns true if and only if the single item computed by the input expression is NULL. The IS NOT NULL operator is equivalent to NOT (IS NULL cond_expr). NULL is explained in Table 2-2.

Example 6-41 IS NULL Operator

Select the id and last name of all users who do not have a known income.

```
SELECT id, lastName FROM users u
WHERE u.income IS NULL;
```

Value Comparison Operators

Syntax

```
comparison_expression ::= concatenate_expression
   [(value_comparison_operator | any_comparison_operator) add_expression]

value_comparison_operator ::= "=" | "!=" | ">" | ">=" | "<" | "<="
```

Semantics

Value comparison operators are primarily used to compare 2 values, one produced by the left operand and another from the right operand (this is in contrast to the sequence comparisons, defined in the following section which compare two sequences of values). 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 !=.

For the remainder of this section, we assume that each operand returns exactly one item. If an operand returns NULL, the result of the comparison expression is also NULL. Otherwise, the result is a boolean value that is computed as follows.

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 or long value is compared to a float or double value, the integer/long will first be cast to float/double. If one of the operands is a number value, the other operand will first be cast to number (if not a number already).
- A string item is comparable to another string item (using the java String.compareTo() method). 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 below; 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). As mentioned above, an enum item is also comparable to a string item, by casting the string to an enum item.
- 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, using the java Boolean.compareTo() method.
- 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 are defined in a recursive fashion. Specifically:

- A record is comparable with another record for equality only and only if they contain comparable values. More specifically, 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. Remember that json documents are modelled as maps, so 2 json documents can be compared for equality. More specifically, 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. Furthermore, comparisons between atomic and complex items return false always.

The reason for returning false for incomparable items, instead of raising an error, is to handle truly 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.

Example 6-42   Value Comparison Operator

We have already seen examples of comparisons among atomic items. Here is an example involving a comparison between two arrays.

Select the id and lastName for users who are connected with users 3, 20, and 10 only and in exactly this order. In this example, an array constructor (see Array and Map Constructors) is used to create an array with the values 3, 20, and 10, in this order.

```
SELECT id, lastName FROM users
WHERE connections = [3, 20, 10];
```

Sequence Comparison Operators

Syntax

```
any_comparison_operator ::= "=any" | "!=any" | ">any" | ">=any" | "<any"
| "<=any"
```

Semantics

Comparisons between two sequences is done via another set of operators: =any, !=any, >any, >=any, <any, <=any. These any operators have existential semantics: the result of an any operator on two input sequences S1 and S2 is true if and only if there is a pair of items i1 and i2, where i1 belongs to S1, i2 belongs to S2, and i1 and i2 compare true via the corresponding value comparison operator. Otherwise, if any of the input sequences contains NULL, the result is NULL. Otherwise, the result is false.

Example 6-43   Sequence Comparison Operator

Select the id, lastName and address for users who are connected with the user with id 3. Notice the use of [] after connections: it is an array filter step (see Array-Filter Step Expressions), which returns all the elements of the connections array as a sequence (it is unnesting the array).

```
SELECT id, lastName, address FROM users
WHERE connections[] =any 3;
```

Example 6-44   Sequence Comparison Operator

Select the id and lastName for users who are connected with any users having id greater than 100.

```
SELECT id, lastName FROM users
WHERE connections[] >any 100;
```

Example 6-45   Sequence Comparison Operator

Select the id of each user who is connected with a user having id greater than 10 and is also connected with a user having id less than 100.

```
SELECT id FROM users u
WHERE 10 <any u.connections[]
    AND u.connections[] <any 100;
```

Notice that the above query is not the same as the query: "select the id of each user who is connected with a user having an id in the range between 10 and 100". In the first query, we are looking for some connection with id greater than 10 and another connection (which may or may not be the same as the 1st one) with id less than 100. In the second query we are looking for some connection whose id is between 10 and 100. To make the difference clear, consider a Users table with only 2 users (say with ids 200 and 500) having the following connections arrays respectively: [1, 3, 110, 120] and [1, 50, 130]. Both of these arrays satisfy the predicates in the first query, and as a result, both users will be selected. On the other hand, the second query will not select user 200, because the array [1, 3, 110, 120] does not contain any element in the range 10 to 100.

The second query can be written by a combination of an EXISTS operator and an array filtering step:

```
SELECT id FROM users u
WHERE EXISTS u.connections
    [10 < $element AND $element < 100];
```

and the first query, with the 2 <any operators, is equivalent to the following one:

```
SELECT id FROM users u
WHERE EXISTS u.connections[10 < $element]
    AND EXISTS u.connections[$element < 100];
```

Example 6-46 Sequence Comparison Operator

Select the first and last name of all users who have a phone number with area code 650. Notice that although we could have used [] after phones in this query, it is not necessary to do so, because the phones array (if it is indeed an array) is unnested implicitly by the .area step that follows.

```
SELECT firstName, lastName FROM users u
WHERE u.address.phones.area =any 650;
```

IN Operator

Syntax

```
in_expression ::= in1_expression | in2_expression |
    in3_expression | in4_expression

in1_expression ::= "(" concatenate_expression
    ("," concatenate_expression)* ")"
    IN "(" expression ("," expression)* ")"
in2_expression ::= concatenate_expression
    IN "(" expression ("," expression)* ")"
in3_expression ::= concatenate_expression IN path_expression
in4_expression ::= "(" concatenate_expression
    ("," concatenate_expression)* ")" IN path_expression
```

Semantics

The IN operator is essentially a compact alternative to a number of OR-ed equality conditions. For example, the query

```
SELECT * FROM users WHERE age IN (22, 25, 43)

is equivalent to

SELECT * FROM users WHERE age = 22 OR age = 25 OR age = 43
```

and the query,

```
SELECT * FROM users
    WHERE (firstName, lastName) IN
        (("John","Smith"),("Peter","Paul"),("Mary","Ann"))

is equivalent to

SELECT * FROM users
    WHERE (firstName = "John" AND lastName = "Smith") OR
        (firstName = "Peter" AND lastName = "Paul") OR
        (firstName = "Mary" AND lastName = "Ann")
```

As shown in the grammar, there are 4 syntactic variants of the IN operator. The in1_expression and in2_expression follow the standard SQL syntax. The in2_expression one is actually a special case of the in1_expression, for the case when there is only one expression in the left-hand-side of the operator. For the in1_expression, if K is the number of expressions in the left-hand-side, then each expression list in the right-hand-side must consist of K expressions. If N is the number of expression lists in the right-hand-side, then the whole IN condition is equivalent to:

```
(expr1 = expr11 and expr2 = expr12 and exprK = expr1K) or
(expr1 = expr21 and expr2 = expr22 and exprK = expr2K) or
.... or
(expr1 = exprN1 and expr2 = exprN2 and exprK = exprNK)
```

However, in addition to being more compact, queries using IN operators will be executed more efficiently if appropriate indexes exist. For example, if table users has an index on columns age, firstName and lastName, then both of the above IN queries will use that index to find the qualifying rows, whereas the equivalent OR queries will be executed via full table scans. See also examples in Finding Applicable Indexes.

The in3_expression and in4_expression variants allow a relative large number of search keys to be provided via a single bind variable. For example, if the $keys variable in bound to the array ["John", "Smith", "Peter", "Paul", "Mary", "Ann"], then the following query is equivalent to the second IN query above.

```
DECLARE $keys ARRAY(json);
SELECT * FROM users
    WHERE (firstName, lastName) IN $keys[]
```

In general, with the in3_expression and in4_expression variants, the expression in the right-hand-side is evaluated first. If the number M of items in the resulting sequence is less than the number K of expressions in the left-hand-side, the result of the IN operator is false. If M is not a multiple of K, the last (M mod K) items in the sequence are discarded and M is set to the number of remaining items. Then, the IN expression is equivalent to:

```
(expr1 = k1 and expr2 = k2 and exprK = kK) or
(expr1 = kK+1 and expr2 = kK+2 and exprK = k2*K) or
 .... or
(expr1 = kM-K and expr2 = kM-K+1 and exprK = kM)
```

However, an additional type-checking restriction applies in this case: in each of the above equality conditions, the type of the right-hand-side item must be a subtype of the left-hand-side type.

Regular Expression Conditions

The regex_like function performs regular expression matching. A regular expression is a pattern that the regular expression engine attempts to match with an input string. The syntax for invoking the regex_like function in a query is the same as all other functions, described in the Function Calls section. The regex_like function has 2 signatures with 2 and 3 parameters, respectively.

Syntax

```
boolean regex_like(any*, string)

boolean regex_like(any*, string, string)
```

Semantics

The regex_like function provides functionality similar to the LIKE operator in standard SQL, that is, it can be used to check if an input string matches a given pattern. The input string and the pattern are computed by the first and second arguments, respectively. A third, optional, argument specifies a set of flags that affect how the matching is done.

Normally, the regex_like function expects each of its arguments to return a single string. If that is not the case, it behaves as follows:

- If it can be detected at compile time that the first argument will never return a string, it raises a compile-time error. Otherwise, it returns false if the first argument returns nothing, or more than one items, or a single item that is neither a string nor NULL.
- It raises an error if the pattern or flags do not return a single string or NULL.
- It returns NULL if any of the arguments returns a single NULL.

Otherwise, the regex_like function behaves as follows:

- Raises an error if the pattern string is not valid or its length is greater than 512 characters.
- Returns false if pattern does not match the input string.

- Returns true if pattern matches input string.

The pattern string is the regular expression against which the input text is matched. The syntax of the pattern string is a subset of the one supported by the java Pattern class, see https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html. Specifically, each character in a regular expression is either a literal character that matches itself, or a meta character, that specifies a "construct" having a special meaning. Only the following constructs are supported: quoted characters, the quotation constructs, the period (.), and the greedy quantifier (*).

The period (.) is a meta-character that matches every character expect a new line. The greedy quantifier (*) is a meta-character that indicates zero or more occurrences of the preceding element. For example, the regex "D.*" matches any string that starts with the character 'D' and followed by zero or more characters.

For the full list of supported predefined quoted characters see https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html.

The flags string contains one or more characters, where each character is a flag specifying some particular behavior. The full list of acceptable characters and their semantics is listed in the following table:

Table 6-2 Predefined Quoted Characters

Flag	Full Name	Description
"d"	UNIX_LINES	Enables Unix lines mode. In this mode, only the '\n' line terminator is recognized in the behavior of period (.).
"i"	CASE_INSENSITIVE	Enables case-insensitive matching. By default, CASE_INSENSITIVE matching assumes that only characters in the US-ASCII character set are being matched. You can enable Unicode-aware CASE_INSENSITIVE by specifying the UNICODE_CASE flag in conjunction with this flag. Specifying this flag may impose a slight performance penalty.
"x"	COMMENTS	Permits white space and comments in pattern. In this mode, white space is ignored, and embedded comments starting with # are ignored until the end of a line.
"l"	LITERAL	When LITERAL is specified then the input string that specifies the pattern is treated as a sequence of literal characters. There is no special meaning for Metacharacters or escape sequences. The flags CASE_INSENSITIVE and UNICODE_CASE retain their impact on matching when used in conjunction with this flag. The other flags become superfluous.
"s"	DOTALL	Enables DOTALL mode. In DOTALL mode, the expression dot (.) matches any character, including a line terminator. However, by default, the expression dot (.) does not match line terminators.
"u"	UNICODE_CASE	When you enable the CASE_INSENSITIVE flag, by default, it does matching using only the characters in the US-ASCII character set. When you specify UNICODE_CASE then it does CASE_INSENSITIVE matching using the Unicode standard. Specifying this flag may impose a performance penalty.

Table 6-2 (Cont.) Predefined Quoted Characters

Flag	Full Name	Description
"c"	CANON_EQ	When this flag is specified then two characters will be considered to match if, and only if, their full canonical decompositions match. When you specify this flag, the expression "a\u030A", for example, will match the string "\u00E5. By default, matching does not take canonical equivalence into account. Specifying this flag may impose a performance penalty.
"U"	UNICODE_CHARACTER_CLASS	Enables the Unicode version of Predefined character classes and POSIX character classes. When you specify this flag, then the (US-ASCII only) Predefined character classes and POSIX character classes are in conformance with Unicode Technical Standards. See http://unicode.org/reports/tr18/#Compatibility_Properties. The flag implies UNICODE_CASE; it enables Unicode-aware case folding.Specifying this flag may impose a performance penalty.

Note:

The regex_like function will not be used for index scan optimization.

Example 6-47 Regular Expressions

To select all the people whose last name starts with 'S', use the pattern shown in the following query:

```
SELECT id, lastName FROM users where regex_like(lastName, "S.*");

 +----+----------+
 | id | lastName |
 +----+----------+
 | 10 | Smith    |
 +----+----------+

1 row returned
```

Example 6-48 Regular Expressions

To select all the people whose last name has at least one 'a' or 'A' in it, use the pattern and the flag for case insensitive shown in the following query:

```
SELECT id, lastName from users where regex_like(lastname,".*a.*","i");

 +----+----------+
 | id | lastName |
 +----+----------+
 | 20 | Ann      |
 | 30 | Paul     |
 +----+----------+

2 rows returned
```

Exists Operator

Syntax

```
exists_expression ::= EXISTS concatenate_expression
```

Semantics

The exists operator checks whether the sequence returned by its input expression is empty or not, and returns false or true, respectively. A special case is when the input expression returns NULL. In this case, EXISTS will also return NULL, unless it is known that the input expression will always return at least one item, in which case EXISTS returns true.

Example 6-49   Exists Operator

Find all the users who do not have a zip code in their addresses.

```
SELECT id FROM users u
WHERE NOT EXISTS u.address.zip;
```

Notice that the above query does not select users whose zip code has the json null value. The following query includes those users as well.

```
SELECT id FROM users u
WHERE NOT EXISTS u.address.zip OR u.address.zip = null;
```

What if the Users table contains a row R whose address column is NULL? In general, SQL for Oracle NoSQL Database interprets NULL as an unknown value, rather than an absent value. So, in row R, the address is unknown, and as a result, we don't know what its zip code is or if it even has a zip code. In this case, the expression u.address.zip will return NULL on R and exists u.address.zip will also return NULL, which implies that row R will not be selected by the above queries. On the other hand, row R will be selected by the following query. In this case, we know that every row does have an address, even though the address may be unknown (i.e., NULL) in some rows. So, even though the expression u.address returns NULL, exists u.address return true.

```
SELECT id FROM users u
WHERE EXISTS u.address;
```

Is-Of-Type Operator

Syntax

```
is_of_type_expression ::=
   add_expression IS [NOT] OF [TYPE]
   "(" [ONLY] sequence_type ([ONLY] sequence_type)* ")"
```

Semantics

The is-of-type operator checks the sequence type of its input sequence against one or more target sequence types. If the number N of the target types is greater than one, the expression is equivalent to OR-ing N is-of-type expressions, each having one target type. So, for the remainder of this section, we will assume that only one target type is specified.

The is-type-of operator will return true if both of the following conditions are true:

1. the cardinality of the input sequence matches the quantifier of the target type. Specifically,
   a. if the quantifier is * the sequence may have any number of items,
   b. if the quantifier is + the input sequence must have at least one item,
   c. if the quantifier is ? The input sequence must have at most one item, and
   d. if there is no quantifier, the input sequence must have exactly one item.
2. all the items in the input sequence are instances of the target item-type (type_def), i.e. the type of each input item must be a subtype of the target item-type. For the purposes of this check, a NULL is not considered to be an instance of any type.

If condition (1) is satisfied and the input sequence contains a NULL, the result of the is-type-of operator will be NULL. In all other cases, the result is false.

Example 6-50 Is-Of-Type Operator

Find all the users whose address information has been stored as a single, flat string.

```
SELECT id
FROM users u
WHERE u.address IS OF TYPE (STRING);
```

Concatenation Operator

Syntax

```
concatenation_operator ::= "||"

concatenate_expression ::= add_expression ("||" add_expression)*
```

Semantics

The concatenation operator returns the character string made by joining the operands in the given order. The operands can be of `any*` type. For more details, see the concat Function section.

> **Note:**
>
> According to the operator precedence, the || operator is immediately after +, - (as binary operators).

Example 6-51 Concatenation Operator

This example joins id, firstname, and lastname into a single string and provides the output. Notice that id, which is an integer type, also gets concatenated with the string values.

```
SELECT id || firstname || lastname AS name FROM users;


 +-------------+
 |    name     |
 +-------------+
 | 10JohnSmith |
 | 30PeterPaul |
 | 20MaryAnn   |
 +-------------+
```

Arithmetic Expressions

Syntax

```
add_expression ::= multiply_expression (("+"|"-") multiply_expression)*

multiply_expression ::= unary_expression (("*"|"/"|"div")
unary_expression)*

unary_expression ::= path_expression | (("+"|"-") unary_expression)
```

Semantics

Oracle NoSQL Database supports the following arithmetic operations: +, -, *, / and div. Each operand to these operators must produce at most one numeric item. If any operand returns the empty sequence or NULL, the result of the arithmetic operation is also empty or NULL, respectively. Otherwise, the operator returns a single numeric item, which is computed as follows:

- If any operand returns a Number item, the item returned by the other operand is cast to a Number value (if not a Number already) and the result is a Number item that is computed using java's arithmetic on BigDecimal, otherwise,

- If any operand returns a double item, the item returned by the other operand is cast to a double value (if not a double already) and the result is a double item that is computed using java's arithmetic on doubles, otherwise,
- If any operand returns a float item, the item returned by the other operand is cast to a float value If not a float already) and the result is a float item that is computed using java's arithmetic on floats, otherwise,
- Except for the div operator, if any operand returns a long item, the item returned by the other operand is cast to a long value (if not a long already) and the result is a long item that is computed using java's arithmetic on longs.
- Except for the div operator, if all operands return integer items, the result is an integer item that is computed using java's arithmetic on ints.
- The div operator performs floating-point division, even if both its operands are longs and/or integers. In this case, div returns a double.

Oracle NoSQL Database supports the unary + and – operators as well. The unary + is a no-op, and the unary – changes the sign of its numeric argument.

Example 6-52 Arithmetic Expression

For each user show their id and the difference between their actual income and an income that is computed as a base income plus an age-proportional amount.

```
DECLARE
$baseIncome INTEGER;
$ageMultiplier DOUBLE;
SELECT id,
income - ($baseIncome + age * $ageMultiplier) AS adjustment
FROM Users;
```

Primary Expressions

Syntax

```
primary_expression ::=
    parenthesized_expression |
    constant_expression |
    column_reference |
    variable_reference |
    array_constructor |
    map_constructor |
    case_expression |
    cast_expression |
    extract_expression |
    function_call |
    count_star |
    transform_expression
```

The following subsections describe each of the primary expressions listed in the above grammar rule, except from count_star, which is the count(*) aggregate function defined in the Using Aggregate Functions section.

Parenthesized Expressions

Syntax

```
parenthesized_expression ::= "(" expression ")"
```

Semantics

Parenthesized expressions are used primarily to alter the default precedence among operators. They are also used as a syntactic aid to mix expressions in ways that would otherwise cause syntactic ambiguities. An example of the later usage is in the definition of the field_step parse rule in the Field Step Expressions section.

Example 6-53 Parenthesized Expression

Select the id and the last name for users whose age is less or equal to 30 and either their age is greater than 20 or their income is greater than 100K.

```
SELECT id, lastName
FROM users
WHERE (income > 100000 OR 20 < age) AND age <= 30;
```

Constant Expressions

Syntax

```
constant_expression ::= number | string | TRUE | FALSE | NULL

number ::= [MINUS] (FLOAT_CONSTANT | INT_CONSTANT | NUMBER_CONSTANT)
string ::= STRING_CONSTANT | DSTRING_CONSTANT
```

Semantics

The syntax for INT_CONSTANT, FLOAT_CONSTANT, NUMBER_CONSTANT, STRING_CONSTANT, and DSTRING_CONSTANT was given in the Identifiers section.

In the current version, a query can contain the following constants (a.k.a. literals):

String
String literals are sequences of unicode characters enclosed in double or single quotes. String literals are translated into String items. Notice that any escape sequences appearing in a string literal will be converted to their corresponding character inside the corresponding String item.

Integer
Integer literals are sequences of one or more digits. Integer literals are translated into Integer items, if their value fits in 4 bytes, into Long items, if they fit in 8 bytes, otherwise to Number items.

Floating point
Floating point literals represent real numbers using decimal notation and/or exponent. Floating-point literals are translated into Double items, if possible, otherwise to Number items.

Number
Number literals are integer or floating-point literals followed by the 'n' or 'N' character. Number literals are always translated into Number items.

TRUE / FALSE
The TRUE and FALSE literals are translated to the boolean true and false items, respectively.

NULL
The NULL literal is translated to the json null item.

Column References

Syntax

```
column_reference ::= id ["." id]
```

Semantics

A column-reference expression returns the item stored in the specified column within the context row (the row that an SELECT expression is currently working on). Syntactically, a column-reference expression consists of one identifier, or 2 identifiers separated by a dot. If there are 2 ids, the first is considered to be a table alias and the second the name of a column in that table. We call this form a qualified column name. A single id is resolved to the name of a column in some of the tables referenced inside the FROM clause. However, in this case there must not be more than one tables that participate in the query and have a column with this name. We call this form an unqualified column name.

Variable References

Syntax

```
variable_reference ::= "$" [id]
```

Semantics

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.

Array and Map Constructors

Syntax

```
array_constructor ::=
   "[" expression ("," expression)* "]"
```

```
map_constructor ::=
   ("{" expression ":" expression
   ("," expression ":" expression)* "}") |
   ("{" "}")
```

Semantics

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).

Similarly, 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 and the second returns the associated field value. If a value expression returns more than one items, an array is implicitly constructed to store the items, and that array becomes the field value. If either a field name or a 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 arrays or maps 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 or map may be inserted in another constructed array or map and this "parent" array/map has type ARRAY(JSON) or MAP(JSON), then the "child" array/map will also have type ARRAY(JSON) or MAP(JSON). This is to enforce the restriction that "typed" data are not allowed inside JSON data (see Data Type Hierarchy).

