Oracle

SQL for Oracle NoSQL Database Specification

12c Release 2

Library Version 12.2.4.5

Legal Notice

Copyright © 2011 - 2017 Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Published 6/19/2017

Table of Contents

1 Introduction......6
1.1 Antlr meta-syntax......7
1.2 Comments......7
1.3 Identifiers and literals......8
1.4 SQL grammar......9
1.5 Operator precedence......9
1.6 Reserved words......9
2 Data Model......9
2.1 Atomic values......10
2.2 Complex values......10
2.3 Atomic types......11
2.4 Complex types......11
2.5 Wildcard types and JSON data......12
2.6 Type hierarchy......13
2.7 Tables......15
2.8 Type definitions......15
3 Creating and managing tables......16
3.1 Create Table Statement......16
3.1.1 Example......18
3.2 DROP TABLE Statement......19
3.3 ALTER TABLE Statement......19
4 Querying data with SQL: The Query Statement......20
4.1 Expressions, sequences, and sequence type......20
4.2 External variable declarations......22
4.2.1 Example:......23
4.3 Select-From-Where (SFW) expression......23
4.3.1 FROM clause......24
4.3.2 WHERE Clause......24
4.3.3 ORDER BY clause......25
4.3.4 SELECT Clause......26
4.3.5 OFFSET and LIMIT clauses......26
4.3.6 Examples......27
4.4 Path expressions......27
4.4.1 Field step expressions......28
4.4.2 Examples......29
4.4.3 Map-filter step expressions......30
4.4.4 Examples......30
4.4.5 Array-filter step expressions......31
4.4.6 Examples:......32
4.4.7 Array-slice step expressions......32
4.4.8 Examples:......33
4.5 Logical operators: AND, OR, and NOT......33
4.5.1 Example:......34
4.6 IS NULL and IS NOT NULL operators......34
4.6.1 Example:......34
4.7 Value comparison operators......34
4.7.1 Example......36

4.8 Sequence comparison operators....36
4.8.1 Examples....37
4.9 Exists operator....38
4.9.1 Examples....38
4.10 Is-Of-Type operator....38
4.10.1 Example....39
4.11 Arithmetic expressions....39
4.11.1 Example....40
4.12 Primary expressions....40
4.13 Parenthesized expressions....40
4.14 Constant expressions....41
4.15 Column references....41
4.16 Variable references....42
4.17 Array and map constructors....42
4.17.1 Example....42
4.18 Searched case expression....43
4.18.1 Example....43
4.19 Cast expression....43
4.20 Extract expression....45
4.21 Function calls....46
5 Updating data with SQL: The update statement....46
5.1 The SET clause....48
5.2 The ADD clause....49
5.3 The PUT clause....49
5.4 The REMOVE clause....49
5.5 The SET TTL clause....50
5.6 Examples....50
5.6.1 Example 1....50
5.6.2 Example 2....52
5.6.3 Example 3....53
6 Indexing in Oracle NoSQL....54
6.1 Create Index Statement....54
6.1.1 Simple indexes....58
6.1.2 Simple index examples....58
6.1.3 Multi-key indexes....59
6.1.4 Multi-key index examples....61
6.2 Indexing JSON data....62
6.2.1 Simple typed json indexes....64
6.2.2 Multi-key typed json indexes....66
6.3 Drop index Statement....68
6.4 Using indexes for query optimization....68
6.4.1 Finding applicable indexes....69
Example 1....70
Example 2....70
Example 3....70
Example 4....70
Example 5....71
Example 6....71
Example 7....71
Example 8....71

Example 9 72
Example 10 72
Example 11 72
Example 12 73
Example 13 73
Example 14 73
Example 15 74
Example 16 74
Example 17 74
6.4.2 Choosing the best applicable index 75
7 Built-in functions 76
7.1 Functions on sequences 76
7.2 Functions on complex values 76
7.3 Functions on timestamps 76
7.4 Functions on rows 77
7.5 Miscellaneous Functions 78
8 Appendix : The full SQL grammar 78

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

```
program :
   (
   query
 | update_statement
 | 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
```

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

```
  | alter_user_statement
  | grant_statement
  | revoke_statement
  | describe_statement
  | show_statement)
    EOF
  ;
```

This document is concerned with the first 7 statements in the above list, that is, with read-only queries, update 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 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, array_filter_step : LBRACK expr RBRACK says that a array_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 6.4.2).

