Oracle

SQL for Oracle NoSQL Database Specification

12c Release 1

Library Version 12.1.4.3

ORACLE
NOSQL DATABASE

Legal Notice

Published 1/11/2017

Table of Contents

1 Introduction

This document describes the SQL dialect supported by Oracle NoSQL Database[1]. The data model of Oracle NoSQL supports (a) flat relational data, (b) hierarchical typed (schema-full) data, and (c) schema-less JSON data. SQL for Oracle NoSQL 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 (DML statement), or a data definition command (DDL statement), or 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
 | create_table_statement
 | alter_table_statement
 | drop_table_statement
 | create_index_statement
```

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

```
  | 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
  | describe_statement
  | show_statement)
    EOF
  ;
```

This document is concerned with the first 6 statements in the above list, that is, with queries 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 Getting Started with Oracle NoSQL Database Tables.

1.1 Antlr meta-syntax

This specification uses Antlr meta-syntax to specify the syntax of SQL. The following Antlr notations apply:

- Upper-case words are used to represent keywords, punctuation characters, operator symbols, and other syntactical entities that are recognized by Antlr as terminals (aka tokens) in the query text. For example, SELECT stands for the "select" keyword in the query text and LPAREN stands for a left parenthesis. Notice that keywords are case-insesitive. For example "select" and "sELEct" are both the same keyword, represented by the SELECT terminal.

- Anything enclosed in double quotes is also considered a terminal. For example, the following production rule defines the value-comparison operators as one of the =, !=, >, >=, <, or <= symbols:

```
 val_comp_op : "=" | "!=" | ">" | ">=" | "<" | "<=" ;
```

- Lower-case words are used for non-terminals. For example, filter_step : LBRACK expr RBRACK says that a filter_step is an expr enclosed in square brackets.

- * 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 precedes 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 literal.

- () Parentheses are used to group antlr sub-expressions together. For example, (INT | STRING)? means an integer, or a string, or nothing.

1.2 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. 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 section 5.3.2).

- // 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.

1.3 Identifiers and literals

In this section we describe some important terminals of the grammar, specifically identifiers and literals.

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.

In the grammar rules presented in this document we will use the symbol id to denote identifiers[2].

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:

```
INT_CONST : DIGIT+ ;

FLOAT_CONST : ( DIGIT* '.' DIGIT+ ([Ee] [+-]? DIGIT+)? ) |
```

[2] id is actually a non-terminal, but most of the “work” is done by the underlying ID terminal; see section 6 for the full grammar.

```
        ( DIGIT+ [Ee] [+-]? DIGIT+ ) ;

STRING_CONST : '\'' ((ESC) | .)*? '\'' ; // string with single quotes

DSTRING_CONST : '"' ((ESC) | .)*? '"' ;  // string with double quotes

ESC : '\\' ([\'\\/bfnrt] | UNICODE) ;

DSTR_ESC : '\\' (["\\/bfnrt] | UNICODE) ;

UNICODE : 'u' HEX HEX HEX HEX ;

TRUE : [Tt][Rr][Uu][Ee] ;

FALSE : [Ff][Aa][Ll][Ss][Ee] ;