Example 6-54   Array and Map Constructor

For each user create a map with 3 fields recording the user's last name, their phone information, and the expense categories in which more than $5000 was spent.

```
SELECT
{
    "last_name" : u.lastName,
    "phones" : u.address.phones,
    "high_expenses" : [ u.expenses.keys($value > 5000) ]
}
FROM users u;
```

Notice that the use of an explicit array for the "high_expenses" field guarantees that the field will exist in all the constructed maps, even if the path inside the array constructor returns empty. Notice also that although it is known at compile time that all elements of the constructed arrays will be strings, the arrays are constructed with type ARRAY(JSON) (instead of ARRAY(STRING)), because they are inserted into a JSON map.

Case Expressions

Syntax

```
case_expression ::= CASE
   WHEN expression THEN expression
   (WHEN expression THEN expression)*
   [ELSE expression]
   END
```

Semantics

The searched CASE expression is similar to the if-then-else statements of traditional programming languages. It consists of a number of WHEN-THEN pairs, followed by an optional ELSE clause at the end. Each WHEN expression is a condition, i.e., it must return BOOLEAN?. The THEN expressions as well as the ELSE expression may return any sequence of items. The CASE expression is evaluated by first evaluating the WHEN expressions from top to bottom until the first one that returns true. If it is the i-th WHEN expression that returns true, then the i-th THEN expression is evaluated and its result is the result of the whole CASE expression. If no WHEN expression returns true, then if there is an ELSE, its expression is evaluated and its result is the result of the whole CASE expression; Otherwise, the result of the CASE expression is the empty sequence.

Example 6-55 Case Expressions

For each user create a map with 3 fields recording the user's last name, their phone information, and the expense categories in which more than $5000 was spent.

```
SELECT
{
    "last_name" : u.lastName,
    "phones" : CASE
        WHEN exists u.address.phones
        THEN u.address.phones
        ELSE "Phone info absent or not at the expected place"
    END,
    "high_expenses" : [ u.expenses.keys($value > 5000) ]
}
FROM users u;
```

The query is very similar to the one from array and map constructor. The only difference is in the use of a case expression to compute the value of the phones field. This guarantees that the phones field will always be present, even if the path expression u.address.phones return empty or NULL. Notice that wrapping the path expression with an explicit array constructor (as we did for the high_expenses field) would not be a good solution here, because in most cases u.address.phones will return an array, and we don't want to have construct an extra array containing just another array.

Cast Expression

Syntax

```
cast_expression ::= CAST "(" expression AS sequence_type ")"
```

Semantics

The cast expression creates, if possible, new items of a given target type from the items of its input sequence. Specifically, a cast expression is evaluated as follows:

A cardinality check is performed first:

1. if the quantifier of the target type is * the sequence may have any number of items,
2. if the quantifier is + the input sequence must have at least one item,
3. if the quantifier is ? the input sequence must have at most one item, and
4. if there is no quantifier, the input sequence must have exactly one item.

If the cardinality of the input sequence does not match the quantifier of the target type, an error is raised. Then, each input item is cast to the target item type according to the following (recursive) rules.

- If the type of the input item is equal to the target item type, the cast is a no-op: the input item itself is returned.
- If the target type is a wildcard type other than JSON and the type of the input item is a subtype of the wildcard type, the cast is a no-op; Otherwise an error is raised.
- If the target type is JSON, then (a) an error is raised if the input item is has a non-json atomic type, else (b) if the input item has a type that is a json atomic type or ARRAY(JSON) or MAP(JSON), the cast is a no-op , else (c) if the input item is a non-json array, a new array of type ARRAY(JSON) is constructed, each element of the input array is cast to JSON, and the resulting item is appended into the new json array, else (d) if the input item is a non-json map, a new map of type MAP(JSON) is constructed, each field value of the input map is cast to JSON, and resulting item together with the associated field name are inserted into the new json map, else (e) if the input item is a record, it is cast to a map of type MAP(JSON) as described below.
- If the target type is an array type, an error is raised if the input item is not an array. Otherwise, a new array is created, whose type is the target type, each element in the input array is cast to the element type of the target array, and the resulting item is appended into the new array.
- If the target type is a map type, an error is raised if the input item is not a map or a record. Otherwise, a new map is created, whose type is the target type, each field value in the input map/record is cast to the value type of the target map, and the resulting field value together with the associated field name are inserted to the new map.
- If the target type is a record type, an error is raised if the input item is not a record or a map. Otherwise, a new record is created, whose type is the target type. If the input item is a record, its type must have the same fields and in the same order as the target type. In this case, each field value in the input record is cast to the value type of the corresponding field in the target type and the resulting field value together with the associated field name are added to the new record. If the input

item is a map, then for each map field, if the field name exists in the target type, the associated field value is cast to the value type of the corresponding field in the target type and the resulting field value together with the associated field name are added to the new record. Any fields in the new record whose names do not appear in the input map have their associated field values set to their default values.

- If the target type is string, the input item may be of any type. In other words, every item can be cast to a string. For complex items their “string value” is a json-text representation of their value. For timestamps, their string value is in UTC and has the format "uuuu-MM-dd['T'HH:mm:ss]". For binary items, their string value is a base64 encoding of their bytes.
- If the target type is an atomic type other than string, the input item must also be atomic. Among atomic items and types the following casts are allowed:
  - Every numeric item can be cast to every other numeric type. The cast is done as in Java.
  - Integers and longs can be cast to timestamps. The input value is interpreted as the number of milliseconds since January 1, 1970, 00:00:00 GMT.
  - String items may be castable to all other atomic types. Whether the cast succeeds or not depends on whether the actual string value can be parsed into a value that belongs to the domain of the target type.
  - Timestamp items are castable 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.

Example 6-56 Cast Expression

Select the last name of users who moved to their current address in 2015 or later.

```
SELECT u.lastName FROM Users u
WHERE
CAST (u.address.startDate AS Timestamp(0)) >=
CAST ("2015-01-01T00:00:00" AS Timestamp(0));
```

Since there is no literal for Timestamp values, to create such a value a string has to cast to a Timestamp type.

Extract Expressions

Syntax

```
extract_expression ::= EXTRACT "(" id FROM expression ")"
```

Semantics

The extract expression extract a component from a timestamp. Specifically, the expression after the FROM keyword must return at most one timestamp or NULL. If the result of this expression is NULL or empty, the results of EXTRACT is also NULL or empty, respectively. Otherwise, the component specified by the id is returned. This id must be one of the following keywords:

YEAR
Returns the year for the timestamp, in the range -6383 ~ 9999.

MONTH
Returns the month for the timestamp, in the range 1 ~ 12.

DAY
Returns the day of month for the timestamp, in the range 1 ~ 31.

HOUR
Returns the hour of day for the timestamp, in the range 0 ~ 23.

MINUTE
Returns the minute for the timestamp, in the range 0 ~ 59.

SECOND
Returns the second for the timestamp, in the range 0 ~ 59.

MILLISECOND
Returns the fractional second in millisecond for the timestamp, in the range 0 ~ 999.

MICROSECOND
Returns the fractional second in microsecond for the timestamp, in the range 0 ~ 999999.

NANOSECOND
Returns the fractional second in nanosecond for the timestamp, in the range 0 ~ 999999999.

WEEK
Returns the week number within the year where a week starts on Sunday and the first week has a minimum of 1 day in this year, in the range 1 ~ 54.

ISOWEEK
Returns the week number within the year based on IS0-8601, where a week starts on Monday and the first week has a minimum of 4 days in this year, in range 0 ~ 53.

There are specific built-in functions to extract each of the above components from a time stamp. For example, EXTRACT(YEAR from expr) is equivalent to year(expr). These and other built-in functions are described in Built-in Functions.

Function Calls

Syntax

```
function_call ::= id "(" [expression ("," expression)*] ")"
```

Semantics

Function-call expressions are used to invoke functions, which in the current version can be built-in (system) functions only. Syntactically, a function call starts with an id which identifies the function to call by name, followed by a parenthesized list of zero or more argument expressions separated by a comma.

Each function has a signature, which specifies the sequence type of its result and a sequence type for each of its parameters. Evaluation of a function-call expression starts with the evaluation of each of its arguments. The result of each argument expression must be a subtype of the corresponding parameter type, or otherwise, it must be promotable to the parameter type. In the latter case, the argument value will actually be cast to the expected type. Finally, after type checking and any necessary promotions are done, the function's implementation is invoked with the possibly promoted argument values.

The following type promotions are currently supported:

- INTEGER is promotable to FLOAT or DOUBLE.
- LONG is promotable to FLOAT or DOUBLE.
- STRING is promotable to ENUM, but the cast will succeed only if the ENUM type contains a token whose string value is the same as the input string.

The list of currently available functions is given in the Built-in Functions chapter.

Sequence Transform Expressions

Syntax

```
transform_expression ::= SEQ_TRANSFORM "(" expression "," expression ")"
```

Semantics

A sequence transform expression transforms a sequence to another sequence. Syntactically it looks like a function whose name is seq_transform. The first argument is an expression that generates the sequence to be transformed (the input sequence) and the second argument is a "mapper" expression that is computed for each item of the input sequence. The result of the seq_transform expression is the concatenation of sequences produced by each evaluation of the mapper expression. The mapper expression can access the current input item via the $ variable.

Example 6-57 Sequence Transform Expression

As an example, assume a "sales" table with the following data.

```
CREATE TABLE sales (
    id INTEGER,
    sale RECORD (
        acctno INTEGER,
        year INTEGER,
        month INTEGER,
        day INTEGER,
        state STRING,
        city STRING,
        storeid INTEGER,
        prodcat STRING,
        items ARRAY(
            RECORD (
                prod STRING,
                qty INTEGER,
                price INTEGER
            )
        )
    ),
    PRIMARY KEY (id)
);


INSERT INTO sales VALUES (
    1,
    {
        "acctno" : 349,
        "year" : 2000,
        "month" : 10,
        "day" : 23,
        "state" : "CA",
        "city" : "San Jose",
        "storeid" : 76,
        "prodcat" : "vegies",
        "items" :[
            { "prod" : "tomatoes", "qty" : 3, "price" : 10.0 },
            { "prod" : "carrots", "qty" : 1, "price" : 5.0 },
            { "prod" : "pepers", "qty" : 1, "price" : 15.0 }
        ]
    }
);
```

Assume there is the following index on sales:

```
CREATE INDEX idv1 ON sales (
    sale.acctno, sale.year, sale.prodcat);
```

Then we can write the following query, which returns the total sales per account number and year:

```
SELECT t.sale.acctno,
t.sale.year,
SUM(seq_transform(t.sale.items[], $price * $qty)) AS sales
FROM sales t
GROUP BY t.sale.acctno, t.sale.year;
```

Joins

Oracle NoSQL Database does not currently support the general join operators found in more traditional relational database systems. However, it does support a special kind of join among tables that belong to the same table hierarchy. These joins can be executed efficiently, because only co-located rows may match with each other. As a result, transferring very large amounts of data among servers is avoided.

Joining Tables in the Same Table Hierarchy

To query multiple tables in the same hierarchy, the NESTED TABLES clause must be used inside the FROM clause.

Syntax

```
nested_tables ::=
   NESTED TABLES "(" single_from_table
   [ANCESTORS "(" ancestor_tables ")"]
   [DESCENDANTS "(" descendant_tables ")"]
")"

ancestor_tables ::= nested_from_table ("," nested_from_table)*

descendant_tables ::= nested_from_table ("," nested_from_table)*

nested_from_table ::= aliased_table_name [ON or_expression]
```

The NESTED TABLES clause specifies the participating tables and separates them in 3 groups. First the target table is specified. Then the ANCESTORS clause, if present, specifies a number of tables that must be ancestors of the target table in the table hierarchy. Finally, the DESCENDANTS clause, if present, specifies a number of tables that must be descendants of the target table in the table hierarchy. For each table an alias may be specified (and if not, one is created internally as described in the CREATE TABLE Statement section). The aliases must be unique.

Semantically, a NESTED TABLES clause is equivalent to a number of left-outer-join operations "centered" around the target table. The left-outer-join is an operation defined by standard SQL and supported by all major RDBMSs. For those not familiar with it already, we give a brief description in the last section of this chapter.

Our implementation of left outer join diverges slightly from the standard definition. The difference is in the "shape" of the results. Specifically, the result of a NESTED TABLES clause is a set of records, all having the same type, where (a) the number of fields is equal to the number of participating tables, (b) each field corresponds to one of the tables and stores either a row from that table or NULL, (c) the name of each field is the alias used by the associated table, and (d) the fields are ordered in the order that the participating tables would be encountered in a depth-first traversal of the table hierarchy.

So, in a NESTED TABLES result the columns of each table are grouped inside a subrecord. In contrast, the standard left-outer-join produces a "flat" result, where each result is a record/tuple whose number of fields is the sum of all the columns in the participating tables.

The mapping of a NESTED TABLES to a number of left-outer-joins is best explained with a few examples. For brevity, we will use the keyword LOJ in place of LEFT OUTER JOIN. Let’s start with the following create table statements:

```
create table A (ida integer, a1 string, primary key(ida));
create table A.B (idb integer, b1 string, primary key(idb));
create table A.B.C (idc integer, c1 integer, primary key(idc));
create table A.B.C.D (idd integer, d1 double, primary key(idd));
create table A.B.E (ide integer, e1 integer, primary key(ide));
create table A.G (idg integer, g1 string, primary key(idg));
create table A.G.J (idj integer, j1 integer, primary key(idj));
create table A.G.H (idh integer, h1 integer, primary key(idh));
create table A.G.J.K (idk integer, k1 integer, primary key(idk));
```

The above statements create the following table hierarchy:

The NESTED TABLES clause specifies the join tree as a "projection" of the table hierarchy that includes the tables in the NESTED TABLES. For example, the join tree for NESTED TABLES(A.B) ancestors(A) descendants (A.B.C.D, A.B.E) is shown below. The arrows indicate the direction of the LOJs (from the left table to the right table).

Now, let's look at the following NESTED TABLES cases and their equivalent LOJ operations

1.
   ```
   NESTED TABLES (A.B.C c ancestors(A a, A.B b));
   ```

   is equivalent to

   ```
   (A.B.C c LOJ A.B b ON c.ida = b.ida and c.idb = b.idb) LOJ A a
   ON c.ida = a.ida
   ```

   We can see that the join predicates are implicit in the NESTED TABLES clause, and they are always on the primary key columns of the participating tables.

   Because for each A.B.C row there is at most one matching A and A.B row, the number of records in the result is the same as the number of A.B.C rows. This is always true when the NESTED TABLES clause includes ancestors only. In this case, the effect of the operation is to *decorate* the target table rows with the columns from the matching ancestor rows (if any), without eliminating or adding any other rows.

2.
   ```
   NESTED TABLES (A a descendants(A.B b, A.B.C c))
   ```

   is equivalent to

   ```
   A a LOJ
   (A.B b LOJ A.B.C c ON b.ida = c.ida and b.idb = c.idb)
   ON a.ida = b.ida
   ```

   Another way to explain the semantics of the DESCENDANTS clause is to use the contains relationship defined in Table Hierarchies section, but restricted to the descendant tables in the join tree only. Let R be a target table row, and S(R) be the set containing all the descendant rows that are reachable from R via the contains relationship (i.e., S(R) is the transitive closure of contains applied on R). If S(R) is empty, a single record is returned for R, that contains R and a NULL value for each of the descendant tables in the join tree. Otherwise, let B(R) be the boundary subset of S(R), i.e., all rows in S(R) that do not have any descendant rows in the join tree. Then, a result is generated for each row in B(R) as follows: Let RR be a row in B(R) and T be its containing table. The result associated with RR is a record containing all the rows in the path from R to RR and a NULL value for every table that is not in the path from the target table to T.

3.
   ```
   NESTED TABLES (A a descendants(A.B b, A.G g))
   ```

is equivalent to

```
A a LOJ
(A.B b UNION A.G g)
ON (a.ida = b.ida or b.ida IS NULL) and (a.ida = g.ida or g.ida IS
NULL)
```

As in case 2, target table A is joined with 2 descendant tables. However, because the descendant tables come from 2 different branches of the join tree, we have to use a UNION operation in the SQL expression above. This UNION unions both the rows and the columns of tables A.B and A.G. So, if table A.B has N rows with n columns each and table A.G has M rows with m columns, the result of the UNION has N + M rows, with n + m columns each. The first N rows contain the rows of A.B with NULL values for the m columns, and the following M rows contain the rows of A.G with NULL values for the n columns. When matching A rows with the UNION rows, we distinguish whether a UNION row comes from table A.B or A.g by checking whether g.ida is NULL or b.ida is NULL, respectively.

Notice that the contains-base definition given in case 2 applies here as well.

4. `NESTED TABLES (A a descendants(A.B b, A.B.C c, A.B.E e, A.G.J.K k))`

   is equivalent to

   ```
   A a LOJ
   (
   A.B b LOJ
   (A.B.C c UNION A.B.E e)
   ON (b.ida = c.ida and b.idb = c.idb or c.ida IS NULL) and
   (b.ida = e.ida and b.idb = e.idb or e.ida IS NULL)
   UNION
   A.G.J.K k
   )
   ON (a.ida = b.ida or b.ida IS NULL) and (a.ida = k.ida or k.ida IS
   NULL)
   ```

   This example is just a more complex version of case 3.

5.
   ```
   NESTED TABLES (A.B b ancestors(A a ON a.a1 = “abc”)
   descendants(A.B.C c ON c.c1 > 10,
   A.B.C.D d,
   A.B.E e))
   ```

   is equivalent to

   ```
   (A.B b LOJ A a ON b.ida = a.ida and a.a1 = “abc”) LOJ
   (
   A.B.C c LOJ A.B.C.D
   ON c.ida = d.ida and c.idb = d.idb and c.idc = d.idc
   UNION
   E
   )
   ```

```
ON (b.ida = c.ida and b.idb = c.idb or c.ida IS NULL) and
(b.ida = e.ida and b.idb = e.idb or e.ida IS NULL)
```

This is an example that just puts everything together. It contains both ANCESTOR and DESCENDANT clauses, as well as ON predicates. The mapping from NESTED TABLES to LOJs uses the same patterns as in the previous cases. The ON predicates are just and-ed with the join predicates. In most cases, the ON predicates inside a NESTED TABLES will be local predicates on the right table of an LOJ, but more generally, they can also reference any columns from any table that is an ancestor of the table the ON appears next to.

Apart from two implementation restrictions, the NESTED TABLES feature can be combined with all the other features of SQL for Oracle NoSQL Database. For example, NESTED TABLES queries may also contain order by, group by, offset and limit, etc. The two restrictions are the following:

- Order by and group by can be done only on fields of the target table (if an appropriate index exists on these fields, as explained in the ORDER BY Clause and GROUP BY Clause sections).
- If the NESTED TABLES contains a DESCENDANTS clause we cannot order by the primary-key columns of the target table in descending order. For example, the following query will raise an error: select * from nested tales(A a descendants(A.B b)) order by a.ida desc.

Example: Joining Tables

Let’s consider an application that tracks a population of users and the emails sent or received by these users. Given that SQL for Oracle NoSQL Database does not currently support general purpose joins, the emails are stored in a table that is created as a child of users, so that queries can be written that combine information from both tables using the NESTED TABLES clause. The create table statements for the two tables are shown below. Notice that it is possible for the application to receive emails that are not associated with any user in the users table; such emails will be assigned a uid value that does not exist in the users table.

```
CREATE TABLE users(
uid INTEGER,
name string,
email_address string,
salary INTEGER,
address json,
PRIMARY KEY(uid));

CREATE TABLE users.emails(
eid INTEGER,
sender_address string, // sender email address
receiver_address string, // receiver email address
time timestamp(3),
size INTEGER,
content string,
PRIMARY KEY(eid));
```

Here are two queries that can be written over the users and emails tables.

Example 6-58 Joining Tables

Count the number of emails sent in 2017 by all users whose salary is greater than 200K

```
SELECT count(eid)
FROM NESTED TABLES(
users
descendants(users.emails ON email_address = sender_address and
year(time) = 2017)
)
WHERE salary > 200;
```

In the above query, we are using count(eid) rather than count(*) because there may exist users with no emails or no sent emails. For such a user, the FROM clause will return a record where the eid field will be NULL, and count(eid) will not count such a record.

Example 6-59 Joining Tables

For each email whose size is greater than 100KB and was sent by a user in the the users table, return the name and address of that user.

```
SELECT name, address
FROM NESTED TABLES(users.emails ancestors(users))
WHERE size > 100 AND sender_address = email_address;
```

In the above query will return duplicate results for any user who has sent more than one "large" email. Currently, SQL for Oracle NoSQL Database does not support SELECT DINSTINCT, so the duplicate elimination has to be performed by the application.

Left Outer Join (LOJ)

Note:

The general left outer join operator described here is not supported by Oracle NoSQL Database. This section serves only as background information for users not already familiar with left outer join.

In standard SQL, the syntax for the left outer join looks like this:

```
left_outer_join ::=
   table_reference LEFT [OUTER] JOIN table_reference ON expression
```

where table_ref may be an existing table or any other expression the returns a table (like another left outer join). The table to the left of the LEFT OUTER JOIN keywords is called the left table, and the one to the right of LEFT OUTER JOIN is the right table. The expression after the ON keyword is a condition expression, so it's supposed to return a boolean value.

Like other kind of joins, the LOJ creates a result set containing pairs of matching rows from the left and right tables. However, a LOJ will also preserve all rows of the left table, that is, a left row that does not have a matching row will appear in the result, paired with a NULL value in place of a right row. More formally, a LOJ is, conceptually, processed as follows:

1. The Cartesian product of the 2 tables is formed, i.e., a set S1 of row pairs in which every left row is paired with every right row.
2. The ON expression is evaluated for each pair of rows in S, removing from S all pairs that do not satisfy the condition.
3. Every left row that does not appear in S paired with a right tow, is paired with NULL values for the right table columns and added back to the S. S is now the final result of the LOJ and is treated like any other table by the rest of the query.