- // 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. Section 4.14 describes how the tokens listed below are translated into instances of the data model (which is described in section 2).

```
INT_CONST : DIGIT+ ;

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

NUMBER_CONST : (INT_CONST | FLOAT_CONST) [Nn]
```

[2] id is actually a non-terminal, but most of the "work" is done by the underlying ID terminal; see section 8 for the full grammar.

```
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 8) at the end of this document.

1.5 Operator precedence

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

```
or_expr : and_expr | or_expr OR and_expr ;

and_expr : not_expr | and_expr AND not_expr ;

not_expr : NOT? is_null_expr ;
```

1.6 Reserved words

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

2 Data Model

This section defines the Oracle NoSQL data model in abstract terms. The Oracle NoSQL java 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 : all floating-point numbers that can be represented by the java float data type.
- Doubles : all floating-point numbers that can be represented by the java double data type.
- Numbers : all numbers that can be represented by the java BigDecimal data type.
- 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***. Field values can have different types. Furthermore, records may contain fields with NULL as their value. Contrary to maps and arrays, it is not possible to add or remove fields from a record value, because as we will see below, the number of fields and their field names are part of the record type definition associated with a record 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
- Number : All number 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 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 why 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, double, or number 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. In general, the end result of this parsing is a tree of maps, arrays, and atomic values. For persistent storage, the tree is serialized into a binary format.

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

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 are 2 exceptions 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 are 2 exceptions to the subtype-substitution rule. The first one concerns arrays and maps that may appear inside json data. 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 that would still keep it a json document.

The second exception concerns numeric values. Specifically, Double and Float appear as subtypes of Number. However, Double and Float include 3 special values in their domain: NaN (not a number), positive infinity, and negative infinity. These values are not in the domain of Number. As a result, an operation that expects Number value will also work with Double/Float values as long as these values are not one of 3 special values; otherwise an error will be raised.

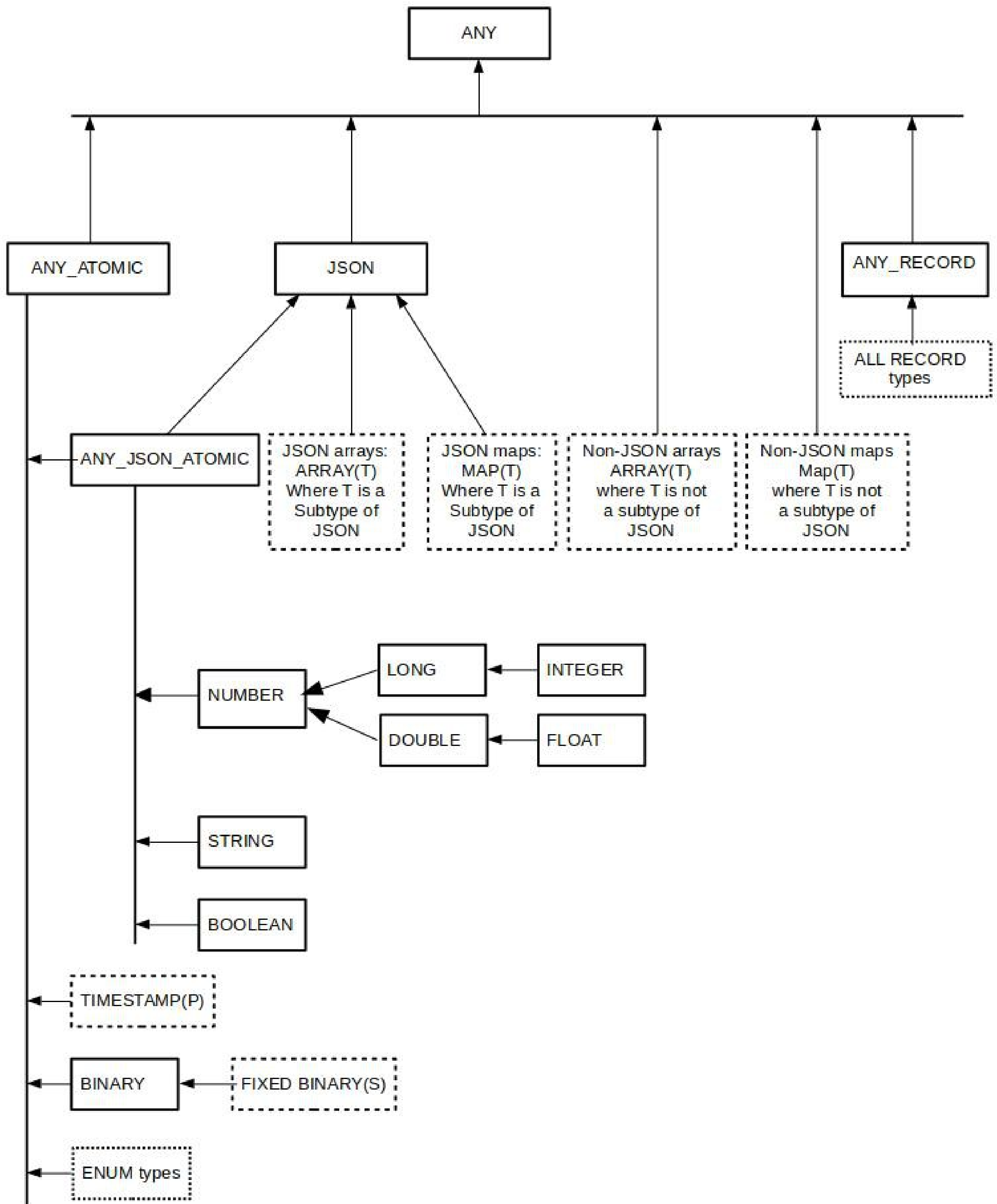

Figure 1: SQL type hierarchy

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 |
  NUMBER |
  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 | NUMBER_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 atomic 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 Time-To-Live (TTL) value to use in computing the expiration time of a row when the row is inserted in the table, if a specific TTL value is not provided via the row insertion API. The expiration time of a row is the point in time when the row will “expire”. Expired rows are not included in query results and are eventually removed from the table automatically by Oracle NoSQL. The expiration time of a row is computed in terms of a TTL value, which is a non-negative integer specifying a number of days or hours. Specifically, for a TTL value of N hours/days, the expiration time is the current time (in UTC) plus N hours/days, rounded up to the next full hour/day. For example, if the current time is 2017-06-01T10:05:30.0 and the TTL is 3 hours, the expiration time will be 2017-06-01T14:00:00.0. Zero can be used as a special TTL value to specify that a row should not expire (infinite expiration time). If the CREATE TABLE statement has no TTL specification, the default table TTL is zero.

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 number, or JNULL.

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 Querying data with SQL: 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, all expressions operate on zero or more input sequences and produce a sequence as their result. A sequence is simply a set of zero or more items (including NULLs). Although, in general, Oracle NoSQL expressions work on sequences and produce sequences, many of them place restrictions on the cardinality of the sequences they accept and/or produce. For example, several expressions are scalar: they require that their input sequence(s) contain no more than one item and they never produce a sequence of more than one item. Notice that a single item is considered equivalent to a sequence containing 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 4.17). This is called **boxing** the sequence. Conversely, there are expressions that **unbox** an array: they select all or a subset of the items contained in the array and return these items as a sequence. There is no implicit unboxing of arrays; an expression must always be applied to do the unboxing. In most cases, sequence boxing must also be done explicitly, that is, the query writer must use an array constructor. There are, however, a couple of cases where boxing is done implicitly, that is, an expression (which is not an array constructor) will convert an input sequence to an array.