NULL : [Nn][Uu][Ll][Ll] ;
```

1.4 SQL grammar

The full SQL grammar is included as an appendix (section 6) at the end of this document.

2 Data Model

This section defines the Oracle NoSQL data model in abstract terms. The Oracle NoSQL programmatic APIs provide specific classes that allow applications to create and navigate specific instances of the data model, both types and values (see Getting Started with Oracle NoSQL Database Tables).

In Oracle NoSQL, data is modeled as ***typed items***. A typed item (or simply ***item***) is a pair consisting of value and a type. A ***type*** is a definition of a set of values. The set defined by a type is called the type's ***domain***. In Oracle NoSQL values are never standalone; they can only exist in items. A value V and a type T can appear together in an item only if V belongs to T's domain. An item ***belongs to*** a type T if the value of the item belongs to T's domain.

Once an item is created, its type cannot be changed. Its value may be changeable, but only if the new value still belongs to the type's domain. For example, as we will see below, the Oracle NoSQL data model includes array values and types. If an array item has type ARRAY(INTEGER), then the associated array value must be an array containing integers only. Furthermore, although we can insert new values into the array, or change existing values, the new values must always be integers.

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

2.1 Atomic values

Currently, Oracle NoSQL supports the following kinds of atomic values:

- Integers : an integer is a 4-byte-long integer number.
- Longs : a long is an 8-byte-long integer number
- Floats : a float is a 4-byte-long real number
- Doubles : a double is an 8-byte-long real number
- Strings : a string is a sequence of unicode characters
- Booleans : there are only two boolean values, true and false
- Binaries : a binary value is an uninterpreted sequence of zero or more bytes
- Enums: an enum value is a symbolic identifier (token). Enums are stored as strings, but are not considered to be strings.
- Timestamps: values 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 the ***precision*** of a timestamp value. For example, a precision of 0 means that no fractional seconds are stored, 3 means that the timestamp stores milliseconds, and 9 means a precision of nanoseconds. 0 is the minimum precision, and 9 is the maximum. There is no timezone information stored in timestamp; they are all assumed to be in the UTC timezone. The number of bytes used to store a timestamp depends on its precision (the on-disk storage varies between 5 and 9 bytes).
- The json null value.
- The SQL NULL. It is a special "value" that is used to indicate the absence of an actual value, or the fact that a value is unknown or inapplicable.

In the remainder of this document we will use the term "NULL" to refer to the SQL NULL, and "JNULL" to refer to the json null value.

2.2 Complex values

Currently, Oracle NoSQL supports the following kinds of complex values:

- Arrays: an ***array*** is an ordered collection of zero or more items. The items of an array are called ***elements*** and they can have different types. However, arrays cannot contain any NULLs.

- Maps: 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 ***field names***, and the associated items are called ***field values.*** Field values can have different types. However, maps cannot contain any NULL field values.

- Records: 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 ***field names***, and the associated items are called ***field values***. Furthermore, records may contain fields with NULL as their value.

2.3 Atomic types

Currently, Oracle NoSQL supports the following primitive atomic types:

- Integer : All integer values

- Long : All long values
- Float : All float values
- Double : All double values
- String : All string values
- Boolean : All boolean values
- Binary : All binary values

Oracle NoSQL also supports the following parametrized atomic types (there is a different concrete type for each possible setting of the associated type parameters):

- FixedBinary(S) : All binary values whose length is equal to S.

- Timestamp(P) : All timestamp values with precision P.

- Enum(T1, T2, ..., Tn) : The ordered collection that contains the tokens (enum values) T1, T2, ... Tn.

Notice that there is no type for NULL. Unless explicitly excluded (as, for example, in the case of array and map elements), NULL is assumed to be in the value space of every type. There is also no specific type for the json null value. As we will see below, JNULL belongs to the JSON wildcard type.

2.4 Complex types

Oracle NoSQL supports the following parametrized complex types:

- Array(T) : All arrays whose elements belong to type T. T is called the ***element type*** of the array type.

- Map(T) : All maps whose field values belong to type T. T is called the ***value type*** of the map type.

- Record(k1 T1 n1, k2 T2 n2, ..., kn Tn nn) : All records of exactly n fields, where for each field i (a) the field name is ki, (b) the field value belongs to type Ti, and (c) the field conforms to the ***nullability property*** ni, which specifies whether the field value may be NULL or not.

2.5 Wildcard types and JSON data

The Oracle NoSQL data model includes the following ***wildcard types*** as well:

- Any : All possible values.
- AnyAtomic : All possible atomic values.
- AnyJsonAtomic : All atomic values that are valid JSON values. This is the union of all numeric values, all string values, the 2 boolean values, and the JNULL value.
- Json : All possible json values. The domain set is D is defined recursively as follows: (a) it includes all AnyJsonAtomic values, (b) it includes all arrays whose elements belong to D, and (c) it includes all maps whose field values belong to D.
- AnyRecord : all possible record values.

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

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

To load json data into a table, the programmatic APIs accept input json as strings or streams containing json text. Oracle NoSQL 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). This is done unconditionally, no matter what the actual contents of the array might be. For example, even if the array contains integers only, the array item that will be created will have type Array(Json). The reason that the array is not created with type Array(Integer) is that this would mean that we could never update the array by putting something other than integers. For the same reason, when a json object is encountered in the input text, a map item is created whose type is Map(Json), unconditionally. When numbers are encountered, they are converted to integer, long, or double items, depending on the actual value of the number (float items are not used for json). Finally, strings in the input text are mapped to string items, boolean values are mapped to boolean items, and json nulls to json null items.

2.6 Type hierarchy

The data model also defines a ***subtype-supertype relationship*** among the types presented above. An item is an ***instance of*** type T if the type of the item is T or a subtype of T. This relationship is important because the usual subtype-substitution rule is supported by SQL for Oracle NoSQL: if an operation expects input items of type T or produces items of type T, then it can also operate on or produce items of type S, where S is a subtype of T. However, there is an exception to this rule, which will be explained below.

Based on this relationship, the Oracle NoSQL types can be arranged in a hierarchy. The top levels of this hierarchy are shown in figure 1 (dotted boxes in the figure represent collections of types). For example, every type is a subtype of ANY, any atomic type is a subtype of AnyAtomic, Integer is a subtype of Long, and an array type is a subtype of Json if its element type is Json or subtype of Json. In addition to the subtype relationships shown in the figure, the following relationships are defined as well:

- 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.

- An enum type is a subtype of another enum type if both types contain the same tokens and in the same order, in which case the types are actually considered equal.

- Timestamp(p1) is a subtype of Timestamp(p2) if p1 <= p2.

- A record type S is a subtype of another record type T if (a) both types contain the same field names and in the same order, (b) for each field, its value type in S is a subtype of its value type in T, and (c) if the field is nullable in S, it is also nullable in T.

- Array(S) is a subtype of Array(T) if S is a subtype of T.

- Map(S) is a subtype of Map(T) if S is a subtype of T.

As mentioned above, there is an exception to the subtype-substitution rule. Specifically, 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 arrays whose element type is Json, Array(Json) or Map(Json). This is in order to disallow strongly type data to be inserted into json data. For example, consider a json document M, i.e., a map value whose associated type is 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 would still keep it a json document.

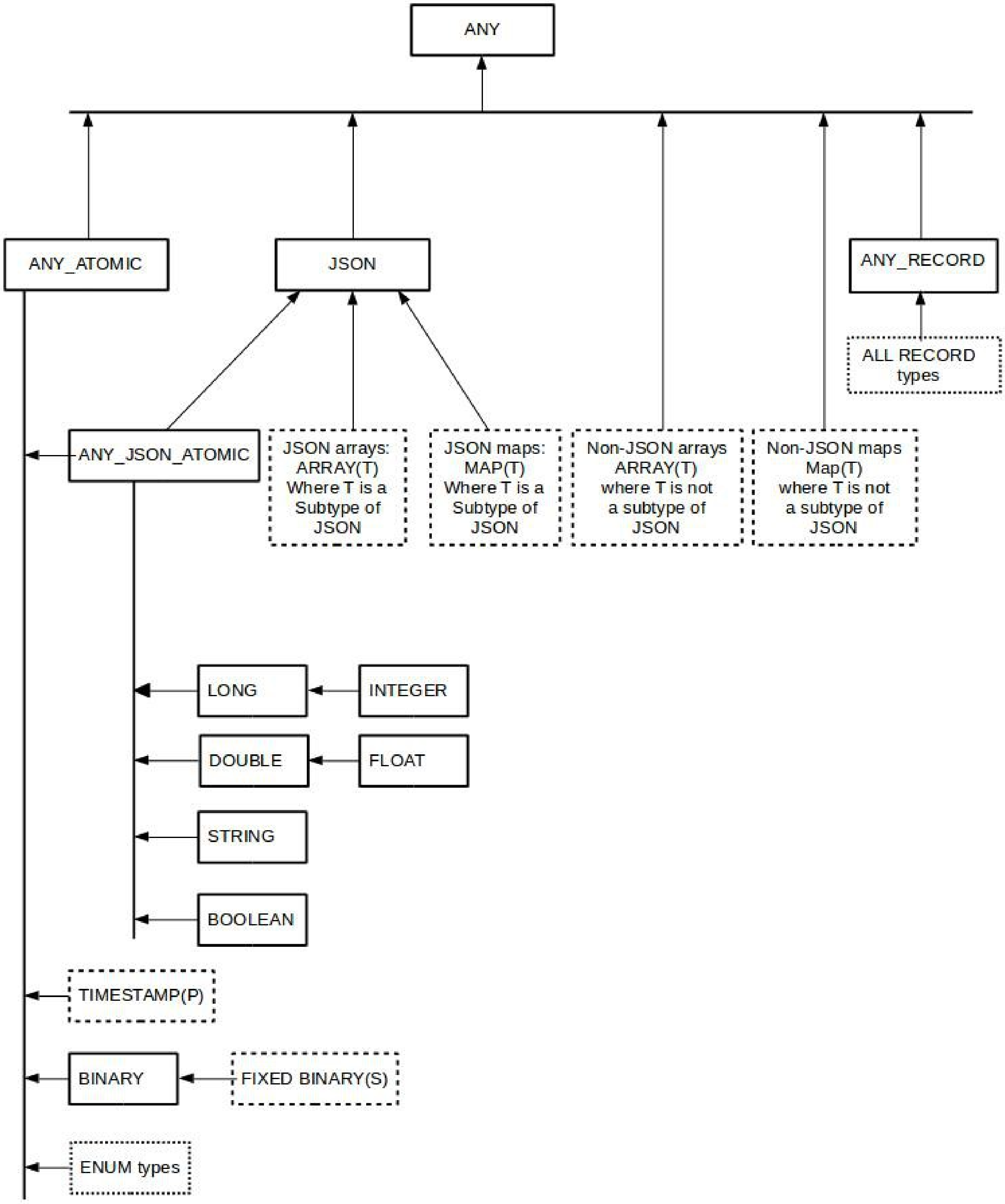
ANY
ANY_ATOMIC
JSON
ANY_RECORD
ALL RECORD types
ANY_JSON_ATOMIC
JSON arrays: ARRAY(T) Where T is a Subtype of JSON
JSON maps: MAP(T) Where T is a Subtype of JSON
Non-JSON arrays ARRAY(T) where T is not a subtype of JSON
Non-JSON maps Map(T) where T is not a subtype of JSON
LONG
INTEGER
DOUBLE
FLOAT
STRING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

Figure 1: SQL type hierarchy

2.7 Tables

In Oracle NoSQL, data is stored and organized in tables. 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 (see section 3.1). The records of a table are called ***rows*** and the record fields are called ***columns***. Therefore, an Oracle NoSQL table is a generalization of the (normalized) relational tables found in more traditional RDBMSs.

2.8 Type definitions

The following syntax is used to refer to the data model types inside SQL statements. Currently, this syntax is used in both DDL statements (mainly in the CREATE TABLE statement, but with some restrictions described in section 3.1) and DML statements. It is also used in this document to describe the sequence types defined in section 4.1.