Let’s look at an example, using 2 tables: departments and employees. The definition of the tables and some sample rows are shown below.

```
CREATE TABLE departments (
   did INTEGER, location STRING, mission STRING, PRIMARY KEY(did));
CREATE TABLE employees (
eid INTEGER, did INTEGER, name STRING, salary INTEGER, PRIMARY
KEY(eid));

{ "did" : 10, "location" : "Hawaii", "mission":"surfing" }
{ "did" : 20, "location" : "Toronto", "mission":"work" }
{ "did" : 30, "location" : "Everest", "mission":"climbing" }
{ "did" : 40, "location" : "Boston", "mission":"work" }
{ "eid" : 100, "did": 10, "name":"Mark", "salary":110000 }
{ "eid" : 200, "did": 10, "name":"Dave", "salary":300000 }
{ "eid" : 300, "did": 30, "name":"Linda", "salary":120000 }
{ "eid" : 400, "did": 30, "name":"Charlie", "salary":100000 }
{ "eid" : 500, "did": 40, "name":"Sam", "salary":90000 }
{ "eid" : 600, "did": 40, "name":"Tim", "salary":100000 }
```

The following query will return the location and mission of each department together with the name of the employees who earn more then 100,000 in that department. The ON expression has 2 predicates: The 1st is the join predicate, which determines the matching condition between a department and an employee row. The 2nd is a local predicate on the employee rows. The result of the query is shown below.

```
SELECT mission, location, name
   FROM departments d LEFT OUTER JOIN employees e
   ON d.did = e.did and salary > 100000

{ "mission":"surfing", "location":"Hawaii", "name":"Mark" }
{ "mission":"surfing", "location":"Hawaii", "name":"Dave" }
{ "mission":"work", "location":"Toronto", "name":NULL }
{ "mission":"climbing", "location":"Everest", "name":"Linda" }
{ "mission":"work", "location":"Boston", "name":NULL }
```

As shown in this result, the 3rd and 5th rows return the departments in Toronto and Boston respectively, but with no associated employees. Toronto has no employees at all, and Boston has no employee earning more than 100,000.

Finally, let’s consider the following query, where the salary predicate is placed in the WHERE clause rather than the ON expression.

```
SELECT mission, location, name
   FROM departments d LEFT OUTER JOIN employees e
   ON d.did = e.did
   WHERE salary > 100000
```

The result of this query is shown below. We can see that it does not contain the Toronto and Boston departments. This is because the WHERE predicate is applied on the result of the FROM clause rather than as part of the LOJ processing. So, for the Toronto department, the LOJ produces a row that has NULL on the salary column, which is then eliminated by the WHERE clause. For the Boston department, the LOJ produces 2 rows with salaries of 90,000 and 100,000, both of which are also eliminated by the WHERE.

```
{ "mission":"surfing", "location":"Hawaii", "name":"Mark" }
{ "mission":"surfing", "location":"Hawaii", "name":"Dave" }
{ "mission":"climbing", "location":"Everest", "name":"Linda" }
```

7
Data Row Management

This chapter describes data rows and inserting and managing data rows in Oracle NoSQL Database.

This chapter contains the following topics:

INSERT Statement

The INSERT statement is used to construct a new row and add it in a specified table.

Syntax

```
insert_statement ::=
   [variable_declaration]
   (INSERT | UPSERT) INTO table_name
   [[AS] table_alias]
   ["(" id ("," id)* ")"]
   VALUES "(" insert_clause ("," insert_clause)* ")"
   [SET TTL ttl_clause]
   [returning_clause]

insert_clause ::= DEFAULT | expression
```

```
returning_clause ::= RETURNING select_list
```

Semantics

If the INSERT keyword is used, the row will be inserted only if it does not exist already. If the UPSERT keyword is used, the row will be inserted if it does not exist already, otherwise the new row will replace the existing one.

Insert statements may start with declarations of external variables that are used in the rest of the statement. See Variable Declaration. However, contrary to queries, external variables can be used in inserts without being declared. This is because inserts do not query any data, and as result, knowing the type of external variables in advance is not important as there isn't any query optimization to be done.

Optional column(s) may be specified after the table name. This list contains the column names for a subset of the table's columns. The subset must include all the primary key columns. If no columns list is present, the default columns list is the one containing all the columns of the table, in the order they are specified in the CREATE TABLE Statement section.

The columns in the columns list correspond one-to-one to the expressions (or DEFAULT keywords) listed after the VALUES clause (an error is raised if the number of expressions/DEFAULTs is not the same as the number of columns). These expressions/DEFAULTs compute the value for their associated column in the new row. Specifically, each expression is evaluated and its returned value is cast to the type of its associated column. The cast behaves like the cast expression as described in the Cast Expression section. An error is raised if an expression returns more than one item. If an expression returns no result, NULL is used as the result of that expression. If instead of an expression, the DEFAULT keyword appears in the VALUES list, the default value of the associated column is used as the value of that column in the new row. The default value is also used for any missing columns, when the number of columns in the columns list is less than the total number of columns in the table.

The expressions in the VALUES list may reference external variables, which unlike query statements, do not need to be declared in a declarations section.

Following the VALUES list a SET TTL clause may be used to set the expiration time of the new row, or in case of UPSERT, update the expiration time of an existing row. As described in the CREATE TABLE Statement section, every row has an expiration time, which may be infinite, and which is computed in terms of a Time-To-Live (TTL) value that is specified as a number of days or hours. Specifically, for a TTL value of N hours/days, where N is greater than zero, the expiration time is computed as the current time (in UTC) plus N hours/days, rounded up to the next full hour/day. For example, if the current time is 2017-06-01T10:05:30.0 and N is 3 hours, the expiration time will be 2017-06-01T14:00:00.0. If N is 0, the expiration time is infinite.

As shown in the syntax, the SET TTL clause comes in two flavors. When the USING TABLE DEFAULT syntax is used, the TTL value is set to the table default TTL that was specified in the CREATE TABLE statement. Otherwise, the SET TTL contains an expression, which computes a new TTL value. If the result of this expression is empty, the default TTL of the table is used. Otherwise, the expression must return a single numeric item, which is cast to an integer N. If N is negative, it is set to 0. To the right of the TTL expression, the keyword HOURS or DAYS must be used to specify whether N is a number of hours or days, respectively.

If the insert statement contains a SET TTL clause, an expiration time is computed as described above and applied to the row being inserted. If no SET TTL clause is used, the default table TTL is used to compute the expiration time of the inserted row. In case of MR Tables, when this row is replicated to other regions, its expiration time is also replicated as an absolute timestamp value. Therefore, the replicated rows will expire along with the original row, irrespective of when they were replicated. If the same row is inserted with a TTL value in multiple regions, then the TTL value will be set in all regions to the value held in the row with the greatest write timestamp.

The last part of the insert statement is the RETURNING clause. If not present, the result of the update statement is a record with a single field whose name is "NumRowsInserted" and whose value is the number of rows inserted: 0 if the row existed already, or 1 otherwise. Otherwise, if there is a RETURNING clause, it acts the same way as the SELECT clause: it can be a "*", in which case, a full row will be returned, or it can have a list of expressions specifying what needs to be returned. In the case of an INSERT where no insertion is actually done (because the row exists already), the RETURNING clause acts on the existing row. Otherwise, the RETURNING clause acts on the new row.

Example 7-1   Inserting a Row

The following statement inserts a row to the *users* table from CREATE TABLE Statement. Notice that the value for the *expenses* column will be set to NULL, because the DEFAULT clause is used for that column.

```
INSERT INTO users VALUES (
    10,
    "John",
    "Smith",
    [ {"first" : "Johny", "last" : "BeGood"} ],
    22,
    45000,
    { "street" : "Main", "number" : 10, "city" : "Reno", "state" :
"NV"},
    [ 30, 55, 43 ],
    DEFAULT
);

INSERT INTO users VALUES (
    20,
    "Mary",
    "Ann",
    null,
    43,
    90000,
    { "street" : "Main", "number" : 89, "city" : "San Jose", "state" :
"CA"},
    null,
    DEFAULT
);

INSERT INTO users VALUES (
    30,
    "Peter",
    "Paul",
    null,
    25,
    53000,
    { "street" : "Main", "number" : 3, "city" : "Fresno", "state" :
"CA"},
    null,
    DEFAULT
);
```

Inserting Rows with an IDENTITY Column

The system generates an IDENTITY column value when the keyword DEFAULT is used as the insert_clause for the IDENTITY column. Here are a few examples that show INSERT statements for both flavors of the IDENTITY column – GENERATED BY DEFAULT and GENERATED ALWAYS.

Example 7-2 Inserting Rows with an IDENTITY Column

```
CREATE Table Test_SGSqlInsert2(
    id INTEGER,
    name STRING,
    deptId INTEGER GENERATED ALWAYS AS IDENTITY (CACHE 1),
PRIMARY KEY(id));


INSERT INTO Test_SGSqlInsert2 VALUES (148, 'sally', DEFAULT);
INSERT INTO Test_SGSqlInsert2 VALUES (250, 'joe', DEFAULT);
INSERT INTO Test_SGSqlInsert2 VALUES (346, 'dave', DEFAULT);
```

The above INSERT statement will insert the following rows. The system generates values 1, 2, and 3 for IDENTITY column deptId.

```
148, 'sally', 1
250, 'joe', 2
346, 'dave', 3
```

The system will raise an exception for the following INSERT since the user supplied a value of 200 for an IDENTITY GENERATED ALWAYS column.

```
INSERT INTO Test_SGSqlInsert2 VALUES (1, 'joe', 200);
```

Example 7-3 Inserting Rows with an IDENTITY Column

```
CREATE TABLE Test_SGSqlInsert_Default (
    id INTEGER,
    name STRING,
    deptId INTEGER GENERATED BY DEFAULT AS IDENTITY (
        START WITH 1
        INCREMENT BY 1
        MAXVALUE 100),
PRIMARY KEY (SHARD(deptId), id));


INSERT INTO Test_SGSqlInsert_Default VALUES (100, 'tim', DEFAULT);
INSERT INTO Test_SGSqlInsert_Default VALUES (200, 'dave', 210);
INSERT INTO Test_SGSqlInsert_Default VALUES (300, 'sam', 310);
INSERT INTO Test_SGSqlInsert_Default VALUES (400, 'Jennifer', DEFAULT);
INSERT INTO Test_SGSqlInsert_Default VALUES (500, 'Barbara', 2);
```

The above statements will insert the following rows into the database. Notice that user supplied values 210, 310 and 2 are accepted because the IDENTITY column is defined as GENERATED BY DEFAULT. Note that IDENTITY column value 2 is a duplicate; the system does not check for duplicates or enforce uniqueness of the GENERATED BY DEFAULT IDENTITY column values. It is the application’s responsibility to ensure that there are no duplicate values.

```
100, 'tim', 1
200, 'dave', 210
300, 'sam', 310
400, 'Jennifer', 2
500, 'Barbara', 2
```

DELETE Statement

The DELETE statement is used to remove from a table a set of rows satisfying a condition.

Syntax

```
delete_statement ::=
   [variable_declaration]
   DELETE FROM table_name [[AS] table_alias]
   WHERE expression
   [returning_clause]

returning_clause ::= RETURNING select_list
```

Semantics

The delete statement is used to delete from a table a set of rows satisfying a condition. The condition is specified in a WHERE clause that behaves the same way as in the SELECT expression. The result of the DELETE statement depends on whether a RETURNING clause is present or not. Without a RETURNING clause the DELETE returns the number of rows deleted. Otherwise, for each deleted row the expressions following the RETURNING clause are computed the same way as in the SELECT clause and the result is returned to the application. Finally, the DELETE statement may start with declarations of external variables used in the rest of the statement. As in queries, such declarations are mandatory.

If any error occurs during the execution of a DELETE statement, there is a possibility that some rows will be deleted and some not. The system does not keep track of what rows got deleted and what rows are not yet deleted. This is because Oracle NoSQL

Database focuses on low latency operations. Long running operations across shards are not coordinating using two-phase commit and lock mechanism. In such cases, it is recommended that the application re-run the DELETE statement.

Example 7-4   Deleting Rows with SQL

The following statement deletes all users whose age is less than 16, returning the first and last name of each deleted user.

```
DELETE FROM users
WHERE age < 16
RETURNING firstName, lastName;
```

UPDATE Statement

An update statement can be used to update a row in a table.

Syntax

```
update_statement ::=
   UPDATE table_name [[AS] table_alias]
   update_clause ("," update_clause)*
   WHERE expression
   [returning_clause]

returning_clause ::= RETURNING select_list
```

Semantics

The update takes place at the server, eliminating the read-modify-write cycle, that is, the need to fetch the whole row at the client, compute new values for the targeted fields (potentially based on their current values) and then send the whole row back to the server.

Both syntactically and semantically, the update statement of the Oracle NoSQL Database is similar to the update statement of standard SQL, but with extensions to handle the richer data model of the Oracle NoSQL Database. So, as shown by the syntax above:

- First, the table to be updated is specified by its name and an optional table alias (the alias may be omitted only if top-level columns only are to be accessed; otherwise, as in read-only queries, the alias is required as the first step of path expressions that access nested fields).
- Then come one or more update clauses.
- The `WHERE` clause specifies what rows to update. In the current implementation, only single-row updates are allowed, so the `WHERE` clause must specify a complete primary key.
- Finally, there is an optional `RETURNING` clause. If not present, the result of the update statement is the number of rows updated. In the current implementation, this number will be 1 or 0. Zero will be returned if there was no row satisfying the conditions in `WHERE` clause, or if the updates specified by the update clauses turned out to be no-ops for the single row selected by the `WHERE` clause. Otherwise, if there is a `RETURNING` clause, it acts the same way as the `SELECT` clause: it can

be a "*", in which case, the full updated row will be returned, or it can have a list of expressions specifying what needs to be returned. Furthermore, if no row satisfies the `WHERE` conditions, the update statement returns an empty result.

Update Clauses

Syntax

```
update_clause ::=
   (SET set_clause ("," (update_clause | set_clause))*) |
   (ADD add_clause ("," (update_clause | add_clause))*) |
   (PUT put_clause ("," (update_clause | put_clause))*) |
   (REMOVE remove_clause ("," remove_clause)*) |
   (SET TTL ttl_clause ("," update_clause)*)
```

Semantics

There are 5 kinds of update clauses:

`SET`
Updates the value of one or more existing fields. See SET Clause.

`ADD`
Adds new elements in one or more arrays. See ADD Clause.

`PUT`
Adds new fields in one or more maps. It may also update the values of existing map fields. See PUT Clause.

`REMOVE`
Removes elements/fields from one or more arrays/maps. See REMOVE Clause.

`SET TTL`
Updates the expiration time of the row. See SET TTL Clause.

The update clauses are applied immediately, in the order they appear in the update statement, so the effects of each clause are visible to subsequent clauses. Although the syntax allows for multiple `SET TTL` clauses, only the last one will be effective; the earlier ones, if any, are ignored.

The `SET`, `ADD`, `PUT`, and `REMOVE` clauses start with a target expression, which computes the items to be updated or removed. In all cases, the target expression must be either a top-level column reference of a path expression starting with the table alias. If the target expression returns nothing, the update clause is a no-op.

SET Clause

Syntax

```
set_clause ::= path_expression "=" expression
```

Semantics

The SET clause consists of two expressions: the target expression and the new-value expression. The target expression returns the items to be updated. Notice that a target item may be atomic or complex, and it will always be nested inside a complex item (its parent item). For each such target item, the new-value expression is evaluated, and its result replaces the target item within the parent item.

If the target expression returns a NULL item, then either the target item itself is the NULL item, or one of its ancestors is NULL. In the former case, the target item will be replaced by the new item. In the latter case the SET is a no-op.

The new-value expression may return zero or more items. If it returns an empty result, the SET is a no-op. If it returns more than one item, the items are enclosed inside a newly constructed array (this is the same as the way the SELECT clause treats multi-valued expressions in the select list). So, effectively, the result of the new-value expression contains at most one item. This new item is then cast to the type expected by the parent item for the target field. This cast behaves like the cast expression as described in the Cast Expression section. If the cast fails, an error is raised; otherwise, the new item replaces the target item within the parent item.

The new-value expression may reference the implicitly declared variable $, which is bound to the current target item. Use of the $ variable makes it possible to have target expressions that return more than one item. As mentioned already, in this case, the SET clause will iterate over the target items, and for each target item T, bind the $ variable to T, compute the new-value expression, and replace T with the result of the new-value expression.

What if the new-value expression is the (reserved) keyword null? Normally, null is interpreted as the json null value. However, if the parent of the target item is a record, then null will be interpreted as the SQL NULL, and the targeted record field will be set to the SQL NULL.

See Example: Updating Rows.

ADD Clause

Syntax

```
add_clause ::=
   path_expression [add_expression] expression
```

Semantics

The ADD clause is used to add new elements into one or more arrays. It consists of a target expression, which should normally return one or more array items, an optional position expression, which specifies the position within each array where the new elements should be placed, and a new-elements expression that returns the new elements to insert.

The ADD clause iterates over the sequence returned by the target expression. For each target item, if the item is not an array it is skipped. Otherwise, the position expression (if present) and the new-elements expression are computed for the current target array. These two expressions may reference the $ variable, which is bound to the current target array.

If the new-values expression returns nothing, the ADD is a no-op. Otherwise, each item returned by this expression is cast to the element type of the array. An error is raised if any of these casts fail. Otherwise, the new elements are inserted into the target array, as described below.

If the position expression is missing, or if it returns an empty result, the new elements are appended at the end of the target array. An error is raised if the position expression returns more than one item or a non-numeric item. Otherwise, the returned item is cast to an integer. If this integer is less than 0, it is set to 0. If it is greater or equal to the array size, the new elements are appended. Otherwise, if the integer position is P and the new-elements expression returns N items, the 1st item is inserted at position P, the 2nd at position P+1, and so on. The existing array elements at position P and afterwards are shifted N positions to the right.

See Example: Updating Rows.

PUT Clause

Syntax

```
put_clause ::= path_expression expression
```

Semantics

The PUT clause is used primarily to add new fields into one or more maps. It consists of a target expression, which should normally return one or more map item and a new-fields expression that returns one or more maps or records, whose fields are inserted in the target maps.

The PUT clause iterates over the sequence returned by the target expression. For each target item, if the item is not a map it is skipped. Otherwise, the new-fields expression is computed for the current target map. The new-maps expression may reference the $ variable, which is bound to the current target map.

If the new-fields expression returns nothing, the PUT is a no-op. Otherwise, for each item returned by the new-fields expression, if the item is not a map or a record, it is skipped, else, the fields of the map/record are "merged" into the current target map. This merge operation will insert a new field into the target map if the target map does not already have a field with the same key; Otherwise, it will set the value of the target field to the value of the new field.

See Example: Updating Rows.

REMOVE Clause

Syntax

```
remove_clause ::= path_expression
```

Semantics

The remove clause consists of a single target expression, which computes the items to be removed. The REMOVE clause iterates over the target items. For each such item, if its parent is a record, an error is raised. Otherwise, if the target item is not NULL, it is removed from its parent. If the target item is NULL, then since arrays and map cannot contain NULLs, one of its ancestors must be NULL. In this case, the NULL is skipped.

See Example: Updating Rows.

SET TTL Clause

Syntax

```
ttl_clause ::=
   (add_expression (HOURS | DAYS)) |
   (USING TABLE DEFAULT)
```

Semantics

If a SET TTL clause is used with an UPDATE statement, a new expiration time is computed and applied to the row being updated. In case of MR Tables, the rows replicated to other regions carry the recalculated expiration time of the row being updated. Therefore, this row will have the same expiration time in all the regions after successful replication. If a TTL value is updated to the same row in multiple regions, then the TTL value will be set in all regions to the value held in the row with the greatest write timestamp.

See Example: Updating TTL.

Updating rows with an IDENTITY Column

An IDENTITY column that is defined as GENERATED ALWAYS cannot be updated. Only IDENTITY column that is defined as GENERATED BY DEFAULT can be updated.

Example: Updating Rows

Let’s assume a table, called "People", with only two columns: an integer "id" column and an "info" column of type JSON. Furthermore, let’s assume the following row to be updated:

```
CREATE TABLE People (
    id INTEGER,
    info JSON,
PRIMARY KEY(id));


INSERT INTO People VALUES (
    0,
    {
        "firstName":"John",
        "lastName":"Doe",
        "profession":"software engineer",
        "income":200000,
        "address": {
            "city" : "San Fransisco",
            "state" : "CA",
            "phones" : [
                { "areacode":415, "number":2840060, "kind":"office" },
                { "areacode":650, "number":3789021, "kind":"mobile" },
                { "areacode":415, "number":6096010, "kind":"home" }
            ]
        },
        "children": {
            "Anna" : {
                "age" : 10,
                "school" : "school_1",
                "friends" : ["Anna", "John", "Maria"]
            },
            "Ron" : { "age" : 2 },
            "Mary" : {
                "age" : 7,
                "school" : "school_3",
                "friends" : ["Anna", "Mark"]
            }
        }
    }
);
```

The following update statement updates various fields in the above row:

```
UPDATE People p
    SET p.info.profession = "surfing instructor",
    SET p.info.address.city = "Santa Cruz",
    SET p.info.income = p.info.income / 10,
    SET p.info.children.values().age = $ + 1,
    ADD p.info.address.phones
      0 { "areacode":831, "number":5294368, "kind":"mobile" },
    REMOVE p.info.address.phones [$element.kind = "office"],
    PUT p.info.children.Ron { "friends" : ["Julie"] },
    ADD p.info.children.values().friends seq_concat("Ada", "Aris")
WHERE id = 0
RETURNING *;
```

After the update, the row looks like this:

```
{
    "id":0,
    "info":{
        "firstName":"John",
        "lastName":"Doe",
        "profession":"surfing instructor",
        "income":20000,
        "address":{
            "city":"Santa Cruz",
            "phones":[
                {"areacode":831,"kind":"mobile","number":5294368},
                {"areacode":650,"kind":"mobile","number":3789021},
                {"areacode":415,"kind":"home","number":6096010}
            ],
            "state":"CA"
        },
            "children":{
                "Anna":{
                    "age":11,
                    "friends":["Anna","John","Maria","Ada","Aris"],
                    "school":"school_1"
                },
                "Ron":{
                    "age":3,
                    "friends":["Julie","Ada","Aris"]
                },
                "Mary":{
                    "age":8,
                    "friends":["Anna","Mark","Ada","Aris"],
                    "school":"school_3"
                }
            }
        }
    }
```

The first two SET clauses change the profession and city of John Doe. The third SET reduces his income to one-tenth. The fourth SET increases the age of his children by 1. Notice the use of the $ variable here: the expression p.info.children.values().age returns 3 ages; The SET will iterate over these ages, bind the $ variable to each age in turn, compute the expression $ + 1 for each age, and update the age with the new value. Notice that the income update could (and can) also have used a $ variable: set p.info.income = $ / 10. This would have saved the re-evaluation of the p.info.income path on the right-hand side or the "=".