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 an expression must have (sequence) type T, what we mean is that when the expression is evaluated, the result sequence must have type T or any subtype of T. Similarly, the usual subtype-substitution rules 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 select_list ;

select_list :
  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 4.15, 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, taking into account the sort_spec associated with each order-by expression.

Each order-by expression must have type AnyAtomic?. If an order-by expression returns an empty sequence, the special (internal) value EMPTY is used as the returned value. A sort_spec specifies the "direction" of the sort (ascending or descending) and how to compare the ***special values*** NULL, JNULL, and EMPTY with the non-special values. If NULLS LAST is specified and the direction is ASC/DESC, the special values are considered greater/less than all non-special values. If NULLS FIRST is specified and the direction is ASC/DESC, the special values are considered less/greater than

all non-special values. The relative ordering among the 3 special values themselves is fixed: if the direction is ASC, the ordering is EMPTY < JNULL < NULL; otherwise the ordering is reversed.

Notice that in the grammar, sort_spes are optional. If no sort_spec is given, the default is ASC order and NULLS LAST. If only the sort order is specified, then NULLS LAST is used if the order is ASC, otherwise NULLS FIRST. If the sort order is not specified, ASC is used.

Taking into account the above rules, the relative order between any two input rows is determined as follows. Let N be the number of order-by expressions and let Vi1, Vi2, … ViN be the atomic values (including EMPTY) returned by evaluating these expressions, from left to right, on a row Ri. 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
- the m-th sort_spec specifies ascending order and Vim is less than Vjm, or
- the m-th sort_spec specifies descending order and Vim is greater than Vjm

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

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 an index (see section 6.1) can be found at compile time 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 and and the i-th index field is not a multi-key one (see section 6.1.3); the later condition guarantees, at compile time, that the order-by expression will not return more than 1 atomic item during query execition. Furthermore, all the sort_specs must specify the same ordering direction and for each sort_spec, the desired ordering with respect to the special values must match the way these values are sorted by the index. In the current implementation, the special values are always sorted last in an index. So, if the sort order is ASC, all sort_specs must specify NULL LAST, and if the sort order is DESC, all sort_specs must specify NULLS FIRST.

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 value must be exactly one item. To achieve this, the following two ***implicit conversions*** are employed: (a) If the result of a field expression is empty, NULL is used as the value of the corresponding field in the created record. (b) If the compiler determines that a field expression may return more than one items, it warps the field expression with a ***conditional array constructor*** (see section 4.17). During runtime, an array will be constructed only if the field expression does actually return more than one item; if so, the returned items will be inserted into the constructed array, which will then be used as the value of the corresponding field in the created record.

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

The SELECT clause 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 6.4.2.

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 4.15 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 : DOT (map_filter_step | map_field_step) ;

map_field_step :
   id | string | var_ref | parenthesized_expr | func_call ;

map_filter_step : (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 4.12), 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 :
  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 4.16, 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).

- 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
```