```
type_def :
  ANY,
  ANYRECORD,
  ANYATOMIC,
  ANYJSONATOMIC,
  JSON,
  INTEGER |
  LONG |
  FLOAT |
  DOUBLE |
  STRING |
  BOOLEAN |
  timestamp_def |
  enum_def |
  binary_def |
  record_def |
  array_def |
  map_def ;

timestamp_def : TIMESTAMP (LP INT_CONST RP)? ;

enum_def : ENUM LPAREN id_list RPAREN ;

binary_def : BINARY (LPAREN INT_CONST RPAREN)? ;

map_def : MAP LPAREN type_def RPAREN ;

array_def : ARRAY LPAREN type_def RPAREN ;

record_def : RECORD LPAREN field_def (COMMA field_def)* RPAREN ;
```

```
field_def : id type_def default_def? comment? ;

default_def : (default_value (NOT NULL)?) |
            (NOT NULL default_value?) ;

comment : COMMENT string ;

default_value : DEFAULT (number | string | TRUE | FALSE | id) ;

number : MINUS? (FLOAT_CONST | INT_CONST) ;

string : STRING_CONST | DSTRING_CONST ;
```

Notice that according to the default_def rule, by default all record fields are nullable. Furthermore, this rule allows for an optional ***default value*** for each field, which is used during the creation of a record belonging to a given record type: If no value is assigned to a field, the default value is assigned by Oracle NoSQL, if a default value has been declared. 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.

Notice also that the field_def rule specifies an optional comment, which if present, is actually stored persistently as the ***field's description***.

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

In specifying a timestamp type, the precision is optional. If omitted, the default precision is 9 (nanoseconds). This implies that the type `Timestamp` (with no precision specified) is a super type of all other timestamp types (with a specified precision). However, `Timestamp` cannot be used in the CREATE TABLE statement; in that context, a precision must be explicitly specified. This restriction is to prevent users from inadvertently creating timestamp values with precision 9 (which take more space), when in reality they don't need that high precision.

3 Creating and managing tables

3.1 Create Table Statement

```
create_table_statement :
  CREATE TABLE (IF NOT EXISTS)? table_name comment?
  LPAREN table_def RPAREN ttl_def? ;

table_name :
  name_path;
```

```
name_path :
  id (DOT id)* ;

table_def :
  (field_def | key_def) (COMMA (field_def | key_def))* ;

key_def :
  PRIMARY KEY
  LPAREN (shard_key_def COMMA?)? id_list_with_size? RPAREN ;

id_list_with_size : id_with_size (COMMA id_with_size)* ;

id_with_size : id storage_size? ;

storage_size : LPAREN INT_CONST RPAREN ;

shard_key_def : SHARD LPAREN id_list_with_size RPAREN;

ttl_def : USING TTL INT_CONST (HOURS | DAYS) ;
```

A CREATE TABLE statement starts with an optional IF NOT EXISTS clause, then specifies the name of the table to create, followed by an optional table-scoped comment, followed by the description of the table's fields (a.k.a. columns) and primary key, enclosed in parentheses., and finishes with an optional specification of the default TTL value for the table.

The table name is specified as a name_path, because in the case of descendant tables, it will consist of a list of dot-separated ids.

By default if a table with the same name exists, the create table statement generates an error indicating that the table exists. If the optional "IF NOT EXISTS" clause is specified **and** the table exists (or is being created) **and** the existing table has the same structure as in the statement, no error is generated.

The TTL specification, if present, gives the default TTL value to associate with a row, when the row is inserted in the table, if a specific TTL value is not provided via the row insertion API. The TTL value associated with a row specifies the time period (in hours or days) after which the row will “expire”. Expired rows are not included in query results and are eventually removed from the table automatically by Oracle NoSQL.

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

The syntax for a field definition uses the field_def grammar rule that defines the fields of a record type (see section 2.8). It specifies the name of the field/column, its data type, whether the field is nullable or not, an optional default value, and an optional comment. As mentioned in section 2.7, tables are containers of records, and the table_def acts as an implicit definition of a record type (the table schema), whose fields are defined by the listed field_defs. However, 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.

The syntax for the primary key specification (key_def) specifies both the primary key of the table and the shard key. The primary key is an ordered list of field names. The field names must be among the ones appearing in the field_defs, and their associated type must be a numeric type or string or enum. A shard key is specified as part of the primary key by using the SHARD key word in the PRIMARY KEY clause to indicate the sublist of the primary-key fields to use for sharding. The sublist must start with the first field in the primary-key list and contain a number of consecutive fields from the primary-key list. Specification of a shard key is optional. By default, for a top-level table (a table without a parent) the shard key is the primary key. A child table must not specify a shard key because it inherits its parent table's shard key.

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.

Size (number of bytes)	Range of values
1	0 - 63
2	64 - 8191
3	8192 - 1048575
4	1048576 - 134217727
5	134217728 - MAX_INT

Finally, a create table statement may include a table-level comment that becomes part of the table's metadata as uninterpreted text. COMMENT strings are displayed in the output of the "DESCRIBE" statement.

3.1.1 Example

The following CREATE TABLE statement defines a table that will be used in the DML and DDL examples shown in rest of this document.

```
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 may be just one string. Or it may not even be an address at all.

3.2 DROP TABLE Statement

```
drop_table_statement : DROP TABLE (IF EXISTS)? name_path ;
```

The DROP TABLE statement removes the specified table and all its associated indexes from the database. By default if the named table does not exist this statement fails. If the optional "IF EXISTS" is specified and the table does not exist no error is reported.

3.3 ALTER TABLE Statement

```
alter_table_statement :
  ALTER TABLE table_name (alter_field_statements | ttl_def) ;

alter_field_statements :
  LPAREN alter_field_stmt (COMMA alter_field_stmt)* RPAREN ;

alter_field_stmt : add_field_stmt | drop_field_stmt ;

add_field_stmt : ADD schema_path type_def default_def? comment? ;

drop_field_stmt : DROP schema_path ;

schema_path : init_schema_path_step (DOT schema_path_step)*;

init_schema_path_step : id (LBRACK RBRACK)* ;

schema_path_step : id (LBRACK RBRACK)* | VALUES LP RP ;
```

The ALTER TABLE statement allows an application to add or remove a schema field from the table schema. It also allows to change the default TTL value for the table. Adding or 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. 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/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. For example to add a "middle" name into the names stored in "other_names":

```
ALTER TABLE Users ADD other_names[].middle STRING
```

The path syntax is a subset of the one used in queries and will be described in section 4.4

4 SQL DML: The Query Statement

In the current SQL version, a ***query*** is a statement that consists of zero or more variable declarations, followed by single SELECT-FROM-WHERE (SFW) expression:

```
query :
  var_decls? sfw_expr ;
```

The result of a query is always a sequence of records. In most queries, the records are constructed by the SFW expression. However, as we will see, some times the SFW expression does not return records. In such cases, a thin layer on top of the query processor will wrap each non-record value V into a record with one field only whose name is "Column_1" and whose value is V.