The ADD clause adds a new phone at position 0 inside the phones array. The REMOVE removes all the office phones (only one in this example). The PUT clause adds a friend for Ron. In this clause, the expression p.info.children.Ron returns the value associated with the Ron child. This value is a map (the json object { "age" : 3 }) and becomes the target of the update. The 2nd expression in the PUT ({ "friends" : ["Julie"] }) constructs and returns a new map. The fields of this map are added to the target map. Finally, the last ADD clause adds the same two new friends to each child. See any* seq_concat(any*, ...) function.

Notice that the update query in this example would have been exactly the same if instead of type JSON, the info column had the following RECORD type:

```
RECORD(
    firstName STRING,
    lastName STRING,
    profession STRING,
    income INTEGER,
    address RECORD(
        city STRING,
        state STRING,
        phones ARRAY(
            RECORD(
                areacode INTEGER,
                number INTEGER,
                kind STRING
            )
        )
    ),
    children MAP(
        RECORD(
            age INTEGER,
            school STRING,
            friends ARRAY(STRING)
        )
    )
)
```

Example: Updating JSON Data

This is an example of handling heterogeneity in json documents. Assume that the Peoples table contains the row from earlier example, as well as the following row:

```
INSERT INTO People VALUES (
    1,
    {
        "firstName":"Jane",
        "lastName":"Doe",
        "address": {
            "city": "Santa Cruz",
            "state" : "CA",
            "phones" : [
                { "areacode":831, "number":5294368, "kind":"mobile" }
            ]
        }
    }
);
```

Jane has a single phone which is not stored inside an array, but as a single json object. We want to write an update query to add a phone for a person. The query must work on any row, where some rows store phones as an array, and others store it as a single json object. Here it is:

```
DECLARE $id INTEGER;
$areacode INTEGER;
$number INTEGER;
$kind STRING;

UPDATE People p
ADD p.info.address[$element.phones IS OF TYPE (ARRAY(any))].phones
    { "areacode" : $areacode, "number" : $number, "kind" : $kind },
SET p.info.address[$element.phones is of type (map(any))].phones =
    [ $, { "areacode" : $areacode, "number" : $number, "kind" :
$kind } ]
WHERE id = $id;
```

In the ADD clause, the expression p.info.address[$element.phones is of type (array(any))].phones checks whether the phones field of the address is an array and if so, returns that array, which becomes the target of the ADD. The 2nd expression in the add is a json-object constructor, that creates the new phone object and appends it into the target array.

Note:

Although p.info.address is not an array, an array filtering step is applied to it in the target expression. This works fine because (as explained in the Array-Filter Step Expressions section) the address will be treated as an array containing the address as its only element; as a result, $element will be bound to the address item, and if the filtering condition is true, the address items becomes the input to the next step in the path expression.

In the SET clause, the first expression checks whether the phones field of the address is a json object, and if so, returns that phone object, which becomes the target of the SET. The second expression constructs a new array with 2 json objects: the first is the existing phone object and the second is the newly constructed phone object. The target object is then replaced with the newly constructed array.

Example: Updating TTL

This example demonstrates an update of the expiration time of a row. Let's assume that the People table was created with a TTL value of 10 hours and a row with id 5 was inserted at time 2017-06-01T10:05:30.0. No explicit TTL was given at insertion time, so the expiration time computed at that time is 2017-06-01T21:00:00.0. Finally, let's assume that the following update statement is executed at time 2017-06-01T12:35:30.0 (2.5 hours after insertion)

```
UPDATE People $p
SET TTL remaining_hours($p) + 3 hours
WHERE id = 5;
```

The above statement extends the life of a row by 3 hours. Specifically, the remaining_hours function returns the number of full hours remaining until the expiration time of the row. See integer remaining_hours(AnyRecord) function. In this example, this number is 8. So, the new TTL value is 8+3 = 11, and the expiration time of the row will be set to 2017-06-02:T08:00:00.0.

Notice the use of the '$' character in naming the table alias for People. This is required so that the table alias acts as a row variable (a variable ranging over the rows of the table) and as a result it can be passed as the argument to the remaining_hours function (if the $ were not used, then calling remaining_hours(p) would return an error, because p is interpreted as a reference to a top-level table column with name "p").

Example: Updating IDENTITY defined as GENERATED ALWAYS

```
CREATE TABLE Test_sqlUpdateAlways (
    idValue INTEGER GENERATED ALWAYS AS IDENTITY,
    name STRING,
PRIMARY KEY(idValue));

INSERT INTO Test_sqlUpdateAlways VALUES (DEFAULT, 'joe');
INSERT INTO Test_sqlUpdateAlways VALUES (DEFAULT, 'jasmine');
```

The Test-sqlUpdateAlways table will have the following rows:

```
1, 'joe'
2, 'jasmine'
```

```
UPDATE Test_sqlUpdateAlways SET idValue = 10 WHERE name=joe;
```

The above UPDATE statement will raise an exception saying that a user cannot set a value for an IDENTITY column that is defined as GENERATED ALWAYS.

Example: Updating IDENTITY defined as GENERATED BY DEFAULT

```
CREATE TABLE Test_sqlUpdateByDefault (
    idValue INTEGER GENERATED BY DEFAULT AS IDENTITY,
    acctNum LONG,
    name STRING,
primary key(acctNum));

INSERT INTO Test_sqlUpdateByDefault VALUES (DEFAULT, 123456, 'joe');
INSERT INTO Test_sqlUpdateByDefault VALUES (400, 23456,'sam');
INSERT INTO Test_sqlUpdateByDefault VALUES (500, 34567,'carl');
```

Table Test-sqlUpdateByDefault will have the following rows:

```
1, 123456, 'joe'
400, 23456, 'jasmine'
500, 34567, 'carl'
```

```
UPDATE Test_sqlUpdateByDefault
SET idValue = 100
WHERE acctNum = 123456;
```

The above UPDATE statement will replace row (1, 123456, 'joe') with (100, 123456, 'joe') in the database.

8

Indexes

This chapter describes indexes and how to create and manage indexes in Oracle NoSQL Database.

Indexes are ordered maps, mapping values contained in the rows of a table back to the containing rows. As such, indexes provide fast access to the rows of a table, when the information we are searching for is contained in the index. In the Create Index Statement section we define the semantics of the CREATE INDEX statement and describe how the contents of an index are computed. As we will see, the Oracle NoSQL Database comes with rich indexing capabilities, including indexing of deeply nested fields, and indexing of arrays and maps. In the Create Index Statement section, we will discuss indexes on strongly typed data only; indexes on JSON data will be discussed in the Indexing JSON section.

This chapter contains the following topics:

CREATE INDEX Statement

The create index statement creates an index and populates it with entries computed from the current rows of the specified table. Currently, all indexes in Oracle NoSQL Database are implemented as B-Trees.

Syntax

```
create_index_statement ::=
   CREATE INDEX [IF NOT EXISTS] index_name
   ON table_name "(" path_list ")" [WITH NO NULLS][comment]

index_name ::= id
path_list ::= index_path ("," index_path)*
index_path ::=
   name_path [path_type] |
   keys_expression |
   values_expression [path_type] |
   brackets_expression [path_type]
```

```
name_path ::= field_name ("." field_name)*
field_name ::= id | DSTRING
keys_expression ::= name_path "." KEYS "(" ")"
values_expression ::=
   name_path "." VALUES "(" ")" ["." name_path]
brackets_expression ::=
   name_path "[" "]" ["." name_path]
path_type ::= AS
   (INTEGER | LONG | DOUBLE | STRING |
   BOOLEAN | NUMBER | POINT | GEOMETRY)
```

Semantics

By default if the named index exists the statement fails. However, if the optional "IF NOT EXISTS" clause is present and the index exists and the index specification matches that in the statement, no error is reported. Once an index is created, it is automatically updated by Oracle NoSQL Database when rows are inserted, deleted, or updated in the associated table.

An index is specified by its name, the name of the table that it indexes, and a list of one or more index paths expressions (or just index paths, for brevity) that specify which table columns or nested fields are indexed and are also used to compute the actual content of the index, as described below. For indexing json data, a type must be specified next to the path to the json data, as described in the Indexing JSON section.The index name must be unique among the indexes created on the same table. A create index statement may include an index-level comment that becomes part of the index metadata as uninterpreted text. COMMENT strings are displayed in the output of the "DESCRIBE" statement. Finally, a create index statement may include a WITH NO NULLS clause, indicting that rows with NULL or EMPTY values on the indexed fields should not be indexed (this is explained further later in this chapter).

An index stores a number of index entries. Index entries can be viewed as record items having a common record type (the index schema) that contains N+K fields, where N is the number of paths in the path_list and K is the number of primary key columns. The last K fields store values that constitute a primary key "pointing" to a row in the underlying table. Each of the first N fields (called index fields) has an atomic type that is one of the indexable types: numeric types, string, boolean, timestamp, or enum. Index fields may also store one of the special values: NULL, json null, or EMPTY (EMPTY will be defined later in this section, and json null is possible only for indexes on JSON data). However, if the WITH NO NULLS clause is used, NULL and EMPTY values will not apear in any index entries. In the index schema, the index fields appear in the same order as the corresponding index paths in path_list.

The index entries are sorted in ascending order. The relative position among any two entries is determined by a lexicographical (string-like) comparison of their field values (with the field values playing the role of the characters in a string). The 3 special values are considered to be greater than any other values (i.e., they sort last in indexes). Among these 3 values, their relative order is EMPTY < json null < NULL.

Indexes can be characterized as simple indexes or multi-key indexes (which index arrays or maps). In both cases, for each table row, the index path expressions are evaluated and one or more index entries are created based on the values returned by the index path expressions. This is explained further in the following sections.

Syntactically, an index path is specified using a subset of the syntax for path expressions in queries (see Path Expressions). The following remarks/restrictions apply to index paths:

- Although query path expressions must include an input expression, this is not allowed for index paths, because by default, the input is a table row that is being indexed.
- A single name_path is a list of one or more dot-separated field names. These field names may be schema fields, i.e., field names that appear in the CREATE TABLE statement used to create the associated table. Using the example table from the CREATE TABLE Statement section, age and lastName are valid index paths for indexes over the Users table. However, name_paths may also contain field names of maps. For example, expenses.books is a valid index path; in this path books is not a schema field, but instead the name of an expense category that each user may or may not have.
- When a name_path is evaluated on a table row (or more generally, on a single input item), it must return at most one item. Furthermore, no arrays should be crossed during the evaluation. We will refer to these 2 restrictions as the name-path-restriction. For example, otherNames.last cannot be used as an index path (because otherNames is an array field), even though it is a valid query path expression. However, the equivalent otherNames[].last path (which is not a single name_path) can be used instead.
- In the context of index creation, if a name_path returns an empty result, the special value EMPTY is used as the result instead. For example, the path expenses.books will return an empty result for a user that does not have a books expense category, but this empty result gets converted to EMPTY. So, in effect, each name_path returns exactly one item.
- A name_path may contain fields whose type is JSON. In this case, the index path is a json index path, and the whole index is said to be a json index. In the remainder of this section we will discuss non json indexes only. Json indexes will be discussed in the Indexing JSON section. Notice that the path_type production in the syntax of the CREATE INDEX statement is required for json index paths only; it must not be used for index paths on typed data.
- As shown by the syntax of the CREATE INDEX statement, filtering is not allowed, that is, .keys(), .values(), and [] steps cannot use a filtering condition. Furthermore, there can be at most one of such steps in each index path.
- .keys() steps can be applied to maps only (not records).

Given that json indexes are discussed in the next section, for the remainder of the current section, we will use a Users table whose address column is not of type JSON, but a record type instead. Specifically, the table definition is the following:

```
CREATE TABLE Users2 (
    id INTEGER,
    income INTEGER,
    address RECORD(
        street STRING,
        city STRING,
        state STRING,
        phones ARRAY(
            RECORD(
                area INTEGER,
                number INTEGER,
                kind STRING
            )
        )
    ),
    connections ARRAY(INTEGER),
    expenses MAP(INTEGER),
    PRIMARY KEY (id)
);
```

We will also assume that Users2 is populated with the following sample rows (shown in JSON format). Notice that the NULL values in the json documents below are converted to the SQL NULL when the documents are mapped to the table rows.

```
INSERT INTO Users2 VALUES (
    0,
    1000,
    {
        "street" : "somewhere",
        "city": "Boston",
        "state" : "MA",
        "phones" : [
            { "area":408, "number":50, "kind":"work" },
            { "area":415, "number":60, "kind":"work" },
            { "area":NULL, "number":52, "kind":"home" }
        ]
    },
    [ 100, 20, 20, 10, 20],
    { "housing" : 1000, "clothes" : 230, "books" : 20 }
);

INSERT INTO Users2 VALUES (
    1,
    NULL,
    {
        "street" : "everywhere",
        "city": "San Fransisco",
        "state" : "CA",
        "phones" : [
            { "area":408, "number":50, "kind":"work" },
            { "area":408, "number":60, "kind":"home" }
        ]
    },
    [],
    { "housing" : 1000, "travel" : 300 }
);

INSERT INTO Users2 VALUES (
    2,
    2000,
    {
        "street" : "nowhere",
        "city": "San Jose",
        "state" : "CA",
        "phones" : [ ]
    },
    NULL,
    NULL
);
```

Simple Indexes

An index is a simple one if it does not index any arrays or maps. More precisely, an index is a simple one if:

- Each index path is a simple name_path (no .keys(), .values(), or [] steps). Under the name-path-restriction described above, this name_path returns exactly one item per table row.
- The item returned by each such index path has an indexable atomic type.

We refer to paths that satisfy the above conditions as simple index paths.

Given the above definition, the content of a simple index is computed as follows: For each row R, the index paths are computed and a single index entry is created whose field values are the items returned by the index paths plus the primary-key columns of R. If the index was created with the WITH NO NULLS clause and the index entry contains any NULL or EMPTY values, the entry is discarded; Otherwise, the entry is inserted into the index. As a result, if no WITH NO NULLS clause was used, the index contains exactly one entry per table row.

Notice that indexes created with the WITH NO NULLS clause may be useful when the data contain a lot of NULL and/or non-existent values on the indexed fields, because the time and space overhead of indexing is reduced. However, use of such indexes by queries is restricted, as explained in the Using Indexes for Query Optimization section.

Example 8-1 Simple Index

```
CREATE INDEX idx1 ON Users2 (income);
```

It creates an index with one entry per user in the Users table. The entry contains the income and id of the user represented by the row. The contents of this index for the sample rows in Users2 are:

```
[ 1000, 0 ]
[ 2000, 2 ]
[ NULL, 1 ]
```

If the WITH NO NULLS clause were used in the above create index statement, the last of the above 3 entries would not appear in the index.

Example 8-2 Simple Index

```
CREATE INDEX idx2 ON Users2 (address.state, address.city, income);
```

It creates an index with one entry per user in the Users table. The entry contains the state, city, income and id of the user represented by the row. The contents of this index for the sample rows in Users2 are:

```
[ "CA", "San Fransisco", NULL, 1 ]
[ "CA", "San Jose", 2000, 2 ]
[ "MA", "Boston", 1000, 0 ]
```

Example 8-3 Simple Index

```
CREATE INDEX idx3 ON Users2 (expenses.books);
```

Creates an index entry for each user. The entry contains the user's spending on books, if the user does record spending on books, or EMPTY if there is no "books" entry in expenses, or NULL if there is no expenses map at all (i.e. the value of the expenses column is NULL). The contents of this index for the sample rows in Users2 are:

```
[ 20, 0 ]
[ EMPTY, 1 ]
[ NULL, 2 ]
```

If the WITH NO NULLS clause were used in the above create index statement, only the first of the above 3 entries would appear in the index.

Example 8-4 Simple Index

```
CREATE INDEX idx4 ON users2 (expenses.housing, expenses.travel);
```

Creates an index entry for each user. The entry contains the user's housing expenses, or EMPTY if the user does not record housing expenses, and the user's travel expenses, or EMPTY if the user does not record travel expenses. If expenses is NULL, both fields in the index entry will be NULL. The contents of this index for the sample rows in Users2 are:

```
[ 1000, 300, 1 ]
[ 1000, EMPTY, 0 ]
[ NULL, NULL, 2 ]
```

Multi-Key Indexes

Multi-key indexes are used to index all the elements of an array, or all the elements and/or all the keys of a map. As a result, for each table row, the index contains as many entries as the number of elements/entries in the array/map that is being indexed (modulo duplicate elimination). To avoid an explosion in the number of index entries, only one array/map may be indexed (otherwise, for each table row, we would have to form the cartesian product among the elements/entries or each array/map that is being indexed). A more precise definition for multi-key indexes is given in the rest of this section.

An index is a multi-key index if:

1. There is at least one index path that uses a multi-key step (.keys(), .values(), or []). Any such index path will be called a multi-key index path, and the associated index field a multi-key field. The index definition may contain more than one multi-key paths, but all multi-key paths must use the same name_path before their multi-key step. Let M be this common name_path. For example, we can index both the area codes and the phone kinds of users, that is, the paths address.phones[].area and address.phones[].kind can both appear in the same CREATE INDEX statement, in which case M is the path address.phones. On the other hand, we cannot create an index on Users2 using both of these paths: connections[] and address.phones[].area, because this is indexing two different arrays in the same index which is not allowed.
2. Any non-multi-key index paths must be simple paths, as defined in Simple Indexes section.
3. The shared path M must specify either an array or a map field and the specified array/map must contain indexable atomic items, or record items, or map items. For example, consider the following table definition:

   ```
   CREATE TABLE Foo (
       id INTEGER,
       complex1 RECORD(
           mapField MAP(ARRAY(MAP(INTEGER)))
       ),
       complex2 RECORD(
           matrix ARRAY(ARRAY(RECORD(a LONG, b LONG)))
       ),
       PRIMARY KEY(id)
   );
   ```

   The path expression complex2.matrix[] is not valid, because the result of this path expression is a sequence of arrays, not atomic items. Neither is complex2.matrix[] [].a valid, because we cannot index arrays inside other arrays (in fact this path will raise a syntax error, because the syntax allows at most one [] per index path). On the other hand, the path complex1.mapField.someKey[].someOtherKey is valid. In this case, M is the path complex1.mapField.someKey, which specifies an array containing maps. Notice that in this index path, someKey and someOtherKey are map-entry keys. So, although we are indexing arrays that are contained inside maps, and the arrays being indexed contain maps, the path is valid, because it is selecting specific entries from the maps involved rather than indexing all the map entries in addition to all the array entries.
4. If M specifies an array (i.e., the index is indexing an array-valued field):
   a. If the array contains indexable atomic items, then:
      i. The index definition must contain exactly one multi-key index path, of the form M[] (without any name_path following after the []). Again, this implies that we cannot index more than one array in the same index.
      ii. In this case, for each table row R, a number of index entries are created as follows: the simple index paths (if any) are computed on R. Then, M[] is computed (as if it were a query path expression), returning either NULL, or EMPTY, or all the elements of the array returned by M. Finally, for each value V returned by M[], an index entry is created whose field values are V and the values of the simple paths.

iii. Any duplicate index entries (having equal field values and the same primary key) created by the above process are eliminated.

iv. If the index was created with the WITH NO NULLS clause, any index entries containing NULL or EMPTY values are discarded. The remaining entries are inserted into the index.

b. If the array contains records or maps, then:

   i. All of the multi-key paths must be of the form M[].name_path. Let Ri be the name_path appearing after M[] in the i-th multi-key index path. Each Ri must return at most one indexable atomic item.

   ii. In this case, for each table row R, a number of index entries are created as follows: the simple index paths (if any) are computed on R. Then, M[] is computed (as if it were a query path expression), returning either NULL, or EMPTY, or all the elements of the array returned by M. Next, for each value V returned by M[], one index entry is created as follows: the Ri's are computed on V, returning a single indexable atomic item (which may be the NULL or EMPTY item), and an index entry is created, whose field values are the values of the simple index paths plus the values computed by the Ri's.

   iii. Any duplicate index entries (having equal field values and the same primary key) created by the above process are eliminated.

   iv. If the index was created with the WITH NO NULLS clause, any index entries containing NULL or EMPTY values are discarded. The remaining entries are inserted into the index.

5. If M specifies a map field (i.e., the index is indexing a map-valued field), the index may be indexing only map keys, or only map elements, or both keys and elements. In all cases, the definition of map indexes can be given in terms of array indexes, by viewing maps as arrays containing records with 2 fields: a field with name "key" and value a map key, and a field named "element" and value the corresponding map element (that is, MAP(T) is viewed as ARRAY(RECORD(key STRING, element T))). Then, the 3 valid kinds for map indexes are:

   a. There is a single multi-key index path using a keys() step. Using the array view of maps, M.keys() is equivalent to M[].key.

   b. There are one or more multi-key index paths, all using a .values() step. Each of these has the form M.values().Ri.. Using the array view of maps, each M.values().Ri path is equivalent to M[].element.Ri.

   c. There is one keys() path and one or more values() paths. This is just a combination of the 2 previous cases.