[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 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 : (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.

- If the context item is an array, the step iterates over the array elements and computes the predicate expression on each element. In addition to the context-item variable ($), the predicate expression may reference the following two implicitly-declared variables: ***$element*** is bound to the ***context element***, i.e., the current element in $, and ***$pos*** is bound to the position of the context element within the array (positions are counted starting with 0). The predicate expression must return a boolean item, or a numeric item, or the empty sequence, or NULL. A NULL or an empty result from the predicate expression is treated as a false value. If the predicate result is true/false, the context element is selected/skipped, respectively. If the predicate result is a number P, the context element is selected only if the condition $pos = P is true. Notice that this implies that if P is negative or greater or equal to the array size, the context element is skipped.

4.4.6 Examples:

- 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 4.17). 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 lastName,
    [ u.address.phones[$element.area = 650].number ] as
    phoneNumbers
from Users u
```

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

```
select lastName,
    [ u.address.phones[$element.area = $[0].area].number ]
from Users u
```

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

```
select [ connections[$element > 100 and $pos < 10] ] as
    interestingConnections
from Users
```

4.4.7 Array-slice step expressions

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

Array slice steps are meant to be used primarily with arrays. In general, an array slice step selects elements of arrays based only on the element positions. The elements to select are the ones whose positions are within a range between a "low" position and a "high" position. The low and high positions are computed by two ***boundary expressions***: a "low" expression for the low position and a "high" expression for the high position. Each boundary expression must return at most one item of type LONG or INTEGER, or NULL. The low and/or the high expression may be missing. The context-item variable ($) is available during the computation of the boundary expressions.

An array filter step processes each context item as follows:

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

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

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

4.4.8 Examples:

- Select the strongest connection of the user with id 10.

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

- For user 10, select his/her 5 strongest connections (i.e. the first 5 ids in the "connections" array). Notice that the slice expression will return at most 5 ids; if user 10 has fewer that 5 connections, all of his/her connections will be returned.