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.

4.1 Expressions, sequences, and sequence type

In general, a query ***expression*** represents a set of operations to be executed in order to produce a result. Expressions are built by combining other (sub)expressions via operators, function calls, or other grammatical constructs. As we will see, the simplest kinds of expressions (having no subexpressions) are constants (aka literals) and references to variables or identifiers.

In SQL for Oracle NoSQL, the result of any expression is a ***sequence*** of zero or more items (including NULLs). We will refer to such sequences as ***svalues***. Notice that a single item is considered equivalent to a sequence containing that single item.

We should emphasize the difference between a sequence and an array. An array is a single item, albeit one that contains other items in it. A sequence is just a set of items. 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. In contrast, arrays are containers. A sequence produced by an expression E can be converted to an array by wrapping E with an array constructor : [E] (see section 4.15). This is called ***boxing*** the sequence. Conversely, there are expressions that ***unbox*** an array: they select 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.

A comparison with standard SQL may also be helpful in clarifying the sequence model used by Oracle NoSQL. 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 too, most expressions are scalar. Like the query blocks of standard SQL, the select-form-where expression of Oracle NoSQL returns a sequence of items. However, to navigate and extract information from complex, hierarchical data, Oracle NoSQL includes path expressions as well (see section 4.4). Path expressions are the other main source of multi-item sequences in Oracle NoSQL. However, if path expressions are viewed as subqueries, the Oracle NoSQL model is not that different from standard SQL.

In the remainder of this chapter we will present the kinds of expressions that are currently supported by Oracle NoSQL. For each expression, we will first show its syntactic form, then define its semantics, and finally give one or more examples of its usage. Part of the semantic definition is to describe the type of the items an expression operates on, and the type of its result set. Given that each expression operates on one or more input sequences and produces a sequence, the concept of a ***sequence type*** is useful in this regard. 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. Specifically, the following syntax is used to specify a ***sequence type***, i.e., a set of svalues:

```
sequence_type : item_type quantifier? ;

item_type : type_def ;
```

```
quantifier : STAR | PLUS | QUESTION ;
```

The item_type is one of the types in the data model. 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.

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 if SUB's item type is a subtype of SUP's item type, and SUB's quantifier is a sub-quantifier of SUP's quantifier, where the sub-quantifier relationship is defined by the following table:

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

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

4.2 External variable declarations

```
var_decls :
  DECLARE var_decl SEMICOLON (var_decl SEMICOLON)*;

var_decl :
  var_name type_def;

var_name :
  DOLLAR id ;
```

As mentioned already, a query starts with a variables declarations section. The variables declared here are called ***external variables***, and they 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++). However, contrary to java or c++, the values of external variables are not known in advance, i.e. 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 Getting Started with Oracle NoSQL Database Tables. The type of the item bound to an external variable must be 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.

As we will see later, in addition to external variables, Oracle NoSQL allows for the (sometimes 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.

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. As we will see, 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.

The names of variables are case-sensitive. A small set of variable names cannot be used as names for external variables: $key, $value, $element, and $pos.

4.2.1 Example:

```
declare $age  integer;
select firstName, lastName
from Users
where age > $age
```

The above 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.

4.3 Select-From-Where (SFW) expression

```
sfw_expr :
  select_clause
  from_clause
  where_clause?
  orderby_clause?
  limit_clause?
  offset_clause? ;

from_clause : FROM table_name tab_alias? ;

table_name : name_path ;

tab_alias : AS ? DOLLAR? id ;
```

```
where_clause : WHERE expr ;

select_clause :
  SELECT hints? (STAR |
             (expr col_alias? (COMMA expr col_alias?)*)) ;

hints : '/*+' hint* '*/' ;

hint : ( (PREFER_INDEXES LP name_path index_name* RP) |
        (FORCE_INDEX    LP name_path index_name  RP) |
        (PREFER_PRIMARY_INDEX LP name_path RP)       |
        (FORCE_PRIMARY_INDEX  LP name_path RP) ) STRING?;

col_alias : AS id ;

orderby_clause :
  ORDER BY expr sort_spec (COMMA expr sort_spec)* ;

sort_spec : (ASC | DESC)? (NULLS (FIRST | LAST))? ;

limit_clause : LIMIT add_expr ;

offset_clause : OFFSET add_expr ;

expr : or_expr ;
```

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

4.3.1 FROM clause

As shown in the grammar, in the current version, the FROM clause is very simple: it can include only a single table. The table is specified by its name, which may be a composite (dot-separated) name in the case of child tables. The table name may be followed by a table alias. The result of the FROM clause is a sequence containing the rows of the referenced table. The FROM clause creates a nested scope, which exists for the rest of the SFW expression.

The SELECT, WHERE, and ORDER BY clauses operate on the rows produced by the FROM clause, processing one row at a time. The row currently being processed is called the ***context row***. The context row can be referenced in expressions by either the table name, or the table alias (and as we will see in section 4.13, sometimes no explicit reference is needed for “simple” column references). 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 and can be referenced within the SFW expression, anywhere an expression returning a single record may be used. Notice that if this variable has the same name as an external variable, it hides the external variable. Because table alias are essentially variables, their names are case-sensitive, like variable names.

4.3.2 WHERE Clause

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

4.3.3 ORDER BY clause

The ORDER BY clause reorders the sequence of rows it receives as input. The relative order between any two input rows is determined by evaluating, for each row, the expressions listed in the order-by clause and comparing the resulting values. Each order-by expression must have type AnyAtomic?. Let N be the number of order-by expressions and let Vi1, Vi2, … ViN be the atomic values returned by evaluating these expressions, from left to right, on row Ri (if an expression returns an empty sequence, NULL is used as the returned value). Two rows Ri, Rj are considered equal if Vik is equal to Vjk for each k in 1, 2, …, N. In this context, NULLs are considered to be equal only to themselves. Otherwise, Ri is considered less than Rj if there is a pair Vim, Vjm such that:

- m is 1, or Vik is equal to Vjk for each k in 1, 2, …, (m-1), and
- Vim is not equal to Vjm, and
- Neither of Vim, Vjm is NULL and either (a) Vim is less than Vjm and the m-th sort_spec specifies ascending order, or (b) Vim is greater than Vjm and the m-th sort_spec specifies descending order, or
- Vim is NULL and either (a) the m-th sort_spec specifies ascending order and NULLS FIRST, or (b) m-th sort_spec specifies descending order and NULLS LAST, or
- Vjm is NULL and either (a) the m-th sort_spec specifies ascending order and NULLS LAST, or (b) m-th sort_spec specifies descending order and NULLS FIRST.