Example 8-5 Multi-Key Index

```
CREATE INDEX midx1 ON Users2 (connections[]);
```

Creates an index on the elements of the connections array. The contents of this index for the sample rows in Users2 are:

```
[ 10, 0 ]
[ 20, 0 ]
[ 100, 0 ]
[ EMPTY, 1 ]
[ NULL, 2 ]
```

If the WITH NO NULLS clause were used in the above create index statement, the last 2 of the above entries would not appear in the index.

Example 8-6   Multi-Key Index

```
CREATE INDEX midx2 ON Users2 (address.phones[].area, income);
```

Creates an index on the area codes and income of users. The contents of this index for the sample rows in Users2 are:

```
[ 408, 1000, 0 ]
[ 408, NULL, 1 ]
[ 415, 1000, 0 ]
[ EMPTY, 2000, 2 ]
[ NULL, 1000, 0 ]
```

Example 8-7   Multi-Key Index

```
CREATE INDEX midx3 ON Users2
    (address.phones[].area, address.phones[].kind, income);
```

Creates an index on the area codes, the phone number kinds, and the income of users. The contents of this index for the sample rows in Users2 are:

```
[ 408, "work", 1000, 0 ]
[ 408, "home", NULL, 1 ]
[ 408, "work", NULL, 1 ]
[ 415, "work", 1000, 0 ]
[ EMPTY, EMPTY, 2000, 2 ]
[ NULL, "home", 1000, 0 ]
```

Example 8-8   Multi-Key Index

```
CREATE INDEX midx4 ON Users2 (
    expenses.keys(), expenses.values());
```

Creates an index on the fields (both keys and values) of the expenses map. The contents of this index for the sample rows in Users2 are:

```
[ "books", 50, 0 ]
[ "clothes", 230, 0 ]
[ "housing", 1000, 0 ]
[ "housing", 1000, 1 ]
[ "travel", 300, 1 ]
[ NULL, NULL, 2 ]
```

SHOW INDEXES Statement

Syntax

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

Semantics

The show indexes statement provides the list of indexes present on the specified table.

AS JSON can be specified if you want the output to be in JSON format.

Example 8-9 Show Indexes

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

```
SHOW INDEXES ON users2;

indexes
  idx1
```

Example 8-10 Show Indexes

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

```
SHOW AS JSON INDEXES ON users2;

{"indexes" : ["idx1"]}
```

DESCRIBE INDEX Statement

The describe index statement provides the definition for the specified index on a table.

Syntax

```
describe_index_statement ::=
    (DESCRIBE | DESC) [AS JSON] INDEX index_name ON table_name
```

Semantics

The description for index contains the following information:

- Name of the table on which the index is defined.
- Name of the index.
- Type of index. Whether the index is primary index or secondary index.
- Whether the index is a multi-key index. If the index is multi-key then 'Y' is displayed, otherwise, 'N' is displayed.
- List of fields on which the index is defined.

- Declared type of the index.
- Description of the index.

AS JSON can be specified if you want the output to be in JSON format.

Example 8-11 Describe Index

The following statement provides information about the index idx1 on users2 table.

```
DESCRIBE INDEX idx1 on users2;

 +--------+------+-----------+----------+--------+--------------
+-------------+
 | table  | name |   type    | multiKey | fields | declaredType |
description |
 +--------+------+-----------+----------+--------+--------------
+-------------+
 | Users2 | idx1 | SECONDARY | N        | income |
|             |
 +--------+------+-----------+----------+--------+--------------
+-------------+
```

Example 8-12 Describe Index

The following statement provides information about the index idx1 on users2 table in JSON format.

```
DESCRIBE AS JSON INDEX idx1 on users2;

{
  "name" : "idx1",
  "table" : "Users2",
  "type" : "secondary",
  "fields" : [ "income" ]
}
```

DROP INDEX Statement

Syntax

```
drop_index_statement ::=
   DROP INDEX [IF EXISTS] index_name ON table_name
```

Semantics

The DROP INDEX statement removes the specified index from the database. By default if the named index does not exist this statement fails. If the optional "IF EXISTS" is specified and the index does not exist no error is reported.

Indexing JSON

In general, an index is a json index if it indexes at least one field that is contained inside json data. Json indexes are very similar to indexes on strongly typed data, both in the way their contents are evaluated as well as in the ways they are used by queries. However, since json data is schema less, the type of an indexed json field may be different across table rows. Currently, such type variation is partially allowed, depending on how the index is declared. Specifically, an index path that specifies a json field must be followed by a type declaration (using the AS keyword) in the CREATE INDEX statement. The type must be one of the json atomic types (numeric types, string, or boolean) or the anyAtomic type. Semantically, such a declaration means that the evaluation of the index path on a row must always result in a sequence of zero or more atomic items, where each item in the sequence is NULL, or json null, of has a type that is the same or a subtype of the declared type. If the sequence is empty, the special value EMPTY is put in the index.

Creation of a json index will fail if the associated table contains any rows with data that violate the restriction imposed by the declare type of the indexed json field(s). Similarly, an insert/update operation will be rejected if the new row does not conform to such type restrictions imposed by one or more existing json indexes.

Notice that declaring a json index path as anyAtomic has the obvious advantage of allowing the indexed field to have values of various types. When these values are stored in the index, they are sorted as follows: Numbers first, then strings, and boolean last. However, this type freedom comes with a space and CPU cost. This is because numeric values of any kind in the indexed field will be cast to Number before stored in the index. This cast takes CPU time and the resulting Number value will in general be larger than the original non-Number value.

A json index where all the json paths are declared as anyAtomic will be called an untyped json index.

Like the non-json indexes, typed json indexes may be simple or multi-key. These are described further in the following sections. The examples shown there are based on a users table that stores all user info as json data. Specifically, we will use the table created like this:

```
CREATE TABLE users3 (id INTEGER, info JSON, PRIMARY KEY(id));
```

We will also assume that table users3 is populated with the following sample rows (shown in JSON format). Contrary to table users2, the sub-documents under "info" remain as json data when these documents are inserted in table users3. This means that the null values inside "info" sub-documents will remain as json null values in the table. Notice however that for the row with id 3, the value of the (top-level) "info" column will be the SQL NULL. Notice also the variations in the data across different rows. For example, info.address.phones is usually an array, but in the row with id 4 it is a single json object.

```
INSERT INTO users3 VALUES (
    0,
    {
        "income" : 1000,
        "address": {
            "street" : "somewhere",
            "city": "Boston",
            "state" : "MA",
            "phones" : [
                { "area":408, "number":50, "kind":"work" },
                { "area":415, "number":60, "kind":"work" },
                { "area":null, "number":52, "kind":"home" }
            ]
        },
        "expenses" : { "housing" : 1000, "clothes" : 230, "books" : 
20 },
        "connections" : [ 100, 20, 20, 10, 20]
    }
);

INSERT INTO users3 VALUES (
    1,
    {
        "income" : null,
        "address": {
            "street" : "everywhere",
            "city": "San Fransisco",
            "state" : "CA",
            "phones" : [
                { "area":408, "number":50, "kind":"work" },
                { "area":408, "number":60, "kind":"home" },
                "4083451232"
            ]
        },
        "expenses" : { "housing" : 1000, "travel" : 300 },
        "connections" : [ ]
    }
);

INSERT INTO users3 VALUES (
    2,
    {
        "income" : 2000,
        "address": {
            "street" : "nowhere",
            "city": "San Jose",
            "state" : "CA",
            "phones" : [ ]
        },
        "expenses" : null,
        "connections" : null
    }
);

INSERT INTO users3 VALUES (3,{});

INSERT INTO users3 VALUES (
    4,
    {
        "address": {
            "street" : "top of the hill",
            "city": "San Fransisco",
            "state" : "CA",
            "phones" : { "area":408, "number":50, "kind":"work" }
        },
        "expenses" : { "housing" : 1000, "travel" : 300},
        "connections" : [ 30, 5, null ]
    }
);

INSERT INTO Users3 VALUES (
    5,
    {
        "address": {
            "street" : "end of the road",
            "city": "Portland",
            "state" : "OR"
        }
    }
);
```

Simple Typed JSON Indexes

A simple typed json index path has the form: name_path path_type. As with non-json index paths, when name_path is evaluated on any table row, it must return at most one atomic item and must not cross any arrays. For non-json indexes, the type of result item is always the same and can be deduced by the table schema. This is not possible for schemaless json data, and as a result, a path_type must be used to declare and enforce a type for the indexed field. The implication of such a declaration is that when name_path is evaluated on any table row, it must return an empty result, or NULL, or json null, or an instance of the declared type. As mentioned already, in case of an empty result, the special EMPTY value is used as the result. Furthermore, if path_type is anyAtomic, no type restrictions are placed on the index field (but at a CPU and storage cost).

If a CREATE INDEX statement consists of at least one simple typed json path and all the other paths are also simple (json or non-json), then the index is a simple typed json index. The contents of such an index are determined the same way as for non-json simple indexes.

The following examples are the json versions of the examples in Simple Indexes section.

Example 8-13   Simple Typed json Index

```
CREATE INDEX jidx1 ON users3(info.income AS INTEGER);
```

It creates an index with one entry per user in the Users table. The entry contains the income and id (the primary key) of the user represented by the row. The contents of this index for the sample rows in Users3 are:

```
[ 1000, 0 ]
[ 2000, 2 ]
[ EMPTY, 4 ]
[ EMPTY, 5 ]
[ JNULL, 1 ]
[ NULL, 3 ]
```

Example 8-14   Simple Typed json Index

```
CREATE INDEX jidx1u ON users3 (
    info.income AS ANYATOMIC)
```

It creates an untyped index on info.income. The contents of this index are the same as in jidx1 above, but the values 1000 and 200 are stored as Numbers instead of integers. If the following row is added to the users3 table:

```
INSERT INTO users3 VALUES (
    6,
    {
        "address": {},
        "expenses" : {},
        "connections" : []
    }
)

The index will look like this:
    [ "none", 6 ]
    [ EMPTY,  5 ]
    [ EMPTY,  4 ]
    [ NULL,   3 ]
    [ 2000,   2 ]
    [ JNULL,  1 ]
    [ 1000,   0 ]
```

Example 8-15   Simple Typed json Index

```
CREATE INDEX jidx2 ON users3 (
    info.address.state AS STRING,
    info.address.city AS STRING,
    info.income AS INTEGER);
```

It creates an index with one entry per user in the Users table. The entry contains the state, city, income and id (the primary key) of the user represented by the row. The contents of this index for the sample rows in Users3 are:

```
[ "CA", "San Fransisco", EMPTY, 4 ]
[ "CA", "San Fransisco", JNULL, 1 ]
[ "CA", "San Jose", 2000, 2 ]
[ "MA", "Boston", 1000, 0 ]
[ "OR", "Portland", EMPTY, 5 ]
[ NULL, NULL, NULL, 3 ]
```

Example 8-16   Simple Typed json Index

```
CREATE INDEX jidx3 ON users3 (
    info.expenses.books AS INTEGER);
```

Creates an index entry for each user. The entry contains the user's spending on books, if the user does record spending on books, or EMPTY if there is no "books" entry in expenses or there is no expenses map at all, or NULL if there is no info at all (i.e. the value of the info column is NULL). The contents of this index for the sample rows in Users3 are:

```
[ 20, 0 ]
[ EMPTY, 1 ]
[ EMPTY, 2 ]
[ EMPTY, 4 ]
[ EMPTY, 5 ]
[ NULL, 3 ]
```

Example 8-17   Simple Typed json Index

```
CREATE INDEX jidx4 ON users3 (
    info.expenses.housing AS INTEGER,
    info.expenses.travel AS INTEGER);
```

Creates an index entry for each user. The entry contains 2 fields: (a) the user's housing expenses, or EMPTY if the user does not record housing expenses or there is no expenses field at all, and (b) the user's travel expenses, or EMPTY if the user does not record travel expenses or there is no expenses field at all. If info is NULL, both fields in the index entry will be NULL. The contents of this index for the sample rows in Users3 are:

```
[ 1000, 300, 1 ]
[ 1000, 300, 4 ]
[ 1000, EMPTY, 0 ]
[ EMPTY, EMPTY, 2 ]
[ EMPTY, EMPTY, 5 ]
[ NULL, NULL, 3 ]
```

Multi-Key Typed JSON Indexes

An index is a multi-key typed json index if its definition includes at least one json index path that uses a multi-key step (keys(), values(), or []). Furthermore, if the index path uses values() or [], its declaration in the CREATE INDEX statement must be followed by a path_type. As mentioned already, when evaluated on any table row, such an index path must return zero or more items, and each item returned must be either NULL, or JNULL, or an instance of the declared type.

Multi-key typed json indexes are very similar to non-json multi-key indexes. In fact, the description in the Multi-Key Indexes section applies almost identically to json indexes as well. The only difference is that an actual row may not have an array or a map at the expected place. For example:

- Let a.b.c[].d be a json index path. The fact that [] appears after c implies that the user expects c to be an array (containing json objects having a d field). Furthermore, none of a, b, or d may be arrays. Oracle NoSQL Database enforces the later restriction, but it allows c to be any kind of item. If it’s not an array, it will be treated as if it were an array containing that single item. In other words, for each table row, the path a.b.c[].d will be evaluated as if it were a DML path expression in a query. If c is an atomic, the result will be EMPTY; if c is a map, the value of the d field in that map (if any) will be indexed.

- Let a.b.c.values().d be a json index path. The fact that values() appears after c implies that the user expects c to be a json object (containing json objects having a d field). Furthermore, none of a, b, or d may be arrays. In this case, c may not be an array either (because then all the objects in the array would be indexed, as well as all the values in each such object). However, c may be an atomic, in which case the result will be EMPTY.

The following examples are the json versions of the examples in the Multi-Key Indexes section.

Example 8-18 Multi-Key Typed json Index

```
CREATE INDEX jmidx1 ON users3 (
    info.connections[] AS INTEGER);
```

Creates an index on the elements of the connections array. The contents of this index for the sample rows in Users3 are:

```
[ 5, 4 ]
[ 10, 0 ]
[ 20, 0 ]
[ 30, 4 ]
[ 100, 0 ]
[ EMPTY, 1 ]
[ EMPTY, 5 ]
[ JNULL, 2 ]
[ JNULL, 4 ]
[ NULL, 3 ]
```

Example 8-19 Multi-Key Typed json Index

```
CREATE INDEX jmidx2 ON users3 (
    info.address.phones[].area AS INTEGER,
    info.income AS INTEGER);
```

Creates an index on the area codes and income of users. The contents of this index for the sample rows in Users3 are:

```
[ 408, 1000, 0 ]
[ 408, EMPTY, 4 ]
[ 408, JNULL, 1 ]
[ 415, 1000, 0 ]
[ EMPTY, 2000, 2 ]
[ EMPTY, EMPTY, 5 ]
[ EMPTY, JNULL, 1 ]
[ JNULL, 1000, 0 ]
[ NULL, NULL, 3 ]
```

Example 8-20 Multi-Key Typed json Index

```
CREATE INDEX jmidx2u ON users3 (
    info.address.phones[].area AS ANYATOMIC,
    info.income AS INTEGER)
```

This is a variation of the jmidx2 index, where the first index path is untyped and second is typed. The contents of jmidx2 and jmidx2u are the same, except that in jmidx2u the numeric values in the first column are stored as Numbers instead of integers.

Example 8-21 Multi-Key Typed json Index

```
CREATE INDEX jmidx3 ON users3 (
    info.address.phones[].area AS INTEGER,
    info.address.phones[].kind AS string,
    info.income AS INTEGER);
```

Creates an index on the area codes, the phone number kinds, and the income of users. The contents of this index for the sample rows in Users3 are:

```
[ 408, "home", JNULL, 1 ]
[ 408, "work", 1000, 0 ]
[ 408, "work", EMPTY, 4 ]
[ 408, "work", JNULL, 1 ]
[ 415, "work", 1000, 0 ]
[ EMPTY, EMPTY, 2000, 2 ]
[ EMPTY, EMPTY, EMPTY, 5 ]
[ EMPTY, EMPTY, JNULL, 1 ]
[ JNULL, "home", 1000, 0 ]
[ NULL, NULL, NULL, 3 ]
```

Example 8-22 Multi-Key Typed json Index

```
CREATE INDEX jmidx4 ON users3 (
    info.expenses.keys(),
    info.expenses.values() AS INTEGER);
```

Creates an index on the fields (both keys and values) of the expenses map. Notice that the keys() portion of the index definition must not declare a type. This is because the type will always be String. The contents of this index for the sample rows in Users2 are:

```
[ "books", 50, 0 ]
[ "clothes", 230, 0 ]
[ "housing", 1000, 0 ]
[ "housing", 1000, 1 ]
[ "housing", 1000, 4 ]
[ "travel", 300, 1 ]
[ "housing", 1000, 4 ]
[ EMPTY, EMPTY, 2 ]
[ EMPTY, EMPTY, 5 ]
[ NULL, NULL, 3 ]
```

9

Query Optimization

This chapter discusses about query optimization in Oracle NoSQL Database.

This chapter contains the following topics:

Using Indexes for Query Optimization

In Oracle NoSQL Database, the query processor can identify which of the available indexes are beneficial for a query and rewrite the query to make use of such an index. "Using" an index means scanning a contiguous subrange of its entries, potentially applying further filtering conditions on the entries within this subrange, and using the primary keys stored in the surviving index entries to extract and return the associated table rows. The subrange of the index entries to scan is determined by the conditions appearing in the WHERE clause, some of which may be converted to search conditions for the index. Given that only a (hopefully small) subset of the index entries will satisfy the search conditions, the query can be evaluated without accessing each individual table row, thus saving a potentially large number of disk accesses.

Notice that in Oracle NoSQL Database, a primary-key index is always created by default. This index maps the primary key columns of a table to the physical location of the table rows. Furthermore, if no other index is available, the primary index will be used. In other words, there is no pure "table scan" mechanism; a table scan is equivalent to a scan via the primary-key index.

When it comes to indexes and queries, the query processor must answer two questions:

1. Is an index applicable to a query? That is, will accessing the table via this index be more efficient than doing a full table scan (via the primary index).
2. Among the applicable indexes, which index or combination of indexes is the best to use?

Regarding question (1), for queries with NESTED TABLES, secondary indexes will be considered for the target table only; in the current implementation, ancestor and/or descendant tables will always be accessed via their primary index.

Regarding question (2), the current implementation does not support index anding or index oring. As a result, the query processor will always use exactly one index (which may be the primary-key index). Furthermore, there are no statistics on the number and distribution of values in a table column or nested fields. As a result, the query processor has to rely on some simple heuristics in choosing among the applicable indexes. In addition, SQL for Oracle NoSQL Database allows for the inclusion of index hints in the queries, which are used as user instructions to the query processor about which index to use.

Finding Applicable Indexes

To find applicable indexes, the query processor looks at the conditions in the WHERE clause, trying to "match" such predicates with the index paths that define each index and convert the matched predicates to index search conditions. In general the WHERE clause consists of one or more conditions connected with AND or OR operators, forming a tree whose leaves are the conditions and whose internal nodes are the AND/OR operators. Let a predicate be any subtree of this WHERE-clause tree. The query processor will consider only top-level AND predicates, i.e., predicates that appear as the operands of a root AND node. If the WHERE clause does not have an AND root, the whole WHERE expression is considered a single top-level AND predicate. Notice that the query processor does not currently attempt to reorder the AND/OR tree in order to put it in conjunctive normal form. On the other hand, it does flatten the AND/OR tree so that an AND node will not have another AND node as a child, and an OR node will not have another OR node as a child. For example, the expression a = 10 and b < 5 and (c > 10 or c < 0) has 3 top-level AND predicates: a = 10, b < 5, and (c > 10 or c < 0), whereas the expression a = 10 and b < 5 and c > 10 or c < 0 has an OR as its root and the whole of it is considered as a single top-level AND predicate. For brevity, in the rest of this section we will use the term "predicate" to mean top-level AND predicate.

The query processor will also look at the expressions in the ORDER BY and GROUP BY clauses in order to find sorting indexes, that is, indexes that sort the table rows according to the expressions appearing in these clauses. As explained in sections GROUP BY Clause and ORDER BY Clause,, use of a sorting index will result in more efficient and memory-sparing sorting and grouping.

The query processor will consider an index applicable to a query if the index is a sorting one or if the query contains at least one index predicate: a predicate that can be evaluated during an index scan, using the content of the current index entry only, without the need to access the associated table row. Index predicates are further categorized as start/stop predicates or filtering predicates. A start/stop predicate participates in the establishment of the first/last index entry to be scanned during an index scan. A filtering predicate is applied during the index scan on the entries being scanned. In the current implementation, the following kinds of predicates are considered as candidate start/stop predicates:

- comparisons, using either the value or sequence (any) comparison operators, but not != or !=any,
- IS NULL and IS NOT NULL operators,
- EXISTS and NOT EXISTS predicates, and
- IN predicates

However, if an index is created with the WITH NO NULLS clause, IS NULL and NOT EXISTS predicates cannot be used as index predicates for that index. In fact, such an index can be used by a query only if the query has an index predicate for each of the indexed fields.

An index is called a covering index with respect to a query if the query can be evaluated using only the entries of that index, that is, without the need to retrieve the associated rows.

If an index is used in a query, its index predicates are removed from the query because they are evaluated by the index scan. We say that index predicates are "pushed to the index". In the rest of this section we explain applicable indexes further via a number of example queries, and using the non-json indexes from the Simple Indexes and Multi-Key Indexes sections. The algorithm for finding applicable json indexes is essentially the same as for non-json indexes. The same is true for geometry indexes, with the exception that geosearch predicates that are pushed to the index are not removed from the query, because they need to stay there to eliminate false positive results from the index scans.