```
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

```
or_expr : and_expr | or_expr OR and_expr ;

and_expr : not_expr | and_expr AND not_expr ;

not_expr : NOT? is_null_expr ;
```

```
is_null_expr : cond_expr (IS NOT? NULL)? ;
```

```
cond_expr : comp_expr | exists_expr | is_of_type_expr ;
```

The binary *and* and *or* operators and the unary *not* operator have the usual semantics. Their operands are conditional expressions, which must have type BOOLEAN?. An empty result from an operand is treated as the false value. If an operand returns NULL then:

- The *and* operator returns false if the other operand returns false; otherwise, it returns NULL.
- The *or* operator returns true if the other operand returns true; otherwise it returns NULL.
- The *not* operator returns NULL.

4.5.1 Example:

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

```
select id, lastName
from Users
where 30 <= age and age <= 40 or income > 100000
```

4.6 IS NULL and IS NOT NULL operators

```
is_null_expr : cond_expr (IS NOT? NULL)? ;
```

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

4.6.1 Example:

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

```
select id, lastName
from Users u
where u.income IS NULL
```

4.7 Value comparison operators

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

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

Value comparison operators are primarily used to compare 2 values, one produced by the left operand and another from the right operand (this is in contrast to the *any* comparisons, defined in the next section, which compare two sequences of values). If any operand returns more than one item, an error is raised. If both operands return the empty sequence, the operands are considered equal (so true will be returned if the operator is =, <=, or >=). If only one of the operands returns empty, the result of the comparison is false unless the operator is !=.

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

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

- A numeric item is comparable with any other numeric item. If an integer or long value is compared to a float or double value, the integer/long will first be cast to float/double. If one of the operands is a number value, the other operand will first be cast to number (if not a number already).

- A string item is comparable to another string item (using the java String.compareTo() method). A string item is also comparable to an enum item. In this case, before the comparison the string is cast to an enum item in the type of the other enum item. Such a cast is possible only if the enum type contains a token whose string value is equal to the source string. If the cast is successful, the two enum items are then compared as explained below; otherwise, the two items are incomparable and false is returned.

- Two enum items are comparable only if they belong to the same type. If so, the comparison is done on the ordinal numbers of the two enums (not their string values). As mentioned above, an enum item is also comparable to a string item, by casting the string to an enum item.

- Binary and fixed binary items are comparable with each other for equality only. The 2 values are equal if their byte sequences have the same length and are equal byte-per-byte.

- A boolean item is comparable with another boolean item, using the java Boolean.compareTo() method.

- A timestamp item is comparable to another timestamp item, even if their precisions are different..

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

The semantics of comparisons among complex items are defined in a recursive fashion. Specifically:

- A record is comparable with another record for equality only and only if they contain comparable values. More specifically, to be equal, the 2 records must have equal sizes (number of fields) and for each field in the first record, there must exist a field in the other record such that the two fields are at the same position within their containing records, have equal field names, and equal values.

- A map is comparable with another map for equality only and only if they contain comparable values. 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.

4.7.1 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 4.17) 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]
```

4.8 Sequence comparison operators

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

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

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

4.8.1 Examples

- 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).

```
select id, lastName, address
from Users
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 id of each user who is connected with a user having id greater than 10 and is also connected with a user having id less than 100.

```
select id
from Users u
where 10 <any u.connections[] and u.connections[] <any 100
```

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

How can the second query be written in SQL for OracleNoSQL? By a combination of an EXISTS operator (to be defined in the next section) and an array filtering step:

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

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

```
select id
from Users u
where exists u.connections[10 < $element] and
```

```
exists u.connections[$element < 100]
```

- 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
```

4.9 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.

4.9.1 Examples

- 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
```

Notice that EXISTS will return true if its input expression returns NULL. As a result, EXISTS is not useful for checking the "existence" of fields inside strongly typed data. For example, in the following query, the EXIST predicate will always be true, even for rows where the value of the income column is NULL. This implies that in SQL for Oracle NoSQL, the interpretation of NULL is that of an unknown value, rather than an absent value.

```
select id
from Users u
where exists u.income
```

4.10 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.