In the above rules, comparison of any two values Vik and Vjk, when neither of them is NULL, is done according to the rules of the value-comparison operators defined in section 4.5

Notice that in the grammar, a sort_spec is optional. If no sort_spec is given, the default is ASC order and NULLS LAST. If only the sort order is specified, then NULLs sort last if the order is ASC, otherwise they sort first. If the sort order is not specified, ASC is used.

The above rules describe the general semantics of the ORDER BY clause. However, the current implementation imposes an important restriction on when ordering can actually be done. Specifically, ordering is possible only if there is an index (see section 5.1) that already sorts the rows in the desired order. More precisely, let e1, e2, …, eN 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 order_specs must specify the same ordering direction and for each sort_spec, the desired ordering with respect to NULLs must match the way NULLs are sorted by the index. In the current implementation, NULLs 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.

4.3.4 SELECT Clause

The SELECT clause comes in two forms: one containing a single star symbol (*) and the other containing a list of expressions, where each expression is optionally associated with a name. In the second form, we will refer to the listed expressions and their associated names as *field expressions* and *field names* respectively.

In its "select star" form, the SELECT clause is a noop; it simply returns its input sequence of rows.

In its "projection" form, the SELECT clause creates a new record for each input row, unless there is only one field expression with no associated name. In the later case, the SELECT clause just returns the value computed by the single field expression. The value may or may not be a record. As mention earlier, if it is not a record it will be wrapped into a record before it gets returned to the application. In the former case, the new record 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 associated with the field expression. 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 expression must have type ANY? (i.e., must return at most one item). If the compiler determines that a field expression may return more than one results, it warps the field expression with an array constructor (see section 4.15). 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. If the result of a field expression is empty, NULL is 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 may also contain one or more hints, that help the query processor choose an index to use for the query. Hints are explained further in section 5.3.2.

4.3.5 OFFSET and LIMIT clauses

The offset clause is used to specify a number N of initial query results that should be skipped (not returned to the application). The limit clause is used to specify the maximum number M of results to return to the application. N and M are each 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.

Although it's possible to use offset/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 skipped (if offset is used) and the subset of results returned (if limit is used) will be different each time the query is run.

4.3.6 Examples

In this section we show some simple SFW examples. More complex examples will be shown in following sections that present other kinds of expressions.

Select all information for all users

```
select * from Users
```

Select all information for users whose first name is "John"

```
select * from Users where firstName = "John"
```

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 (see section 4.13 for more details).

```
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
```

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
```

Select 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
```

4.4 Path expressions

```
path_expr :
  primary_expr (map_step | array_step)* ;
```

```
map_step : map_field_step | map_filter_step ;
```

```
map_field_step : DOT ( id | string | var_ref | parenthesized_expr | func_call );

map_filter_step : DOT (KEYS | VALUES) LP expr? RP ;

array_step : array_filter_step | array_slice_step;

array_filter_step : LBRACK expr? RBRACK ;

array_slice_step : LBRACK expr? COLON expr? RBRACK ;
```

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 section 4.11), 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.

4.4.1 Field step expressions

```
map_field_step :
  DOT ( id | string | var_ref | parenthesized_expr | func_call);
```

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 the 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 $ 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).

4.4.2 Examples

- 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 (address in the above example), a table alias must be used together with the column name. Otherwise, as explained in section 4.14, 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)[3].

- 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).

[3] 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.

- 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
```

- 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
```

- For each user select their id and all of their phone numbers (without the are 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.
```

4.4.3 Map-filter step expressions

```
map_filter_step : DOT (KEYS | VALUES) LP expr? RP ;
```

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 of 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).

4.4.4 Examples

- 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
```

- 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
```

- For each user select their id and their expenses in all categories except housing.

```
select id, u.expenses.values($key != housing)
from Users u
```

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.

4.4.5 Array-filter step expressions

```
array_filter_step : LBRACK expr? RBRACK ;
```

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 of 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.

- For each user, select their last name and his/her phone numbers with area code 650. Notice the the path expression in the select clause is enclosed in square brackets, which is the syntax used for array-constructor expressions (see section 4.15). 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.

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

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

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

4.5 Logical operators: AND, OR, and NOT

- 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. 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.

4.6.1 Example

We have already seen examples of comparisons among atomic items. Here is an example involving 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 section 4.15) 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]
```

4.7 Sequence comparison operators

```
comp_expr : add_expr ((comp_op | any_op) add_expr)? ;

any_comp_op :
  "=any" | "!=any" | ">any" | ">=any" | "<any" | "<=any" ;
```

- 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 section 4.4.5), which returns all the elements of the connections array as a sequence (it is *unnesting* the array).

```
where connections[] =any 3
```

- 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
```

- 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
```

4.8 Exists operator

```
exists_expr : EXISTS add_expr ;
```

The exists operator is very simple: it just checks whether its input sequence is empty or not and returns false or true respectively.

4.8.1 Example

- 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 JNULL 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
```

4.9 Is-Of-Type operator

```
is_of_type_expr :
  add_expr IS NOT? OF TYPE?b
  LP ONLY? sequence_type (COMMA ONLY? sequence_type)* RP;

sequence_type : type_def quantifier? ;

quantifier : STAR | PLUS | QUESTION ;
```

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:

(a) the cardinality of the input sequence matches the quantifier of the target type. Specifically, (1) if the quantifier 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.

(b) 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 (a) 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.

4.9.1 Example

- 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)
```

4.10 Arithmetic expressions

```
add_expr : multiply_expr ((PLUS | MINUS) multiply_expr)* ;

multiply_expr : unary_expr ((STAR | DIV) unary_expr)* ;

unary_expr : path_expr | (PLUS | MINUS) unary_expr ;
```

Oracle NoSQL supports the usual arithmetic operations: +, -, *, and /. 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 double item, the item returned by the other operand is cast to a double value 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 and the result is a float item that is computed using java's arithmetic on floats, otherwise,
- If any operand returns a long item, the item returned by the other operand is cast to a long value and the result is a long item that is computed using java's arithmetic on longs, otherwise,
- All operands return integer items, and the result is an integer item that is computed using java's arithmetic on ints.

Oracle NoSQL supports the unary + and – operators as well. The unary + is a noop, and the unary – changes the sign of its numeric argument.