Examples: Using Indexes for Query Optimization

Example 9-1 Using Indexes for Query Optimization

```
SELECT * FROM Users2
WHERE 10 < income AND income < 20;
```

The query contains 2 index predicates. Indexes idx1, idx2, midx2, and midx3 are all applicable. For index idx1, 10 < income is a start predicate and income < 20 is a stop predicate. For the other indexes, both predicates are filtering predicates. If, say, idx2 were to be used, the subrange to scan is the whole index. Obviously, idx1 is better than the other indexes in this case. Notice however, that the number of table rows retrieved would be the same whether idx1 or idx2 were used. If midx2 or midx3 were used, the number of distinct rows retrieved would be the same as for idx1 and idx2, but a row would be retrieved as many times as the number of elements in the phones array of that row. Such duplicates are eliminated from the final query result set.

Notice that if index idx2 was created WITH NO NULLS, it would not be applicable to this query, because it does not have index predicates for fields address.state and address.city. For example, if Users2 contains a row where address.city is NULL and income is 15, the index would not contain any entry for this row, and as a result, if the index was used, the row would not appear in the result, even though it does qualify. The same is true for indexes midx2 and midx3. On the other hand, even if idx1 was created WITH NO NULLS, it would still be applicable, because it indexes a single field (income) and the query contains 2 start/stop predicates on that field.

Example 9-2 Using Indexes for Query Optimization

```
SELECT * FROM Users2
WHERE 20 < income OR income < 10;
```

The query contains 1 index predicate, which is the whole WHERE expression. Indexes idx1, idx2, midx2, midx3 are all applicable. For all of them, the predicate is a filtering predicate.

Example 9-3 Using Indexes for Query Optimization

```
SELECT * FROM Users2
WHERE 20 < income OR age > 70;
```

There is no index predicate in this case, because no index has information about user ages.

Example 9-4 Using Indexes for Query Optimization

```
SELECT * FROM Users2 u
WHERE u.address.state = "CA"
    AND u.address.city = "San Jose";
```

Only idx2 is applicable. There are 2 index predicates, both of which serve as both start and stop predicates.

Example 9-5 Using Indexes for Query Optimization

```
SELECT id, 2*income FROM Users2 u
WHERE u.address.state = "CA"
    AND u.address.city = "San Jose";
```

Only idx2 is applicable. There are 2 index predicates, both of which serve as both start and stop predicates. In this case, the id and income information needed in the SELECT clause is available in the index. As a result, the whole query can be answered from the index only, with no access to the table. We say that index idx2 is a covering index for the query in Example 5. The query processor will apply this optimization.

Example 9-6 Using Indexes for Query Optimization

```
SELECT * FROM Users2 u
WHERE u.address.state = "CA"
    AND u.address.city = "San Jose"
    AND u.income > 10;
```

idx1, idx2, midx2, and midx3 are applicable. For idx2, there are 3 index predicates: the state and city predicates serve as both start and stop predicates; the income predicate is a start predicate. For idx1 only the income predicate is applicable, as a start predicate. For midx2 and midx3, the income predicate is a filtering one.

Example 9-7 Using Indexes for Query Optimization

```
SELECT * FROM Users2 u
WHERE u.address.state = "CA"
    AND u.income > 10;
```

idx1, idx2, midx2, and midx3 are applicable. For idx2, there are 2 index predicates: the state predicate serves as both start and stop predicate; the income predicate is a filtering predicate. The income predicate is a start predicate for idx1 and a filtering predicate for midx2 and midx3.

Example 9-8 Using Indexes for Query Optimization

```
DELCARE $city STRING;

SELECT * FROM Users2 u
WHERE u.address.state = "CA"
    AND u.address.city = $city
    AND (u.income > 50
        OR (10 < u.income
            AND u.income < 20));
```

idx1, idx2, midx2, and midx3 are applicable. For idx2, there are 3 index predicates. The state and city predicates serve as both start and stop predicates. The composite income predicate is a filtering predicate for all the applicable indexes (it's rooted at an OR node).

Example 9-9 Using Indexes for Query Optimization

```
SELECT u.address.city, SUM(u.expenses.values())
    FROM Users2 u
    WHERE u.address.state = "CA"
    GROUP BY u.address.city
    ORDER BY SUM(u.expenses.values());
```

In this example, for each city in California, the total amount of user expenditures in that city is returned. The query orders the results by the total amount. Only idx2 is applicable. The state predicate is both a stop and a start predicate. Furthermore, the index is a sorting index, because for any given state it sorts the table rows by the names of the cities in that state and the GROUP BY groups by the cities in CA. As a result, the grouping in this query will be index-based and the ORDER BY will be generic. Notice that if instead of idx2 there were 2 separate indexes, one on states and another on cities, both would be applicable: the first because of the state predicate, and the second because of the grouping. In this case, the query processor would choose the state index in order to reduce the number of rows accessed, at the expense of doing a generic GROUP BY.

Example 9-10 Using Indexes for Query Optimization

```
SELECT id FROM Users3 u
WHERE EXISTS u.info.income;
```

In this example we use table Users3, which stores all information about users as json data. The query looks for users who record their income. Index jidx1 is applicable. The EXISTS condition is actually converted to 2 index start/stop conditions: u.info.income < EMPTY and u.info.income > EMPTY. As a result, two range scans are performed on the index.

Example 9-11 Using Indexes for Query Optimization

```
SELECT * FROM users2 u
    WHERE (u.address.state, u.address.city) IN
        (("CA","San Jose"), ("MA","Boston"))
```

In this example, the idx2 index will be used. Two scans will be performed on the index: one for entries whose state and city fields are "CA" and "San Jose", respectively, and another for entries whose state and city fields are "MA" and "Boston", respectively.

Example 9-12   Using Indexes for Query Optimization

```
SELECT * FROM users2 u
    WHERE u.address.state in ("CA", "MA") AND
        u.address.city in ("San Jose","Boston")
```

In this example, the idx2 index will be used. Four scans will be performed on the index. The search keys for these scans are determined by the cartesian product of the keys in the right-hand-side of the two IN operators: ("CA", "San Jose"), ("CA", "Boston"), ("MA, "San Jose"), and ("MA", "Boston").

Example 9-13   Using Indexes for Query Optimization

```
SELECT * FROM users2 u
    WHERE (u.address.state, u.income) IN
        (("CA", 10000), ("MA", 20000))
```

In this example, the idx2 index will be used. Two scans will be performed on the index: one for entries whose state field is "CA", and another for entries whose state field is "MA". Furthermore, the whole IN condition will be used as a filtering predicate on the entries returned by the two scans.

As the above examples indicate, a predicate will be used as a start/stop predicate for an index IDX only if:

- It is of the form <path expr> op <const expr>, or <const expr> op <path expr>, or (<path expr1>, … <path exprN>) IN (<const exprs>)
- op is a comparison operator (EXISTS, NOT EXISTS, IS NULL and IS NOT NULL are converted to predicates of this form, as shown in Q9).
- <const expr> is an expression built from literals and external variables only (does not reference any tables or internal variables)
- <path expr> is a path expression that is "matches" an index path P appearing in the CREATE INDEX statement for IDX. So far we have seen examples of exact matches only. In the examples below we will see some non-exact matches as well.
- If P is not IDX's 1st index path, there are equality start/stop predicates for each index path appearing before P in IDX's definition.
- The comparison operator may be one of the "any" operators. Such operators are matched against the multi-key index paths of multi-key indexes. As shown in the examples below, additional restrictions apply for such predicates.

Example 9-14   Using Indexes for Query Optimization

```
SELECT * FROM users2 u
WHERE u.connections[] = any 10;
```

midx1 is applicable and the predicate is both a start and a stop predicate.

Example 9-15   Using Indexes for Query Optimization

```
SELECT * FROM users2 u
WHERE u.connections[0:4] = any 10;
```

midx1 is applicable. The predicate to push down to mdx1 is u.connections[] =any 10, in order to eliminate users who are not connected at all with user 10. However, the original predicate (u.connections[0:4] =any 10) must be retained in the query to eliminate users who do have a connection with user 10, but not among their 5 strongest connections. This is an example where the query path expression does not match exactly the corresponding index path.

Example 9-16 Using Indexes for Query Optimization

```
SELECT * FROM users2 u
WHERE u.connections[] > any 10;
```

midx1 is applicable and the predicate is a start predicate.

Example 9-17 Using Indexes for Query Optimization

```
SELECT id FROM users2 u
WHERE 10 < any u.connections[]
    AND u.connections[] < any 100 ;
```

midx1 is applicable, but although each predicate by itself is an index predicate, only one of them can actually be used as such. To see why, first notice that the query asks for users that have a connection with id greater than 10 and another connection (which may or may not be the same as the 1st one) with id less than 100. Next, consider a Users2 table with only 2 users (say with ids 200 and 500) having the following connections arrays respectively: [1, 3, 110, 120] and [1, 50, 130]. Both of these arrays satisfy the predicates in the query, and both users should be returned as a result. Now, consider midx1; it contains the following 7 entries:

[1, 200], [1, 500], [3, 200], [50, 500], [110, 200], [120, 200], [130, 500]

By using only the 1st predicate as a start predicate to scan the index, and applying the 2nd predicate on the rows returned by the index scan, the result of the query is 500, 200, which is correct. If on the other hand both predicates were used for the index scan, only entry [50, 500] would qualify, and the query would return only user 500.

Example 9-18 Using Indexes for Query Optimization

To search for users who have a connection in the range between 10 and 100, the following query can be used:

```
SELECT id FROM users2 u
WHERE exist u.connections
    [10 < $element AND $element < 100];
```

Assuming the same 2 users as in Example 13, the result of this query is user 500 only and both predicates can be used as index predicates (start and stop), because both predicates apply to the same array element. The query processor will indeed push both predicates to mdx1.

Example 9-19 Using Indexes for Query Optimization

```
SELECT * FROM Users2 u
WHERE u.address.phones.area = any 650
```

```
    AND u.address.phones.kind = any "work"
    AND u.income > 10;
```

This query looks for users whose income is greater than 10, and have a phone number with area code 650, and also have a work phone number (whose area code may not be 650). Index midx3 is applicable, but the address.phones.kind predicate cannot be used as an index predicate (for the same reason as in Example 13). Only the area code predicate can be used as a start/stop predicate and the income predicate as a filtering one. Indexes idx1, idx2, and midx2 are also applicable in Example 15.

Example 9-20 Using Indexes for Query Optimization

```
SELECT * FROM Users2 u
WHERE u.expenses.housing = 10000;
```

idx4 is applicable and the predicate is both a start and a stop predicate. midx4 is also applicable. To use midx4, two predicates must be pushed to it, even though only one appears in the query. The 1st predicate is on the "keys" index field and the second on the "values" field. Specifically, the predicates key = "price" and value = 10000 are pushed as start/stop predicates. This is another example where the match between the query path expression and an index path is not exact: we match expenses.housing with the expenses.values() index path, and additionally, generate an index predicate for the properties.keys() index path.

Example 9-21 Using Indexes for Query Optimization

```
SELECT * FROM Users2 u
WHERE u.expenses.travel = 1000
    AND u.expenses.clothes > 500;
```

midx4 is applicable. Each of the query predicates is by itself an index predicate and can be pushed to midx4 the same way as the expenses.housing predicate in the previous example. However, the query predicates cannot be both pushed (at least not in the current implementation). The query processor has to choose one of them to push and the other will remain in the query. Because the expenses.travel predicate is an equality one, it's more selective than the greater-than predicate and the query processor will use that.

Choosing the Best Applicable Index

To choose an index for a query, the query processor uses a simple heuristic together with any user-provided index hints.

Syntax

```
hints ::= '/*+' hint* '*/'

hint ::= (
   (PREFER_INDEXES "(" name_path index_name* ")") |
   (FORCE_INDEX "(" name_path index_name ")") |
   (PREFER_PRIMARY_INDEX "(" name_path ")") |
   (FORCE_PRIMARY_INDEX "(" name_path ")")
   ) [STRING]
```

There are 2 kinds of hints: a FORCE_INDEX hint and a PREFER_INDEXES hint. The FORCE_INDEX hint specifies a single index and the query is going to use that index without considering any of the other indexes (even if there are no index predicates for the forced index). The PREFER_INDEXES hint specifies one or more indexes. The query processor may or may not use one of the preferred indexes. Specifically, in the absence of a forced index, index selection works as follows.

The query processor uses the heuristic to assign a score to each applicable index and then chooses the one with the highest score. If two or more indexes have the same score, the index chosen is the one whose name is alphabetically before the others. In general, preferred indexes will get high scores, but it is possible that other indexes may still win. Describing the details of the heuristic is beyond the scope of this document, but a few high-level decisions are worth mentioning:

- If the query has a complete primary key, the primary index is used.
- Indexes that are preferred (via a PREFER hint), covering, or have a complete key (i.e., there is an equality predicate on each of its index fields) get high stores and will normally prevail over other indexes.
- Among 2 indexes where one is a sorting index, the other is not, and the 2 indexes would otherwise have the same score, the sorting index is chosen.

The FORCE_INDEX and PREFER_INDEXES hints specify indexes by their name. Since the primary index has no explicit name, 2 more hints are available to force or to prefer the primary index: FORCE_PRIMARY_INDEX and PREFER_PRIMARY_INDEX. Hints are inserted in the query as a special kind of comment that appears immediately after the SELECT keyword. Here is the relevant syntax:

The '+' character immediately after (with no spaces) the comment opening sequence ('/*') is what turns the comment into a hint. The string at the end of the hint is just for informational purposes (a comment for the hint) and does not play any role in the query execution.

10

GeoJson Data Management

This chapter describes GeoJson data and how to search and index GeoJson data in Oracle NoSQL Database. Support for GeoJson data is available only in the Enterprise Edition of Oracle NoSQL Database.

This chapter contains the following topics:

About GeoJson Data

The GeoJson specification (Internet Engineering Task Force) defines the structure and content of json objects that are supposed to represent geographical shapes on earth (called geometries). Oracle NoSQL Database implements a number of functions that do indeed interpret such json objects as geometries and allow for the search for rows containing geometries that satisfy certain conditions. Search is made efficient via the use of special indexes.

According to the GeoJson specification, for a json object to be a geometry object it must have two fields called "type" and "coordinates", where the value of the "type" field specifies the kind of geometry and the value of "coordinates" must be an array whose elements define the geometrical shape (the GeometryCollection kind is an exception to this rule, as we will see below). The value of the "type" field must be one of the following 7 strings, corresponding to 7 different kinds of geometry objects: "Point", "LineSegment", "Polygon", "MultiPoint", "MultiLineString", "MultiPolygon", and "GeometryCollection". The value of "coordinates" depends on the kind of geometry, but in all cases it is composed of a number of positions. A position specifies a position on the surface of the earth as an array of 2 double numbers, where the first number is the longitude and the second number is the latitude of the position (GeoJson allows the position’s altitude as a 3rd coordinate, but Oracle NoSQL Database does not support altitudes). Longitude and latitude are specified as degrees and must range between -180 to +180 and -90 to +90, respectively.

The 7 kinds of geometry objects are defined as follows: (with an example given in each case)

Point
For type "Point", the "coordinates" field is a single position.

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

LineString
A LineString is one or more connected lines; the end-point of one line is the start-point of the next line. The "coordinates" member is an array of two or more positions: the 1st position is the start point of the 1st line and each subsequent position is the end point of the current line and the start of the next line. Lines may cross each other.

```
{
"type" : "LineString",
"coordinates" : [ [121.9447, 37.2975],
[121.9500, 37.3171],

[121.9892, 37.3182],
[122.1554, 37.3882],
[122.2899, 37.4589],
[122.4273, 37.6032],
[122.4304, 37.6267],
[122.3975, 37.6144]
]
}
```

Polygon
A polygon defines a surface area by specifying its outer perimeter and the perimeters of any potential holes inside the area. More precisely, a polygon consists of one or more linear rings, where (a) a linear ring is a closed LineString with four or more positions, (b) the first and last positions are equivalent, and they must contain identical values, (c) a linear ring is the boundary of a surface or the boundary of a hole in a surface, and (d) a linear ring must follow the right-hand rule with respect to the area it bounds, i.e., for exterior rings their positions must be ordered counterclockwise, and for holes their position must be ordered clockwise. Then, the "coordinates" field of a polygon must be an array of linear ring coordinate arrays, where the first must be the exterior ring, and any others must be interior rings. The exterior ring bounds the surface, and the interior rings (if present) bound holes within the surface. The example below shows a polygon with no holes.

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

MultiPoint
For type "MultiPoint", the "coordinates" field is an array of two or more positions.

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

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

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

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

```
{
"type": "MultiPolygon",
"coordinates": [
[
[
[102.0, 2.0],
[103.0, 2.0],
[103.0, 3.0],
[102.0, 3.0],
[102.0, 2.0]
]
],
[
[
[100.0, 0.0],
[101.0, 0.0],
[101.0, 1.0],
[100.0, 1.0],
[100.0, 0.0]
]
]
]
}
```

GeometryCollection
Instead of a "coordinates" field, a GeometryCollection has a "geometries" field. The value of "geometries" is an array. Each element of this array is a GeoJSON object whose kind is one of the 6 kinds defined earlier. So, in general, a GeometryCollection is a heterogeneous composition of geometries.

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

The GeoJson specification defines 2 additional kinds of entities, called Feature and FeatureCollection, which allow for combining geometries with other, non-geometrical properties. The specification uses defined above) or a Feature or a FeatureCollection. Feature and FeatureCollection are defined as follows:

Feature
A Feature object has a "type" member with the value "Feature". A Feature object has a "geometry" member, whose value either a geometry object of the 7 kinds defined above or the JSON null value. A Feature object has a "properties" member, whose value is any JSON object or the JSON null value.

FeatureCollection
A FeatureCollection object has a "type" member with the value "FeatureCollection". A FeatureCollection object has a "features" member, whose value is a JSON array. Each element of the array is a Feature object as defined above. It is possible for this array to be empty.

Lines and Coordinate System

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

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

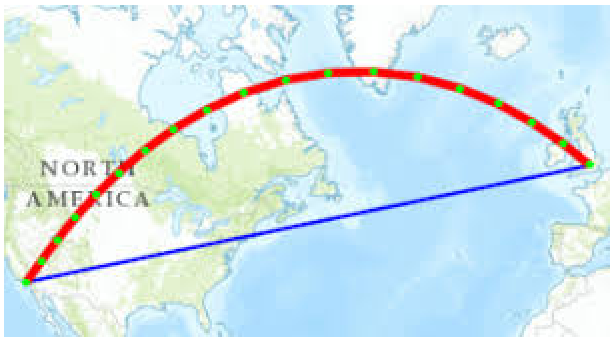
with t being a real number greater than or equal to 0 and smaller than or equal to 1.

However, Oracle NoSQL Database uses a geodetic coordinate system (WGS 84) and as a result deviates from the GeoJson specification by using **geodetic lines**: A geodetic line between 2 points is the shortest line that can be drawn between the

2 points on the ellipsoidal surface of the earth. For a simplified, but more illustrative definition, assume for a moment that the earth surface is a sphere. Then, the geodetic line between two points on earth is the minor arc between the two points on the **great circle** corresponding to the points, i.e., the circle that is formed by the intersection of the sphere and the plane defined by the center of the earth and the two points.

The following figure shows the difference between the geodetic and straight lines between Los Angeles and London.

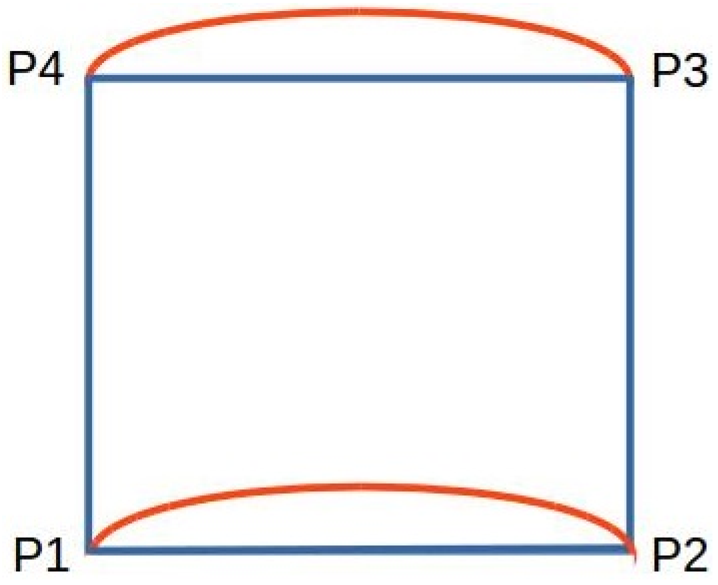
Figure 10-1 Geodetic vs Straight Line



(source: https://developers.arcgis.com)

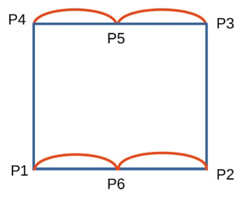
The following figure shows the difference between the two coordinate systems for a square defined by points P1, P2, P3, and P4. The square is assumed to be in the northern hemisphere. The 2 vertical lines of the square are the same in both systems; points on each of these lines have the same longitude. This is not true for the "horizontal" lines. In the GeoJson system all points on the P1-P2 line (the blue line) have the same latitude (so this line is part of an earth parallel). But the geodetic line between P1 and P2 forms a curve (the red line) to the north of the GeoJson line. The difference between the two lines (the curvature) gets more pronounced closer to the poles and as the distance between P1 and P2 increases.