4.10.1 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)
```

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

4.11.1 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
```

4.12 Primary expressions

```
primary_expr :
  parenthesized_expr ;
  const_expr |
  column_ref |
  var_ref |
  array_constructor |
  map_constructor |
```

```
  case_expr |
  cast_expr |
  extract_expr |
  func_call ;
```

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

4.13 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
```

4.14 Constant expressions

```
const_expr : number | string | TRUE | FALSE | NULL ;

number : '-'? (INT_CONST | FLOAT_CONST | NUMBER_CONST);

string : STRING_CONST | DSTRING_CONST ;
```

The syntax for INT_CONST, FLOAT_CONST, NUMBER_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 literal are sequences of unicode characters enclosed in double or single quotes. String literals are translated into String items. Notice that any escape sequences appearing in a string literal will be converted to their corresponding character inside the corresponding String item.
- Integer literals are sequences of one or more digits. Integer literals are translated into Integer items, if their value fits in 4 bytes, into Long items, if they fit in 8 bytes, otherwise to Number items.
- Floating point literals represent of real numbers using decimal notation and/or exponent. Floating-point literals are translated into Double items, if possible, otherwise to Number items.
- Number literals are integer or floating-point literals followed by the ‘n’ or ‘N’ character. Number literals are always translated into Number items.
- The TRUE and FALSE literals are translated to the boolean true and false items, respectively..

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

4.15 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.

4.16 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.

4.17 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).

4.17.1 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
```

4.18 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.

4.18.1 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 4.17.1. 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
```

4.19 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 according to the following (recursive) rules.

- 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 other than JSON and the type of the input item is a subtype of the wildcard type, the cast is a noop; otherwise an error is raised.

- If the target type is JSON, then (a) an error is raised if the input item is has a non-json atomic type, else (b) if the input item has a type that is a json atomic type or ARRAY(JSON) or MAP(JSON), the cast is a noop , else (c) if the input item is a non-json array, a new array of type ARRAY(JSON) is constructed, each element of the input array is cast to JSON, and the resulting item is appended into the new json array, else (d) if the input item is a non-json map, a new map of type MAP(JSON) is constructed, each field value of the input map is cast to JSON, and resulting item together with the associated field name are inserted into the new json map, else (e) if the input item is a record, it is cast to a map of type MAP(JSON) as described below.

- If the target type is an array type, an error is raised if the input item is not an array. Otherwise, a new array is created, whose type is the target type, each element in the input array is cast to the element type of the target array, and the resulting item is appended into the new array.

- If the target type is a map type, an error is raised if the input item is not a map or a record. Otherwise, a new map is created, whose type is the target type, each field value in the input map/record is cast to

the value type of the target map, and the resulting field value together with the associated field name are inserted to the new map.

- If the target type is a record type, an error is raised if the input item is not a record or a map. Otherwise, a new record is created, whose type is the target type. If the input item is a record, its type must have the same fields and in the same order as the target type. In this case, each field value in the input record is cast to the value type of the corresponding field in the target type and the resulting field value together with the associated field name are added to the new record. If the input item is a map, then for each map field, if the field name exists in the target type, the associated field value is cast to the value type of the corresponding field in the target type and the resulting field value together with the associated field name are added to the new record. Any fields in the new record whose names do not appear in the input map have their associated field values set to their default values.

- If the target type is string, the input item may be of any type. In other words, every item can be cast to a string. For complex items their "string value" is a json-text representation of their value. For timestamps, their string value is in UTC and has the format "uuuu-MM-dd['T'HH:mm:ss]". For binary items, their string value is a base64 encoding of their bytes.

- If the target type is an atomic type other than string, the input item must also be atomic. Among atomic items and types the following casts are allowed:

  - Every numeric item can be cast to every other numeric type. The cast is done as in Java.

  - String items may be castable to all other atomic types. Whether the cast succeeds or not depends on whether the actual string value can be parsed into a value that belongs to the domain of the target type.

  - Timestamp items are castable to all the timestamp types. If the target type has a smaller precision that the input item, the resulting timestamp is the one closest to the input timestamp in the target precision. For example, consider the following 2 timestamps with precision 3: 2016-11-01T10:00:00.236 and 2016-11-01T10:00:00.267. The result of casting these timestamps to precision 1 is: 2016-11-01T10:00:00.2 and 2016-11-01T10:00:00.3, respectively.

Example

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))
```