4.10.1 Example

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
```

4.11 Primary expressions

```
primary_expr :
  const_expr |
  column_ref |
  var_ref |
  array_constructor |
  map_constructor |
  case_expr |
  cast_expr |
  func_call |
  parenthesized_expr ;
```

The following sections describe each of the primary expressions listed in the above grammar rule.

4.12 Constant expressions

```
const_expr : INT_CONST | FLOAT_CONST | string | TRUE | FALSE | NULL ;

string : STRING_CONST | DSTRING_CONST ;
```

The syntax for INT_CONST, FLOAT_CONST, STRING_CONST, and DSTRING_CONST was given in section 1.3.

In the current version, a query can contain 5 kinds of constants (a.k.a. literals):

- Strings: sequences of unicode characters enclosed in double or single quotes. String literals are translated into String items.
- Integer numbers: sequences of one or more digits. Integer literals are translated into Integer items, if their value fits in 4 bytes, otherwise into Long items.
- Real numbers : representation of real numbers using “dot notation” and/or exponent. Real literals are translated into Double items.
- The boolean values true and false.
- The json null value.

4.13 Column references

```
column_ref : id (DOT id)? ;
```

A column-reference expression returns the item stored in the specified column within the context row (the row that a WHERE, ORDER BY, or SELECT clause is currently working on). 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 name/alias and the second a column in that table. A single id refers to a column in the table referenced inside the FROM clause.

Notice that child tables in Oracle NoSQL have composite names using dot as a separator among multiple ids. As a result, a child-table name cannot be used in a column-reference expression; instead, a table alias must be used to access a child table column via the 2-id format.

4.14 Variable references

```
var_ref : DOLLAR id? ;
```

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.

4.15 Array and map constructors

```
array_constructor : LBRACK expr (COMMA expr)* RBRACK ;

map_constructor :
   (LBRACE expr COLON expr (COMMA expr COLON expr)* RBRACE) |
   (LBRACE RBRACE) ;
```

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 section 2.6).

4.15.1 Example

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. Notice that the use of an explicit array for the “high_expenses” field guarantees that the field will exist in all of 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.

```
select
```

```
 {
   “last_name” : u.lastName,
   “phones” : u.address.phones,
   “high_expenses” : [ u.expenses.keys($value > 5000) ]
 }
from Users u
```

4.16 Searched case expression

```
case_expr :
  CASE WHEN expr THEN expr (WHEN expr THEN expr)* (ELSE expr)? END;
```

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.

4.16.1 Example

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. The query is very similar to the one from section 4.15.1. 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 emty 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.

```
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
```

4.17 Cast expression

cast_expr : CAST LP expr AS sequence_type RP ;

sequence_type : type_def quantifier? ;

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. If the type of the input item is equal to the target item type, the cast is a noop: the input item itself is returned. If the target type is a wildcard type, the cast is a noop if the type of the input item is a subtype of the wildcard type; otherwise an error is raised. An error is also raised if the input item is not castable to the target type. The following rules specify the castability among the various types.

- Every atomic item is castable to the String type
  .
- Every numeric item can be cast to every other numeric type.

- 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.

- Array items may be cast to array types only. A new array is created, whose type is the target type, and each element in the input array to cast to the element type of the target array. If this cast is successful, the new element is added to the new array.

- Map items may be cast to map types only. A new map is created, whose type is the target type, and each field value in the input map to cast to the value type of the target map. If this cast is successful, the new field value and the associated field name are added to the new map.

- Record items may be cast to record types only. The target type must have the same fields and in the same order as the type of the input item. A new record is created, whose type is the target type, and each field value in the input record to cast to the value type of the corresponding field in the target type. If this cast is successful, the new field value and the associated field name are added to the new record.

Example

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

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

4.18 Function calls

```
func_call : id LPAREN (expr (COMMA expr)*)? RPAREN ;
```

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 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 later 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 only function currently implemented is the size() function:

- INTEGER? **size**(item ANY?)

Returns the size of a complex item (array, map, record). Although the parameter type appears as ANY?, the runtime implementation of this function raises 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.

4.19 Parenthesized expressions

```
parenthesized_expr : LPAREN expr RPAREN;
```

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 (see section 4.4.1). An example of the former usage is this:

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
```

5 Indexing in Oracle NoSQL

Roughly speaking, indexes are ordered maps, mapping values contained in the rows of a table back to the containing rows. As such, indexes provided fast access to the rows of a table, when the information we are searching for is contained in the index. In section 5.1 we define the semantics of the CREATE INDEX statement and describe how the contents of an index are computed. As we will see, Oracle NoSQL comes with rich indexing capabilities, including indexing of deeply nested fields, and indexing of arrays and maps. However, the current implementation does not support indexing of JSON data. In section 5.2 we describe the DROP INDEX statement. Finally, in section 5.3 we will see how indexes are used to optimize queries.

5.1 Create Index Statement

```
create_index_statement :
   CREATE INDEX (IF NOT EXISTS)?
   index_name ON table_name LPAREN path_list RPAREN comment?;

index_name : id ;

path_list : index_path (COMMA index_path)* ;

index_path : (name_path | keys_expr | values_expr | brackets_expr) ;

name_path : id (DOT id)* ;

keys_expr : name_path DOT KEYS LP RP ;

values_expr : name_path DOT VALUES LP RP (DOT name_path)?;

brackets_expr : name_path LBRACK RBRACK (DOT name_path)?
```

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 are implemented as B-Trees.

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 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. 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.

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, or string, timestamp, or enum. Index fields may also store NULL values. In the index schema, the 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). NULL values are considered to be greater than any other values (i.e., NULLs sort last in indexes).

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 sections 5.1.2 and 5.1.3 below.

Syntactically, an index path is specified using a subset of the syntax for path expressions in queries (see section 4.4). The following remarks/restrictions apply to index paths:

- Although query path expressions must include an input expression, this is not required 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 identifiers. Normally, these identifiers are ***schema fields***, i.e., field names that appear in the CREATE TABLE statement used to create the associated table. Using the example table from section 3.1.1, age and otherNames.last are valid index paths for indexes over the Users table. However, name_path identifiers may also be 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.

- A name_path cannot contain any schema fields that have an array type. We will refer to this restriction as the ***name-path-restriction***. For example, address.otherNames.last cannot be used as an index path (because otherNames is an array field), although it is a valid query path expression. However, the equivalent address.otherNames[].last path can be used instead. A consequence of the name-path-restriction is that when evaluated on a table row, a name_path will always return at most one result item.

- A name_path cannot contain any field whose type is JSON or a subtype of JSON. Indexing JSON data is not supported yet.

- In the context of index creation, if a name_path returns an empty result, NULL 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 NULL. So, in effect, each name_path returns exactly one item.

- 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 indexing JSON data is not supported yet, for the remainder of section 5, 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 Users (
  id INTEGER,
  firstName STRING,
  lastName STRING,
  otherNames ARRAY(RECORD(first STRING, last STRING)),
  age INTEGER,
  income INTEGER,
  address RECORD(street : STIRNG,
              number : INTEGER,
              city : STRING,
              state : STRING,
              phones : ARRAY(RECORD(area : INTEGER,
                                    number : INTEGER,
                                    kind : STRING))
             ),
  connections ARRAY(INTEGER),
  expenses MAP(INTEGER),
  PRIMARY KEY (id),
)
```

5.1.1 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. As a result, there is exactly one index entry per table row. Notice that the entries of a simple index over a table T are essentially the result of the following query:

```
select IP1, …, IPn, PK1, …, PKk
from T
```

where IPi is the i-th index path appearing in the CREATE INDEX statement, and PKi is the i-th primary-key column of table T.

5.1.2 Simple index examples

- create index idx1 on Users (income)

  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.

- create index idx2 on Users (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 (the primary key) of the user represented by the row.

- create index idx3 on nodes (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 NULL otherwise.

- create index idx4 on users (expenses.housing, expenses.travel)

  Creates an index entry for each user. The entry contains the user's housing expenses, or NULL if the user does not record housing expenses, and the user's travel expenses, or NULL if the user does not record travel expenses.

5.1.3 Multi-key indexes

Mutli-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. 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 Users 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 section 5.1.1.

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 complex2.matrix array contains other arrays. Neither is complex2.matrix[][].a valid, because the syntax allows at most one [] per index path. As a result, we cannot index two arrays/maps where one is nested inside the other. 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):

4.1 If the array contains indexable atomic items, then:

4.1.1 There must be a single 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.

4.1.2 In this case, for each table row R, a number of index entries are created as follows: M and the simple index paths (if any) are computed on R. Then, M[] is computed, returning either NULL, if M returned NULL, 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.

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

4.2 If the array contains records or maps, then:

4.2.1 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.

4.2.2 In this case, for each table row R, a number of index entries are created as follows: M and the simple index paths (if any) are computed on R. Then, M[] is computed, returning either NULL, if M returned NULL, 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, 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.

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

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:

5.1 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.

5.2 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.

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

5.1.4 Multi-key index examples

In this section we give some examples of multi-key indexes. For each index we also present a query that computes the content of the index. However, the queries shown here are not expressible in the current SQL version.

- create index midx1 on Users (connections[])

Creates an index on the elements of the connections array. The elementof keyword in the above statement is optional. The contents of this index are the results of the following query:

```
 select distinct connection, user.id
from Users as user, user.connections[] as connection
```

- create index midx2 on Users (address.phones[].area, income)

Creates an index on the area codes and income of users. The contents of this index are the results of the following query:

```
select distinct phone.area, u.income, u.id
from Users as u, u.address.phones[] as phone
```

-
  ```
  create index midx3 on Users
  (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 are the results of the following query:

```
select distinct phone.area, phone.kind, u.income, u.id
from Users as u, u.address.phones[] as phone
```

-
  ```
  create index midx4 on nodes (expenses.keys(), expenses.values())
  ```

  Creates an index on the fields (both keys and values) of the expenses map. The contents of this index are the results of the following query:

```
select distinct $mapEntry.key, $mapEntry.value, $u.id
from Users as $u,
   (select { “key” : $key, “value” : $u.expenses.$key }
    from $u.expenses.keys() as $key) as $mapEntry
```

5.2 Drop index Statement

```
drop_index_statement :
  DROP INDEX (IF EXISTS)? index_name ON table_name ;
```

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.

5.3 Using indexes for query optimization

In Oracle NoSQL, 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, 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 (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 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.

5.3.1 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. 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 consider an index applicable to a query 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 to the entries within the subrange being scanned. 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 indexes from sections 5.1.2 and 5.1.4.

Q1.

```
select *
```

```
from Users
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.

Q2.

```
select *
from Users
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.

Q3.

```
select *
from Users
where 20 < income or age > 70
```

There is no index predicate in this case, because no index has information about user ages.

Q4.

```
select *
from Users 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.

Q5.

```
select id, 2*income
from Users 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 query Q5. The query processor will apply this optimization.

Q6.

```
select *
from Users 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.

Q7.

```
select *
from Users 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.

Q8.

```
declare
$city string;
select *
from Users u
where u.address.state = “CA” and
    u.address.city = $city and
    (u.income > 50 or (10 < income and 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).

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>
- op is a comparison operator
- <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.

Q9.

```
select *
from users u
where u.connections[] =any 10
```

midx1 is applicable and the predicate is both a start and a stop predicate.

Q10.

```
select *
from users 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.

Q11.

```
select *
from users u
where u.connections[] >any 10
```

midx1 is applicable and the predicate is a start predicate.

Q12.

```
select id
from users 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 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 Q11, 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.

To search for users who have a connection in the range between 10 and 100, the following query can be used:

```
select id
from users u
where exist u.connections[10 < $element and $element < 100]
```

The result of this query is user 500 only and both predicates can be used as index predicates (start and stop) . However, in the current implementation, the query processor cannot detect that mdx1 is applicable to the above query.

Q13.

```
 select *
from Users u
where u.address.phones.area =any 650 and
     u.address.phones.kind =any work and
     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. Only the area code predicate can be used as a start/stop predicate and the income predicate as a filtering one. Given that no predicate can be pushed to the address.phones.kind index path, midx3 is not really useful for queries in the current SQL version (midx3 can still be "queried" directly, via programmatic APIs). Indexes idx1, idx2, and midx2 are also applicable in Q11.

Q14.

```
select *
from Users
where 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.

Q15.

```
select *
from Nodes
where expenses.travel = 1000 and 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 most likely more selective than the greater-than predicate and the query processor will use that.

5.3.2 Choosing the best applicable index

As mentioned already, to choose an index for a query, the query processor uses a simple heuristic together with any user-provided index hints. 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). However, if the query has an order by and the forced index is not the sorting index, an error will be thrown. 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.
- If the query has a complete shard key, the primary index is used. Using the primary index in this case implies that a single table partition (in a single shard) will be scanned (because all the qualifying rows will be in that single partition). Using any other index in this case would require sending the query to all the shards and potentially scanning a lot of data on those shard for no good reason. However, if the query has an order by and the primary index is not the sorting index, an error will be thrown.
- If the query has an order by and the previous bullets do not apply, the sorting index is used, even if other indexes may be more selective.
- If none of the previous bullets apply, 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.

The FORCE_INDEX and PREFER_INDEXES hints specified 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:

```
select_clause :
  SELECT hints? ( STAR |
                (expr col_alias (COMMA expr col_alias)*) ) ;

hints : '/*+' hint* '*/' ;

hint : ( (PREFER_INDEXES LP name_path index_name* RP) |
        (FORCE_INDEX    LP name_path index_name  RP) |
        (PREFER_PRIMARY_INDEX LP name_path RP)       |
        (FORCE_PRIMARY_INDEX  LP name_path RP) ) STRING?;
```

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.