Figure 10-2 Geodetic vs GeoJson Box



When searching for points or other geometries inside the [P1, P2, P3, P4] polygon (using one of the functions described in the next section), Oracle NoSQL Database uses the geodetic view of the search polygon. So, for example, points that are between the blue and the red P1-P2 lines will not appear in the result. What if you really want to search inside the blue box? Such a search can be approximated by adding points between P1-P2 and P4-P3 in the definition of the search polygon. This is illustrated in the following figure, where we have added points P5 and P6. We can see that with the [P1, P6, P2, P3, P5, P4] polygon, the area difference between the geodetic and GeoJson boxes is smaller than with the [P1, P2, P3, P4] polygon.

Figure 10-3   Approximating a Search within a GeoJson Box



Restrictions on GeoJson Data

The following 2 restrictions apply to the kind of GeoJson data supported by Oracle NoSQL Database:

Anti-meridian crossing
Geometries that cross the anti-meridian line cannot be indexed and cannot appear as arguments to the geo search functions described in the following section.

Too big geometries
A geometry is considered "too big" if its Minimum Bounding Box (MBR) has a side whose end points are more than 120 degrees or latitude or longitude apart. Such geometries cannot be indexed and cannot appear as arguments to the geo search functions described in the following section.

Searching for GeoJson Data

Oracle NoSQL Database provides 4 functions to search for GeoJson data that have a certain relationship with a search geometry.

boolean geo_intersect(any*, any*)

Raises an error if it can be detected at compile time that an operand will not return a single valid geometry object. Otherwise, the runtime behavior is as follows:

- Returns false if any operand returns 0 or more than 1 items.
- Returns NULL if any operand returns NULL.
- Returns false if any operand returns an item that is not a valid geometry object.

Finally, if both operands return a single geometry object, it returns true if the 2 geometries have any points in common; otherwise false.

boolean geo_inside(any*, any*)

Raises an error if it can be detected at compile time that an operand will not return a single valid geometry object. Otherwise, the runtime behavior is as follows:

- Returns false if any operand returns 0 or more than 1 items.
- Returns NULL if any operand returns NULL.
- Returns false if any operand returns an item that is not a valid geometry object (however, if it can be detected at compile time that an operand will not return a valid geometry, an error is raised).
- Returns false if the second operand returns a geometry object that is not a polygon.

Finally, if both operands return a single geometry object and the second geometry is a polygon, it returns true if the first geometry is completely contained inside the second polygon, i.e., all its points belong to the interior of the polygon; otherwise false. The interior of a polygon is all the points in the polygon area except the points on the linear rings that define the polygon's boundary.

boolean geo_within_distance(any*, any*, double)

Raises an error if it can be detected at compile time that any of the first two operands will not return a single valid geometry object. Otherwise, the runtime behavior is as follows:

- Returns false if any of the first two operands returns 0 or more than 1 items.
- Returns NULL if any of the first two operands returns NULL.
- Returns false if any of the first two operands returns an item that is not a valid geometry object.

Finally, if both of the first two operands return a single geometry object, it returns true if the first geometry is within a distance of N meters from the second geometry, where N is the number returned by the third operand; otherwise false. The distance between 2 geometries is defined as the minimum among the distances of any pair of points where the first point belongs to the first geometry and the second point to the second geometry. If N is a negative number, it is set to 0.

boolean geo_near(any*, any*, double)

geo_near is converted internally to geo_within_distance plus an (implicit) order-by the distance between the two geometries. However, if the query has an (explicit) order-by already, no ordering by distance is performed. The geo_near function can appear in the WHERE clause only, where it must be a top-level predicate, i.e, not nested under an OR or NOT operator.

In addition to the above search functions, the following two functions are also provided:

double geo_distance(any*, any*)

Raises an error if it can be detected at compile time that an operand will not return a single valid geometry object. Otherwise, the runtime behavior is as follows:

- Returns -1 if any operand returns zero or more than 1 items.
- Returns NULL if any operand returns NULL.

- Returns -1 if any of the operands is not a geometry.

Otherwise it returns the geodetic distance between the 2 input geometries. The returned distance is the minimum among the distances of any pair of points where the first point belongs to the first geometry and the second point to the second geometry. Between two such points, their distance is the length of the geodetic line that connects the points.

boolean geo_is_geometry(any*)

- Returns false if the operand returns zero or more than 1 items.
- Returns NULL if the operand returns NULL.
- Returns true if the input is a single valid geometry object. Otherwise, false.

Notice that the above geo functions operate on geometry objects, but not on Features or FeatureCollections. Nevertheless, Features and FeatureCollections can still be queried effectively by passing their contained geometry objects to the geo function. An example of this is shown in the following section.

Example 10-1 Searching for GeoJson Data

Consider a table whose rows store points of interest. The table has an id column as its primary key and a poi column of type json.

```
CREATE TABLE PointsOfInterest (
    id INTEGER, poi JSON,
PRIMARY KEY(id));

INSERT INTO PointsOfInterest VALUES (
    1,
    {
        "kind" : "city hall",
        "address" : {
            "state" : "CA",
            "city" : "Campbell",
            "street" : "70 North 1st street"
        },
        "location" : {
            "type" : "point",
            "coordinates" : [121.94,37.29]
        }
    }
);
INSERT INTO PointsOfInterest VALUES (
    2,
    {
        "kind" : "nature park",
        "name" : "castle rock state park",
        "address" : {
            "state" : "CA",
            "city" : "Los Gatos",
            "street" : "15000 Skyline Blvd"
        },
        "location" : {
            "type" : "polygon",
            "coordinates" : [
                [
                    [122.1301, 37.2330],
                    [122.1136, 37.2256],
                    [122.0920, 37.2291],
                    [122.1020, 37.2347],
                    [122.1217, 37.2380],
                    [122.1301, 37.2330]
                ]
            ]
        }
    }
);
```

The following query looks for nature parks in northern California. The query uses geo_intersect, instead of geo_inside, to include parks that straddle the border with neighbor states.

```
SELECT t.poi AS park
FROM PointsOfInterest t
WHERE t.poi.kind = "nature park"
AND
geo_intersect(
    t.poi.location,
    {
        "type" : "polygon",
        "coordinates" : [
            [
                [121.94, 36.28],
                [117.52, 37.38],
                [119.99, 39.00],
                [120.00, 41.97],
                [124.21, 41.97],
                [124.39, 40.42],
                [121.94, 36.28]
            ]
        ]
    }
);
```

The following query looks for gas stations within a mile of a given route. The returned gas stations are ordered by ascending distance from the route.

```
SELECT
t.poi AS gas_station,
geo_distance(
    t.poi.location,
    {
        "type" : "LineString",
        "coordinates" : [
            [121.9447, 37.2975],
            [121.9500, 37.3171],
            [121.9892, 37.3182],
            [122.1554, 37.3882],
            [122.2899, 37.4589],
                [122.4273, 37.6032],
                [122.4304, 37.6267],
                [122.3975, 37.6144]
            ]
        }
) AS distance
FROM PointsOfInterest t
WHERE t.poi.kind = "gas station"
AND
geo_near(
    t.poi.location,
    {
        "type" : "LineString",
        "coordinates" : [
            [121.9447, 37.2975],
            [121.9500, 37.3171],
            [121.9892, 37.3182],
            [122.1554, 37.3882],
            [122.2899, 37.4589],
            [122.4273, 37.6032],
            [122.4304, 37.6267],
            [122.3975, 37.6144]
        ]
    },
    1609
);
```

Example 10-2   Searching for GeoJson data

This example shows how FeatureCollections can be queried in Oracle NoSQL Database. Consider a "companies" table that stores info about companies, including the locations where each company has offices and some properties for each office location.

```
CREATE TABLE companies (
    id INTEGER, info JSON, PRIMARY KEY(id));

INSERT INTO companies VALUES (
    1,
    {
        "id" : 1,
        "info" : {
            "name" : "acme",
            "CEO" : "some random person",
            "locations" : {
                "type" : "FeatureCollection",
                "features" : [
                    {
                        "type" : "Feature",
                        "geometry" : {
                            "type" : "point",
                            "coordinates" : [ 23.549, 35.2908 ]
                        },
                        "properties" : {
                            "kind" : "development",
                            "city" : "palo alto"
                        }
                    },
                    {
                        "type" : "Feature",
                        "geometry" : {
                            "type" : "point",
                            "coordinates" : [ 23.9, 35.17 ]
                        },
                        "properties" : {
                            "kind" : "sales",
                            "city" : "san jose"
                        }
                    }
                ]
            }
        }
    }
);
```

The following query looks for companies that have sales offices within a search region and returns, for each such company, an array containing the geo-locations of the sales offices within the same search region.

```
SELECT id,
c.info.locations.features [
    geo_intersect(
        $element.geometry,
        {
            "type" : "polygon",
            "coordinates" : [
                [
                    [23.48, 35.16],
                    [24.30, 35.16],
                    [24.30, 35.70],
                    [23.48, 35.70],
                    [23.48, 35.16]
                ]
            ]
        }
    )
    AND
    $element.properties.kind = "sales"
].geometry AS loc
FROM companies c
WHERE EXISTS c.info.locations.features [
    geo_intersect(
        $element.geometry,
        {
            "type" : "polygon",
            "coordinates" : [
                [
                    [23.48, 35.16],
                    [24.30, 35.16],
                        [24.30, 35.70],
                        [23.48, 35.70],
                        [23.48, 35.16]
                    ]
                ]
            }
        )
        AND
        $element.properties.kind = "sales"
    ] ;
```

For efficient execution of this query, the following index can be created:

```
CREATE INDEX idx_kind_loc ON companies (
info.locations.features[].properties.kind AS STRING,
info.locations.features[].geometry AS POINT);
```

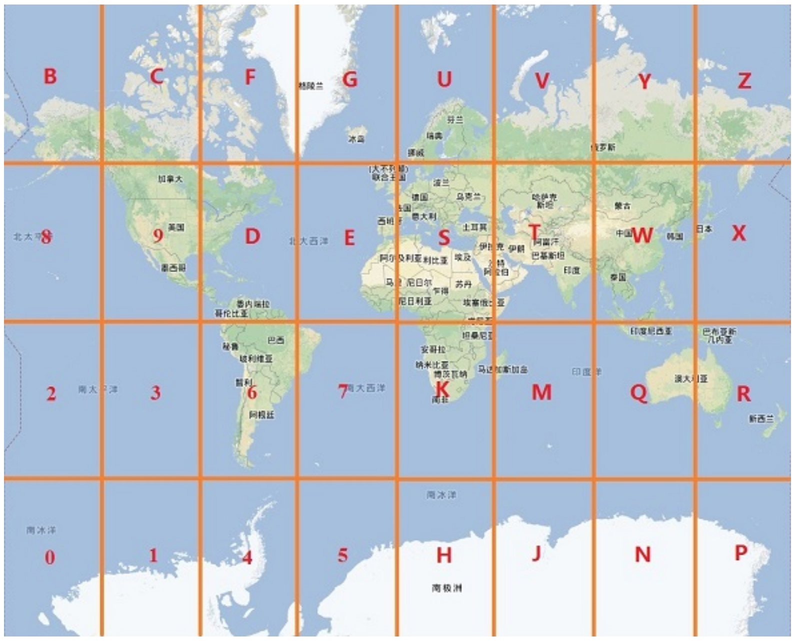
Indexing GeoJson Data

Indexing GeoJson data is similar to indexing other json data. In the GeoJson case, the GEOMETRY or POINT keyword must be used after an index path that leads to geometry objects. POINT should be used only if all rows in the table are expected to have single point geometries at the indexed field (GEOMETRY can also be used in this case, but POINT is recommended for better performance). As in the case of other json data, an error will be raised if for some row the value of the index path is not a valid GeoJson point or geometry, unless that value is NULL, json null, or EMPTY.

An index that includes a path to geometry objects is called a geometry index. A geometry index can index other fields as well, but some restrictions apply: (a) a geometry index cannot index more than one GeoJson field, (b) the GeoJson field cannot be inside an array, unless it is a POINT field, and (c) a geometry index cannot be a multi-key index, unless the GeoJson field is a POINT field and the array or map being indexed is the one that contains the POINT field.

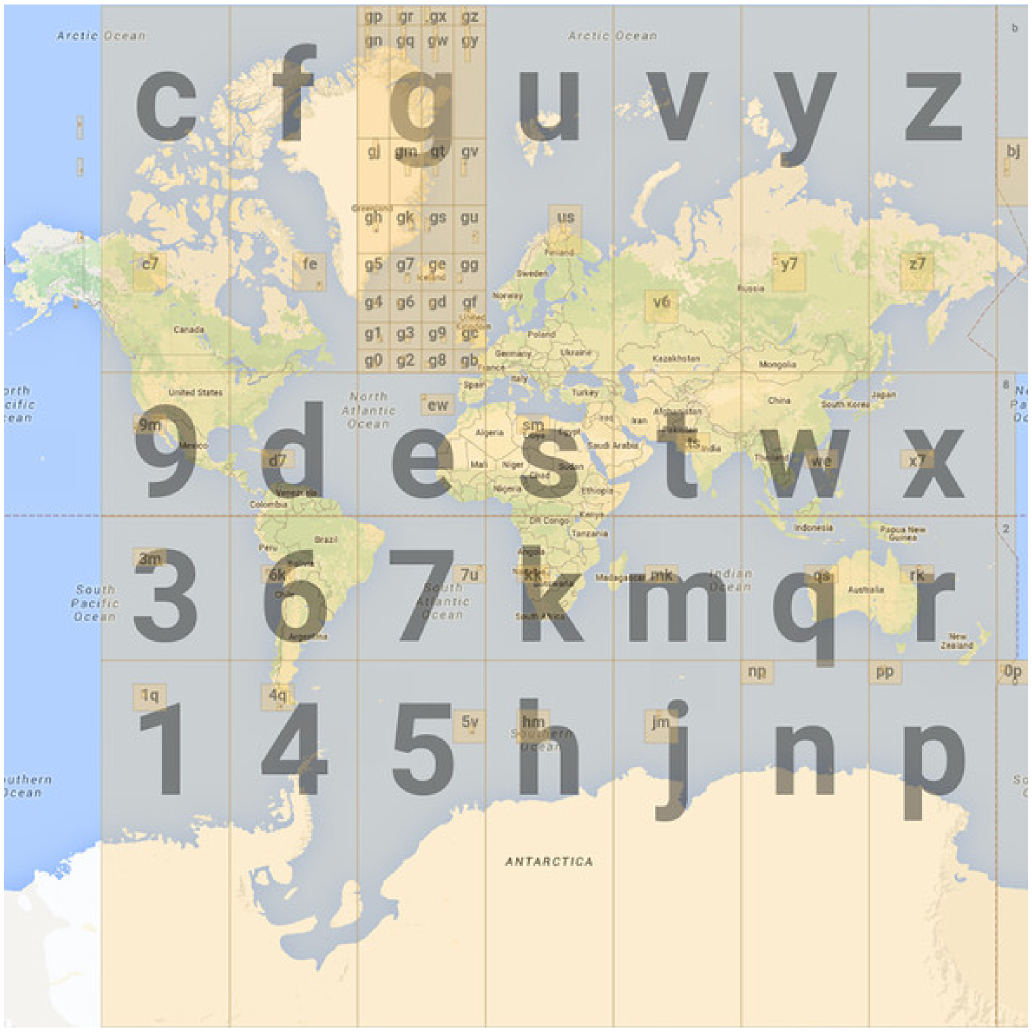
Indexing of geometries is based on geohashing. Geohashing is an encoding of a longitude/latitude pair to a string. It works by recursively partitioning the 2-D longitude/latitude coordinate system into a hierarchy of rectangulars called cells. The initial (level-0) cell is the whole world, i.e., all points with a longitude between -180 and +180 and latitude between -90 and +90. The first (level-0) split creates the 32 level-1 cells shown in the following figure. Each cell is assigned a "name", which is a character out of this 32-char-long string G = "0123456789bcdefghjkmnpqrstuvwxyz". This name is called the geohash of the cell.

Figure 10-4   32 Level-1 Geohash Cells



The next (level-1) split splits each level-1 cell into 32 level-2 cells. The following figure shows all the level-2 cells inside the "g" cell.

Figure 10-5 Level-1 and Level-2 Geohash Cells



As shown, the geohash of each level-2 is 2 chars long, where the 1st char is the geohash of the parent cell, and the 2nd char is again drawn from the same char set. This process continues down to some given level L, called the geohash length. During an even-numbered split, each cell is split into 8 vertical slices and 4 horizontal slices. During an odd-numbered split, each cell is split into 4 vertical slices and 8 horizontal slices. In both cases, for each of the 32 sub-cells, its geohash is formed by using the parent-cell geohash as a prefix and appending a char out of G. The extra char for each sub-cell is chosen the same way as shown in both the earlier figures for even and odd splits respectively.

Oracle NoSQL Database uses a geohash length of 10. Cells at level 10 have an area of about 1 square meter. When indexing a point, the level-10 cell that contains the point is computed and the geohash of that cell is placed in the index entry. So, for points, a single index entry is generated for each point, and a geometry index on a POINT field behaves like a simple (non-multikey) index, unless the POINT field itself is inside an array or map that is being indexed. Notice that all points inside the same level-10 cell will have the same geohash.

With the geohashing algorithm described above, points that are close to each other will usually (but not always) be close together in the geometry index as well, i.e., have long common prefixes. So, searching for points using one of the functions described in the previous section translates to one or more range scans in the geometry index. These range scans may return false positives, so the search function itself must still be applied on the rows returned by index scans to eliminate the false positives.

When indexing a LineString or Polygon, the geometry’s minimum bounding box (MBR) is computed first, and then a set of cells is found that completely cover the MBR. The level of the covering cells depends on the size and shape and position of the MBR (usually it will be less than 10). Then, an index entry is created for each of the covering cells containing the geohash of that cell. So, a geometry index on a LineString or Polygon is always a multi-key index since multiple index entries will be created for a single row. For MultiPoints, MultiLineStrings, MultiPolygons, and GeometryCollections each of the constituent geometries is indexed separately and the index for such geometries it also a multi-key index.

11

Built-in Functions

This chapter discusses about Built-in functions supported in Oracle NoSQL Database.

This chapter contains the following topics:

Functions on Complex Values

integer? size(any?)

Returns the number of fields/entries of a complex item (array, map, record). Although the parameter type appears as ANY?, the function will actually raise an error if the given item is not complex. The function accepts an empty sequence as argument, in which case it will return the empty sequence. The function will return NULL if its input is NULL.

Functions on Sequences

any* seq_concat(any*, ...)

seq_concat is a variadic function: it can have any number of arguments. It simply evaluates its arguments (if any) in the order they are listed in the argument list, and concatenates the sequences returned by these arguments.

In addition to the above there are also the following aggregate functions on sequences. They are described in the Sequence Aggregate Functions section.

- long seq_count(any*)
- number seq_sum(any*)
- number seq_avg(any*)
- any_atomic seq_min(any*)
- any_atomic seq_max(any*)

Functions on Timestamps

integer? year(timestamp?)

Returns the year for the given timestamp. The returned value is in the range -6383 to 9999. If the argument is NULL or empty, the result is also NULL or empty.

integer? month(timestamp?)

Returns the month for the given timestamp, in the range 1 ~ 12. If the argument is NULL or empty, the result is also NULL or empty.

integer? day(timestamp?)

Returns the day of month for the timestamp, in the range 1 ~ 31. If the argument is NULL or empty, the result is also NULL or empty.

integer? hour(timestamp?)

Returns the hour of day for the timestamp, in the range 0 ~ 23. If the argument is NULL or empty, the result is also NULL or empty.

integer? minute(timestamp?)

Returns the minute for the timestamp, in the range 0 ~ 59. If the argument is NULL or empty, the result is also NULL or empty.

integer? second(timestamp?)

Returns the second for the timestamp, in the range 0 ~ 59. If the argument is NULL or empty, the result is also NULL or empty.

integer? millisecond(timestamp?)

Returns the fractional second in millisecond for the timestamp, in the range 0 ~ 999. If the argument is NULL or empty, the result is also NULL or empty.

integer? microsecond(timestamp?)

Returns the fractional second in microsecond for the timestamp, in the range 0 ~ 999999. If the argument is NULL or empty, the result is also NULL or empty.

integer? nanosecond(timestamp?)

Returns the fractional second in nanosecond for the timestamp, in the range 0 ~ 999999999. If the argument is NULL or empty, the result is also NULL or empty.

integer? week(timestamp?)

Returns the week number within the year where a week starts on Sunday and the first week has a minimum of 1 day in this year, in the range 1 ~ 54. If the argument is NULL or empty, the result is also NULL or empty.

integer? isoweek(timestamp?)

Returns the week number within the year based on IS0-8601, where a week starts on Monday and the first week has a minimum of 4 days in this year, in range 0 ~ 53. If the argument is NULL or empty, the result is also NULL or empty.

long current_time_millis()

Returns the current time in UTC, as the number of milliseconds since January 1, 1970 UTC.

timestamp(3) current_time()

Returns the current time in UTC, as a timestamp value with millisecond precision.

Functions on Rows

As described in the Table Management section, table rows are record values conforming to the table schema, but with some additional properties that are not part of the table schema. To extract the value of such properties, the functions listed in this sub-section must be used.

Although the signature of these functions specifies AnyRecord as the type of the input parameter, the functions actually require a row as input. The only expression that returns a row is a row variable, that is, a table alias whose name starts with '$'. The Example: Updating TTL section shows an example of using the remaining_hours() function, which is one of the row available functions.

integer remaining_hours(AnyRecord)

Returns the number of full hours remaining until the row expires. If the row has no expiration time, it returns a negative number.

integer remaining_days(AnyRecord)

Returns the number of full days remaining until the row expires. If the row has no expiration time, it returns a negative number.

timestamp(0) expiration_time(AnyRecord)

Returns the expiration time of the row, as a timestamp value of precision zero. If the row has no expiration time, it returns a timestamp set on January 1, 1970 UTC.

long expiration_time_millis(AnyRecord)

Returns the expiration time of the row, as the number of milliseconds since January 1, 1970 UTC. If the row has no expiration time, it returns zero.