4.20 Extract expression

```
extract _expr : EXTRACT LP id FROM expr RP ;
```

The extract expression extract a component from a timestamp. Specifically, the expression after the FROM keyword must return at most one timestamp or NULL. If the result of this expression is NULL or empty, the results of EXTRACT is also NULL or empty, respectively. Otherwise, the component specified by the id is returned. This id must be one of the following keywords: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, MILLISECOND, MICROSECOND, NANOSECOND, WEEK, or ISOWEEK. The definitions of this components are listed below:

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

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

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

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

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

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

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

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

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

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

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

As a convenience, SQL for OracleNoSQL also includes specific built-in functions to extract each of the above components from a time stamp. For example, EXTRACT(YEAR from expr) is equivalent to year(expr). These and other built-in functions are described in section 7.

4.21 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 list of currently available functions is given in section 7.

5 Updating data with SQL: The update statement

```
update_statement :
  prolog?
  UPDATE table_name AS? tab_alias?
       update_clause (COMMA update_clause)*
  WHERE expr
  returning_clause? ;

returning_clause : RETURNING select_list ;

update_clause :
  (SET set_clause (COMMA (update_clause | set_clause))*) |
  (ADD add_clause (COMMA (update_clause | add_clause))*) |
  (PUT put_clause (COMMA (update_clause | put_clause))*) |
  (REMOVE remove_clause (COMMA remove_clause)*)
  (SET TTL ttl_clause (COMMA update_clause)*) ;

set_clause : target_expr EQ expr ;

add_clause : target_expr pos_expr? expr ;

put_clause : target_expr expr ;

remove_clause : target_expr ;

ttl_clause : (add_expr (HOURS | DAYS)) | (USING TABLE DEFAULT) ;

target_expr : path_expr ;

pos_expr : add_expr ;
```

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

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

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

There are 5 kinds of update clauses:

- SET : Updates the value of one or more existing fields.
- ADD : Adds new elements in one or more arrays.
- PUT : Adds new fields in one or more maps. It may also update the values of existing map fields.
- REMOVE: Removes elements/fields from one or more arrays/maps.
- SET TTL: Updates the expiration time of the row.

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

The SET, ADD, PUT, and REMOVE clauses start with a target expression, which computes the items to be updated or removed. In all cases, the target expression must be either a top-level column reference
of a path expression starting with the table alias. If the target expression returns nothing, the update

clause is a noop. The following sub-sections describe the update clauses in more detail. Examples are shown in section 5.6

5.1 The SET clause

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

If the target expression returns a NULL item, then either the target item itself is the NULL item, or one of its ancestors is NULL. In the former case, the target item will be replaced by the new item. In the later case the SET is a noop.

The new-value expression may return zero or more items. If it returns an empty result, the SET is a noop. If it returns more than one item, the items are enclosed inside a newly constructed array (this is the same as the way the SELECT clause treats multi-valued expressions in the select list). So, effectively, the result of the new-value expression contains at most one item. This new item is then cast to the type expected by the parent item for the target field. This cast behaves like the cast expression described in section 4.19. If the cast fails, an error is raised; otherwise the new item replaces the target item within the parent item.

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

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

5.2 The ADD clause

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

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

If the the new-values expression returns nothing, the ADD is a noop. Otherwise, each item returned by this expression is cast to element type of the array. An error is raised if any of these casts fails. Otherwise, the new elements are inserted into the target array, as described below.

If the position expression is missing, or if it returns an empty result, the new elements are appended at the end of the target array. An error is raised if the position expression returns more than one items or a

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

5.3 The PUT clause

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

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

If the the new-fields expression returns nothing, the PUT is a noop. Otherwise, for each item returned by the new-fields expression, if the item is not a map or a record, it is skipped, else, the fields of the map/record are "merged" into the current target map. This merge operation will insert a new field into the target map, if the target map does not already have a field with the same key; otherwise it will set the value of the target field to the value of the new field.

5.4 The REMOVE clause

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

5.5 The SET TTL clause

As described in section 3.1, every table row has an expiration time, which may be infinite, and which is computed in terms of a Time-To-Live (TTL) value that is specified as a number of days or hours. This computation is done when the row is first inserted in a table. The SET TTL clause computes a new expiration time, using a potentially different TTL value, and replaces the current expiration time of the row with the newly compute time.