6 Appendix : The full SQL grammar

```
program :
   (
   query
  | create_table_statement
  | alter_table_statement
  | drop_table_statement
  | create_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
  | describe_statement
  | show_statement)
   EOF
  ;
```

```
query : var_decls? sfw_expr ;

var_decls : DECLARE var_decl SEMICOLON (var_decl SEMICOLON)*;

var_decl : var_name type_def;

var_name : DOLLAR id ;

expr : or_expr ;

sfw_expr :
  select_clause
  from_clause
  where_clause?
  orderby_clause?
  limit_clause?
  offset_clause? ;

from_clause : FROM table_name tab_alias? ;

table_name : name_path ;

tab_alias : AS? DOLLAR? Id

where_clause : WHERE expr ;

select_clause :
  SELECT hints? ( STAR |
                (expr col_alias (COMMA expr col_alias)*) ) ;

hints : '/*+' hint* '*/' ;

hint : ( (PREFER_INDEXES LP name_path index_name* RP) |
       (FORCE_INDEX    LP name_path index_name  RP) |
       (PREFER_PRIMARY_INDEX LP name_path RP)       |
       (FORCE_PRIMARY_INDEX  LP name_path RP) ) STRING?;

col_alias : AS id ;

orderby_clause :
  ORDER BY expr sort_spec (COMMA expr sort_spec)* ;
```

```
sort_spec : (ASC | DESC)? (NULLS (FIRST | LAST))? ;

limit_clause : LIMIT add_expr ;

offset_clause : OFFSET add_expr ;

or_expr : and_expr | or_expr OR and_expr ;

and_expr : not_expr | and_expr AND not_expr ;

not_expr : NOT? cond_expr ;

cond_expr : comp_expr | exists_expr | is_of_type_expr ;

comp_expr : add_expr ((val_comp_op | any_comp_op) add_expr)? ;

val_comp_op : "=" | "!=" | ">" | ">=" | "<" | "<=" ;

any_comp_op :
  "=any" | "!=any" | ">any" | ">=any" | "<any" | "<=any" ;

exists_expr : EXISTS add_expr ;

is_of_type_expr :
  add_expr IS NOT? OF TYPE?
  LP ONLY? sequence_type (COMMA ONLY? sequence_type)* RP;

sequence_type : type_def quantifier? ;

quantifier : STAR | PLUS | QUESTION ;

add_expr : multiply_expr ((PLUS | MINUS) multiply_expr)* ;

multiply_expr : unary_expr ((STAR | DIV) unary_expr)* ;

unary_expr : path_expr | (PLUS | MINUS) unary_expr ;

path_expr :
  primary_expr (map_step | array_step)* ;

map_step : map_field_step | map_filter_step ;

map_field_step : DOT ( id | string | var_ref | parenthesized_expr | func_call );
```

```
map_filter_step : DOT (KEYS | VALUES) LP expr? RP ;

array_step : array_filter_step | array_slice_step;

array_filter_step : LBRACK expr? RBRACK ;

array_slice_step : LBRACK expr? COLON expr? RBRACK ;

primary_expr :
  const_expr |
  column_ref |
  var_ref |
  array_constructor |
  map_constructor |
  case_expr |
  cast_expr |
  func_call |
  parenthesized_expr ;

const_expr : INT_CONST | FLOAT_CONST | string | TRUE | FALSE | NULL;

column_ref : id (DOT id)? ;

var_ref : DOLLAR id? ;

array_constructor : LBRACK expr (COMMA expr)* RBRACK ;

map_constructor :
  (LBRACE expr COLON expr (COMMA expr COLON expr)* RBRACE) |
  (LBRACE RBRACE) ;

case_expr :
  CASE WHEN expr THEN expr (WHEN expr THEN expr)* (ELSE expr)? END;

cast_expr : CAST LP expr AS sequence_type RP ;

func_call : id LPAREN (expr (COMMA expr)*)? RPAREN ;

parenthesized_expr : LPAREN expr RPAREN;

create_table_statement :
  CREATE TABLE (IF NOT EXISTS)? table_name comment?
  LPAREN table_def RPAREN ;

table_name : name_path;

table_def : (field_def | key_def) (COMMA (field_def | key_def))* ;

key_def :
  PRIMARY KEY
  LPAREN (shard_key_def COMMA?)? id_list_with_size? RPAREN ttl_def? ;

id_list_with_size : id_with_size (COMMA id_with_size)* ;

id_with_size : id storage_size? ;

storage_size : LPAREN INT_CONST RPAREN ;

shard_key_def : SHARD LPAREN id_list_with_size RPAREN;

ttl_def : USING TTL INT_CONST (HOURS | DAYS) ;

drop_table_statement : DROP TABLE (IF EXISTS)? name_path ;

alter_table_statement :
  ALTER TABLE name_path (alter_field_statements | ttl_def);

alter_field_statements :
  LPAREN alter_field_stmt (COMMA alter_field_stmt)* RPAREN ;

alter_field_stmt :add_field_stmt | drop_field_stmt ;

add_field_stmt : ADD schema_path type_def default_def? comment? ;

drop_field_stmt : DROP schema_path ;

schema_path : schema_path_step (DOT schema_path_step)*;

schema_path_step : id (LBRACK RBRACK)*;

create_index_statement :
    CREATE INDEX (IF NOT EXISTS)?
    index_name ON table_name LPAREN path_list RPAREN comment?;

index_name : id ;
```

```
path_list : index_path (COMMA index_path)* ;

index_path : (name_path | keys_expr | values_expr | brackets_expr) ;

name_path : id (DOT id)* ;

keys_expr : name_path DOT KEYS LP RP ;

values_expr : name_path DOT VALUES LP RP (DOT name_path)?;

brackets_expr : name_path LBRACK RBRACK (DOT name_path)?

type_def :
  INTEGER |
  LONG |
  FLOAT |
  DOUBLE |
  STRING |
  timestamp_def |
  enum_def |
  binary_def |
  BOOLEAN |
  record_def |
  array_def |
  map_def |
  ANY |
  JSON |
  ANYRECORD |
  ANYATOMIC |
  ANYJSONATOMIC
  ;

timestamp_def : TIMESTAMP (LP INT_CONST RP)? ;

enum_def : ENUM LPAREN id_list RPAREN ;

binary_def : BINARY (LPAREN INT_CONST RPAREN)? ;

map_def : MAP LPAREN type_def RPAREN ;

array_def : ARRAY LPAREN type_def RPAREN ;
```

```
record_def : RECORD LPAREN field_def (COMMA field_def)* RPAREN ;

field_def : id type_def default_def? comment? ;

default_def :
  (default_value (NOT NULL)?) | (NOT NULL default_value?) ;

comment : COMMENT string ;

default_value : DEFAULT (number | string | TRUE | FALSE | id) ;

number : MINUS? (FLOAT_CONST | INT_CONST) ;

string : STRING_CONST | DSTRING_CONST ;

id_list : id (COMMA id)* ;

id : ID |
     ADD | ALTER | AND | ANY | ANYATOMIC | ANYJSONATOMIC |
     ANYRECORD | ARRAY | AS | ASC | BINARY | BOOLEAN | BY |
     CASE | CAST | COMMENT | CREATE |
     DECLARE | DEFAULT | DESC | DOUBLE | DROP |
     ELSE | END | EXISTS | FIRST | FLOAT | FROM | ENUM |
     IF | INDEX | INTEGER | IS | JSON | KEY | KEYS |
     LAST | LIMIT | LONG | MAP | NOT | NULLS | OF | OFFSET | ON |
     OR | ORDER | PRIMARY | RECORD | SELECT | SHARD | STRING |
     TABLE | THEN | TYPE | VALUES | WHEN | WHERE;

ID : ALPHA (ALPHA | DIGIT | '_')* ;


fragment ALPHA : 'a'..'z'|'A'..'Z' ;

fragment DIGIT : '0'..'9' ;

INT_CONST : DIGIT+ ;

FLOAT_CONST : ( DIGIT* '.' DIGIT+ ([Ee] [+-]? DIGIT+)? ) |
              ( DIGIT+ [Ee] [+-]? DIGIT+ ) ;

STRING_CONST : '\'' ((ESC) | .)*? '\'' ; // string with single quotes

DSTRING_CONST : '"' ((ESC) | .)*? '"' ;  // string with double quotes
```

```
fragment ESC : '\\' ([\'\\/bfnrt] | UNICODE) ;

fragment DSTR_ESC : '\\' (["\\/bfnrt] | UNICODE) ;

fragment UNICODE : 'u' HEX HEX HEX HEX ;

TRUE : [Tt][Rr][Uu][Ee] ;

FALSE : [Ff][Aa][Ll][Ss][Ee] ;

NULL : [Nn][Uu][Ll][Ll] ;
```