Functions on GeoJson Data

boolean geo_intersect(any*, any*)

Raises an error if it can be detected at compile time that an operand will not return a single valid geometry object. Otherwise, the runtime behavior is as follows:

- Returns false if any operand returns 0 or more than 1 items.

- Returns NULL if any operand returns NULL.
- Returns false if any operand returns an item that is not a valid geometry object.

Finally, if both operands return a single geometry object, it returns true if the 2 geometries have any points in common; otherwise false.

boolean geo_inside(any*, any*)

Raises an error if it can be detected at compile time that an operand will not return a single valid geometry object. Otherwise, the runtime behavior is as follows:

- Returns false if any operand returns 0 or more than 1 items.
- Returns NULL if any operand returns NULL.
- Returns false if any operand returns an item that is not a valid geometry object (however, if it can be detected at compile time that an operand will not return a valid geometry, an error is raised).
- Returns false if the second operand returns a geometry object that is not a polygon.

Finally, if both operands return a single geometry object and the second geometry is a polygon, it returns true if the first geometry is completely contained inside the second polygon, i.e., all its points belong to the interior of the polygon; otherwise false. The interior of a polygon is all the points in the polygon area except the points on the linear rings that define the polygon’s boundary.

boolean geo_within_distance(any*, any*, double)

Raises an error if it can be detected at compile time that any of the first two operands will not return a single valid geometry object. Otherwise, the runtime behavior is as follows:

- Returns false if any of the first two operands returns 0 or more than 1 items.
- Returns NULL if any of the first two operands returns NULL.
- Returns false if any of the first two operands returns an item that is not a valid geometry object.

Finally, if both of the first two operands return a single geometry object, it returns true if the first geometry is within a distance of N meters from the second geometry, where N is the number returned by the third operand; otherwise false. The distance between 2 geometries is defined as the minimum among the distances of any pair of points where the first point belongs to the first geometry and the second point to the second geometry. If N is a negative number, it is set to 0.

boolean geo_near(any*, any*, double)

geo_near is converted internally to geo_within_distance plus an (implicit) order-by the distance between the two geometries. However, if the query has an (explicit) order-by already, no ordering by distance is performed. The geo_near function can appear in the WHERE clause only, where it must be a top-level predicate, i.e, not nested under an OR or NOT operator.

double geo_distance(any*, any*)

Raises an error if it can be detected at compile time that an operand will not return a single valid geometry object. Otherwise, the runtime behavior is as follows:

- Returns -1 if any operand returns zero or more than 1 items.
- Returns NULL if any operand returns NULL.
- Returns -1 if any of the operands is not a geometry.

Otherwise it returns the geodetic distance between the 2 input geometries. The returned distance is the minimum among the distances of any pair of points where the first point belongs to the first geometry and the second point to the second geometry. Between two such points, their distance is the length of the geodetic line that connects the points.

boolean geo_is_geometry(any*)

- Returns false if the operand returns zero or more than 1 items.
- Returns NULL if the operand returns NULL.
- Returns true if the input is a single valid geometry object. Otherwise, false.

Functions on Strings

This section describes various functions on strings.

substring Function

The substring function extracts a string from a given string according to a given numeric starting position and a given numeric substring length.

Syntax

```
returnvalue substring (source, position [, substring_length] )

source ::= any*
position ::= integer*
substring_length ::= integer*
returnvalue ::= string
```

Semantics

source
The input string from which the substring should be extracted. This argument is implicitly cast to a sequence of strings.

position
This argument indicates the starting point of the substring within the source. The first character of the source string has position 0.
An error is thrown if a non-integer value is supplied for the position.

substring_length
This argument indicates the length of the substring starting from the position value. If the supplied value is greater than the length of the source, then the length of the source is assumed for this argument.
An error is thrown if a non-integer value is supplied for the substring_length.

returnvalue
Returns an empty string ("") if the function did not return any characters.
Returns an empty string ("") if the substring_length is less than 1.
Returns NULL if the source argument is NULL.
Returns NULL if the position argument is less than 0 or greater or equal to the source length.

Example 11-1 substring Function

In this example, the first character in the firstname is selected from the users table. Notice that to select the first character, we have provided the value 0 for the position argument.

```
SELECT substring(firstname,0,1) as Initials FROM users;


 +----------+
 | Initials |
 +----------+
 | J        |
 | P        |
 | M        |
 +----------+
```

Example 11-2 substring Function

This example illustrates that providing a negative value for the position argument will result in a NULL output value.

```
SELECT substring (firstname, -5, 4) FROM users;


 +----------+
 | Column_1 |
 +----------+
 | NULL     |
 | NULL     |
 | NULL     |
 +----------+
```

Example 11-3 substring Function

In this example, we select the first 4 characters from the firstname in the users table.

```
SELECT substring (firstname, 0, 4) FROM users;


 +----------+
 | Column_1 |
 +----------+
 | John     |
 | Pete     |
 | Mary     |
 +----------+
```

Example 11-4 substring Function

In this example, we select 100 characters starting from position 2. Notice that even though none of the rows has more than 5 characters in firstname, still we get the output up to the length of the source starting from position 2.

```
SELECT substring (firstname, 2, 100) FROM users;


 +----------+
 | Column_1 |
 +----------+
 | hn       |
 | ter      |
 | ry       |
 +----------+
```

Example 11-5 substring Function

In this example, the substring_length argument is not provided as it is optional. In such cases, we get the complete substring starting from the given position.

```
SELECT substring (firstname, 2) FROM users;


 +----------+
 | Column_1 |
 +----------+
 | hn       |
 | ter      |
 | ry       |
 +----------+
```

concat Function

Syntax

```
returnvalue concat (source,[source*])

source ::= any*
returnvalue ::= boolean
```

Semantics

The concat function returns arg1 concatenated with arg2. Both arg1 and arg2 can be of any data type.

source
The input values that are joined to get a character string. This argument is implicitly cast to a sequence of strings.

returnvalue
Returns the character string made by joining its character string operands in the order given.
If any of the arguments is a sequence, then all the items are concatenated to the result in the order they appear in the sequence.
If all the arguments are empty sequence, then an empty sequence is returned.
If all the arguments are NULL, then a NULL is returned. This is because a NULL argument is converted to an empty string during concatenation unless all arguments are NULL, in which case the result is NULL. So NULL can result only from the concatenation of two or more NULL values.

Note:

For security/denial of service reasons the maximum number of chars of the returned string will be less than STRING_MAX_SIZE = 2^18 - 1 in chars i.e. 512kb. If the number of chars exceeds this number, then a runtime query exception is thrown.

Example 11-6 concat function

This example joins id, firstname, and lastname into a single string and provides the output. Notice that id, which is an integer type, also gets concatenated with the string values.

```
SELECT concat(id, firstname, lastname) AS name FROM users;


 +-------------+
 |    name     |
 +-------------+
 | 10JohnSmith |
 | 30PeterPaul |
 | 20MaryAnn   |
 +-------------+
```

upper Function

The upper function converts all the characters in a string to uppercase.

Syntax

```
returnvalue upper (source)

source ::= any*
returnvalue ::= string
```

Semantics

source
The input string that should be converted to uppercase. This argument is implicitly cast to a sequence of strings.

returnvalue
Returns NULL if the source argument is NULL.
Returns NULL if the source argument is an empty sequence or a sequence with more than one item.

Note:

If you want to convert a sequence with more than one item, see the Sequence Transform Expressions section.

Example 11-7 upper Function

In this example, the lastname field is converted to uppercase.

```
SELECT id, firstname, upper(lastname) FROM users;


 +----+-----------+----------+
 | id | firstname | Column_3 |
 +----+-----------+----------+
 | 10 | John      | SMITH    |
 | 20 | Mary      | ANN      |
 | 30 | Peter     | PAUL     |
 +----+-----------+----------+
```

lower Function

The lower function converts all the characters in a string to lowercase.

Syntax

```
returnvalue lower (source)

source ::= any*
returnvalue ::= string
```

Semantics

source
The input string that should be converted to lowercase. This argument is implicitly cast to a sequence of strings.

returnvalue
Returns NULL if the source argument is NULL.
Returns NULL if the source argument is an empty sequence or a sequence with more than one item.

Note:

If you want to convert a sequence with more than one item, see the Sequence Transform Expressions section.

Example 11-8 lower Function

In this example, the lastname field is converted to lowercase.

```
SELECT id, firstname, lower(lastname) FROM users;


 +----+-----------+----------+
 | id | firstname | Column_3 |
 +----+-----------+----------+
 | 10 | John      | smith    |
 | 20 | Mary      | ann      |
 | 30 | Peter     | paul     |
 +----+-----------+----------+
```

trim Function

The trim function enables you to trim leading or trailing characters (or both) from a string.

Syntax

```
returnvalue trim(source [, position [, trim_character]])

source ::= any*
position ::= "leading"|"trailing"|"both"
trim_character ::= string*
returnvalue ::= string
```

Semantics

source
The input string that should be trimmed. This argument is implicitly cast to a sequence of strings.
If you provide only the source argument, then the leading and trailing blank spaces are removed.

position
This argument indicates whether leading or trailing or both leading and trailing characters should be removed. The following are the valid values that can be specified for this argument.

- If *leading* is specified, then the characters equal to the trim_character argument are removed from the beginning of the string.
- If *trailing* is specified, then the characters equal to the trim_character argument are removed at the end of the string.

- If *both* is specified, then the characters equal to the trim_character argument are removed from both the beginning and end of the string.
- If no value is specified, then *both* value is assumed.
- If any value other than the above valid values are specified, then NULL is returned.

trim_character
This argument specifies the characters that should be removed from the source string. If you do not specify this argument, then a blank space is taken as the default value.
Only one character is allowed for this argument. If there are more than one character, then the first character will be used.
If an empty string is specified, then no trimming happens.

return_value
Returns NULL if any of the arguments is NULL.
Returns NULL if any argument is an empty sequence or a sequence with more than one item.

Example 11-9 trim function

Create this table and insert values in it to run the trim, ltrim, and rtrim function examples.

```
CREATE TABLE trim_demo (
  id INTEGER,
  name STRING,
  yearofbirth STRING,
  PRIMARY KEY (id)
);

INSERT INTO trim_demo VALUES (10, "  Peter  ", 1980);
INSERT INTO trim_demo VALUES (20, "Mary", 1973);
INSERT INTO trim_demo VALUES (30, "  Oliver", 2000);
INSERT INTO trim_demo VALUES (40, "John  ", 2000);

SELECT * FROM trim_demo;


 +----+-----------+-------------+
 | id |   name    | yearofbirth |
 +----+-----------+-------------+
 | 10 |   Peter   | 1980        |
 | 20 | Mary      | 1973        |
 | 30 |   Oliver  | 2000        |
 | 40 | John      | 2000        |
 +----+-----------+-------------+
```

Example 11-10   trim Function

In this example, the id and yearofbirth are selected from the trim_demo table. Notice that the zeros at the end of the yearofbirth are removed using the trim function.

```
SELECT id, trim(yearofbirth,"trailing",'0') FROM trim_demo;


 +----+----------+
 | id | Column_2 |
 +----+----------+
 | 10 | 198      |
 | 20 | 1973     |
 | 30 | 2        |
 | 40 | 2        |
 +----+----------+
```

Example 11-11   trim Function

In this example, '19' is provided as the trim_character. However, as per semantics, only the first character '1' will be considered for trimming.

```
SELECT id, trim(yearofbirth,"leading",'19') FROM trim_demo;


 +----+----------+
 | id | Column_2 |
 +----+----------+
 | 10 | 980      |
 | 20 | 973      |
 | 30 | 2000     |
 | 40 | 2000     |
 +----+----------+
```

ltrim Function

The ltrim function enables you to trim leading characters from a string.

Syntax

```
returnvalue ltrim(source)

source ::= any*
returnvalue ::= string
```

Semantics

source
The input string that should be trimmed. The leading spaces in this string are removed. This argument is implicitly cast to a sequence of strings.

returnvalue
Returns NULL if the source argument is NULL.

Returns NULL if the source argument is an empty sequence or a sequence with more than one item.

Example 11-12 trim Function

This example demonstrates ltrim function. Notice that the empty spaces at the beginning are removed but the empty spaces at the end are not removed.

Note:

You can use JSON query output mode so that the empty spaces are visible.

```
MODE JSON

SELECT id, ltrim(name) FROM trim_demo;


{"id":10,"Column_2":"Peter  "}
{"id":20,"Column_2":"Mary"}
{"id":30,"Column_2":"Oliver"}
{"id":40,"Column_2":"John  "}
```

rtrim Function

The rtrim function enables you to trim trailing characters from a string.

Syntax

```
returnvalue rtrim(source)

source ::= any*
returnvalue ::= string
```

Semantics

source
The input string that should be trimmed. The trailing spaces in this string are removed. This argument is implicitly cast to a sequence of strings.

returnvalue
Returns NULL if the source argument is NULL.
Returns NULL if the source argument is an empty sequence or a sequence with more than one item.

Example 11-13 trim Function

This example demonstrates rtrim function. Notice that the empty spaces at the end are removed but the empty spaces at the beginning are not removed.

> **Note:**
>
> You can use JSON query output mode so that the empty spaces are visible.

```
MODE JSON

SELECT id, rtrim(name) FROM trim_demo;


{"id":10,"Column_2":"  Peter"}
{"id":20,"Column_2":"Mary"}
{"id":30,"Column_2":"  Oliver"}
{"id":40,"Column_2":"John"}
```

length Function

The length function returns the length of a character string. The length function calculates the length using the UTF character set.

Syntax

```
returnvalue length(source)

source ::= any*
returnvalue ::= integer
```

Semantics

source
The input string for which the length should be determined. This argument is implicitly cast to a sequence of strings.

returnvalue
Returns NULL if the source argument is NULL.
Returns NULL if the source argument is an empty sequence or a sequence with more than one item.

> **Note:**
>
> Characters that are represented on 32 or more bits, the length is considered 1, while Java String.length() returns 2 for UTF32 chars, 4 for UTF64, etc.

Example 11-14 length Function

In this example, the length of the first name is selected from the users table.

```
SELECT firstname, length(firstname) as length FROM users;
```

```
 +-----------+--------+
 | firstname | length |
 +-----------+--------+
 | John      |      4 |
 | Mary      |      4 |
 | Peter     |      5 |
 +-----------+--------+
```

contains Function

The contains function indicates whether or not a search string is present inside the source string.

Syntax

```
returnvalue contains(source, search_string)

source ::= any*
search_string ::= any*
returnvalue ::= boolean
```

Semantics

source
The input string to be searched. This argument is implicitly cast to a sequence of strings.

search_string
The string that should be searched in the source. This argument is implicitly cast to a sequence of strings.

returnvalue
Returns true if search_string exists inside source else returns false.
Returns false if any argument is an empty sequence or a sequence with more than one item.
Returns NULL if source or search_string argument is NULL.

Example 11-15 contains Function

In this example, the firstname field values that contain the string "ar" in it is indicated as true.

```
SELECT firstname, contains(firstname,"ar") FROM users;
```

```
 +-----------+----------+
 | firstname | Column_2 |
 +-----------+----------+
 | John      | false    |
 | Peter     | false    |
 | Mary      | true     |
 +-----------+----------+
```

starts_with Function

The starts_with function indicates whether or not the source string begins with the search string.

Syntax

```
returnvalue starts_with(source, search_string)

source ::= any*
search_string ::= any*
returnvalue ::= boolean
```

Semantics

source
The input string to be searched. This argument is implicitly cast to a sequence of strings.

search_string
The string that should be searched in the source. This argument is implicitly cast to a sequence of strings.

returnvalue
Returns true if source begins with search_string else returns false.
Returns false if any argument is an empty sequence or a sequence with more than one item.
Returns NULL if source or search_string is NULL.

Example 11-16 starts_with Function

In this example, the firstname field values that starts with the string "Pe" is indicated as true.

```
SELECT firstname, starts_with(firstname,"Pe") FROM users;


 +-----------+----------+
 | firstname | Column_2 |
 +-----------+----------+
 | John      | false    |
 | Peter     | true     |
 | Mary      | false    |
 +-----------+----------+
```

ends_with Function

The ends_with function indicates whether or not the source string ends with the search string.

Syntax

```
returnvalue ends_with(source, search_string)

source ::= any*
search_string ::= any*
returnvalue ::= boolean
```

Semantics

source
The input string to be searched. This argument is implicitly cast to a sequence of strings.

search_string
The string that should be searched in the source. This argument is implicitly cast to a sequence of strings.

returnvalue
Returns true if source ends with search_string else returns false.
Returns false if any argument is an empty sequence or a sequence with more than one item.
Returns NULL if source or search_string is NULL.

Example 11-17 ends_with Function

In this example, the firstname field values that ends with the string "hn" is indicated as true.

```
SELECT firstname, ends_with(firstname,"hn") FROM users;


 +-----------+----------+
 | firstname | Column_2 |
 +-----------+----------+
 | John      | true     |
 | Peter     | false    |
 | Mary      | false    |
 +-----------+----------+
```

index_of Function

The index_of function determines the position of the first character of the search string at its first occurrence, if any.

Syntax

```
returnvalue index_of(source, search_string [, start_position])

source ::= any*
search_string ::= any*
start_position ::= integer*
returnvalue ::= integer
```

Semantics

source
The input string to be searched. This argument is implicitly cast to a sequence of strings.

search_string
The string that should be searched in the source. This argument is implicitly cast to a sequence of strings.

start_position
An optional integer indicating, numerically, the position in the source from where the search should begin.
The default start_position is 0 which is also the position of the first character in the source.
If a negative value is supplied to start_position then 0 is assumed.

returnvalue
Returns the position of the first character of the search string at its first occurrence.
Returns -1 if search_string is not present in source.
Returns 0 for any value of source if the search_string is of length 0.
Returns NULL if any argument is NULL.
Returns NULL if any argument is an empty sequence or a sequence with more than one item.
Returns error if start_position argument is not an integer.

Note:

The returnvalue is relative to the beginning of source, regardless of the value of start_position.

Example 11-18 index_of Function

In this example, the index of "r" is selected in the firstname.

In the output, John has no occurrence of "r" so -1 is returned. Peter and Mary has "r" at 4 and 2 position respectively.

```
SELECT firstname, index_of(firstname,"r") FROM users;
```

```
 +-----------+----------+
 | firstname | Column_2 |
 +-----------+----------+
| John      |       -1 |
| Peter     |        4 |
| Mary      |        2 |
+-----------+----------+
```

Example 11-19 index_of Function

In this example, the index of "e" is selected in the firstname. In the output, notice that although "e" occurs twice in Peter, only the position of the first occurrence is returned.

```
SELECT firstname, index_of(firstname,"e") FROM users;


 +-----------+----------+
 | firstname | Column_2 |
 +-----------+----------+
 | John      |       -1 |
 | Peter     |        1 |
 | Mary      |       -1 |
 +-----------+----------+
```

replace Function

The replace function returns the source with every occurrence of the search string replaced with the replacement string.

Syntax

```
returnvalue replace(source, search_string [, replacement_string])

source ::= any*
search_string ::= any*
replacement_string ::= any*
returnvalue ::= string
```

Semantics

source
The input string that should be searched. This argument is implicitly cast to a sequence of strings.

search_string
The string that should be searched in the source. This argument is implicitly cast to a sequence of strings.

replacement_string
The string that should be substitued in place of search_string in the source. This is an optional argument. If replacement_string is omitted or empty sequence, then all occurrences of search_string are removed from source. The result will be checked so that the result would not be bigger than STRING_MAX_SIZE = 2^18 - 1 in chars ie. 512kb, if that is the case a runtime query exception is thrown. This argument is implicitly cast to a sequence of strings.

returnvalue
Returns source if the search_string argument is NULL.
Returns NULL if source argument is NULL.
Returns NULL if either source or search_string argument is an empty sequence.
Returns NULL if any argument is a sequence with more than one item.

Example 11-20 replace Function

In this example, the string "e" is replaced with "X" in all the occurences in firstname. Notice the occurrence of "X" in Peter.

```
SELECT firstname, replace(firstname,"e","X") FROM users;
```

```
 +-----------+----------+
 | firstname | Column_2 |
 +-----------+----------+
 | John      | John     |
 | Peter     | PXtXr    |
 | Mary      | Mary     |
 +-----------+----------+
```

Example 11-21 replace Function

In this example, the string "ar" is replaced with "urph". Notice that in the source the remaining characters after the search_string are retained for output. This yields the output for "Mary" as "Murphy".

```
SELECT firstname, replace(firstname,"ar","urph") FROM users;
```

```
 +-----------+----------+
 | firstname | Column_2 |
 +-----------+----------+
 | John      | John     |
 | Peter     | Peter    |
 | Mary      | Murphy   |
 +-----------+----------+
```

Example 11-22 replace Function

In this example, the replacement_string is not specified. Since the replacement_string is not specified, the search_string is removed and the remaining source is displayed.

```
SELECT firstname, replace(firstname,"oh") FROM users;
```

```
 +-----------+----------+
 | firstname | Column_2 |
 +-----------+----------+
 | John      | Jn       |
 | Peter     | Peter    |
 | Mary      | Mary     |
 +-----------+----------+
```

reverse Function

The reverse function returns the characters of the source string in reverse order, where the string is written beginning with the last character first. For example, the reverse order for "welcome" is "emoclew".

Syntax

```
returnvalue reverse(source)

source ::= any*
returnvalue ::= string
```

Semantics

source
The input string for which the characters should be reversed. This argument is implicitly cast to a sequence of strings.

returnvalue
Returns NULL if the source argument is NULL.
Returns NULL if the source argument is an empty sequence or a sequence with more than one item.

Example 11-23 reverse Function

In this example, the first name is displayed along with its reverse order.

```
SELECT firstname, reverse(firstname) FROM users;
```

```
 +-----------+----------+
 | firstname | Column_2 |
 +-----------+----------+
 | John      | nhoJ     |
 | Peter     | reteP    |
 | Mary      | yraM     |
 +-----------+----------+
```