As shown in the syntax, the SET TTL clause comes in two flavors. The first contains an expression, which computes a new TTL value. If the result of this expression is empty, the SET TTL clause is a noop. Otherwise, the expression must return a single numeric item, which is cast to an integer N. If N is negative, it is set to 0, which is the special TTL value indicating that the row should not expire. To the right of the TTL expression, the keyword HOURS or DAYS must be used to specify whether N is a number of hours or days, respectively. With the second TTL flavor, the TTL value N is set to table default TTL that was specified in the CREATE TABLE statement (see section 3.1).

In both cases, for a TTL value of N hours/days, where N is greater than zero, the expiration time is computed as the current time (in UTC) plus N hours/days, rounded up to the next full hour/day. For example, if the current time is 2017-06-01T10:05:30.0 and N is 3 hours, the expiration time will be 2017-06-01T14:00:00.0.

5.6 Examples

5.6.1 Example 1

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

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

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

```
update People p
set p.info.profession = "surfing instructor",
set p.info.address.city = "Santa Cruz",
set p.info.income = p.info.income / 10,
```

```
set p.info.children.values().age = $ + 1
add p.info.address.phones 0
   { "areacode":831, "number":5294368, "kind":"mobile" }
remove p.info.address.phones[$element.kind = "office"]
put p.info.children.Ron { "friends" : ["Julie"] },
add p.info.children.values().friends seq_concat(“Ada”, “Aris”)
where id = 0
returning *
```

After the update, the row looks like this:

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

The first two SET clauses change the profession and city of John Doe. The third SET reduces his income to one tenth. The fourth SET increases the age of his children by 1. Notice the use of the $

variable here: the expression p.info.children.values().age returns 3 ages; the SET will iterate over these ages, bind the $ variable to each age in turn, compute the expression $ + 1 for each age, and update the age with the new value. Notice that the income update could (and can) also have used a $ variable: set p.info.income = $ / 10. This would have saved the re-evaluation of the p.info.income path on the right hand side or the "=".

The ADD clause adds a new phone at position 0 inside the phones array. The REMOVE removes all the office phones (only one in this example). The PUT clause adds a friend for Ron. In this clause, the expression p.info.children.Ron returns the value associated with the Ron child. This value is a map (the json object { "age" : 3 }) and becomes the target of the update. The 2nd expression in the PUT ({ "friends" : ["Julie"] }) constructs and returns a new map. The fields of this map are added to the target map. Finally, the last ADD clause adds the same two new friends to each child (the seq_concat function is defined in section 7.1).

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

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

5.6.2 Example 2

This is an example of handling heterogeneity in json documents. Assume that the Peoples table contains the row from example 5.6.1, as well as the following row:

```
{
 "id":1,
 "info":
 {
  "firstName":"Jane",
```

```
  "lastName":"Doe",
  "address":
  {
   "city": "Santa Cruz",
   "state" : "CA",
   "phones" : { "areacode":831, "number":5294368, "kind":"mobile" }
  }
 }
}
```

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

```
declare $id integer;
        $areacode integer;
        $number integer;
        $kind string;
update People p
add p.info.address[$element.phones is of type (array(any))].phones
   { "areacode" : $areacode, "number" : $number, "kind" : $kind }
set p.info.address[$element.phones is of type (map(any))].phones =
   [ $, { "areacode" : $areacode, "number" : $number, "kind" : $kind } ]
where id = $id
```

In the ADD clause, the expression p.info.address[$element.phones is of type (array(any))].phones checks whether the phones field of the address is an array and if so, returns that array, which becomes the target of the ADD[4]. The 2nd expression in the add is a json-object constructor, that creates the new phone object and appends it into the target array.

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

5.6.3 Example 3

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

[4] Notice that although p.info.address is not an array, an array filtering step is applied to it in the target expression. This works fine because (as explained in section 4.4.5) the address will be treated as an array containing the address as its only element; as a result, $element will be bound to the address item, and if the filtering condition is true, the address items becomes the input to the next step in the path expression.

```
update People $p
set TTL remaining_hours($p) + 3 hours
where id = 5
```

The above statement extends the life of a row by 3 hours. Specifically, the remaining_hours function (see section 7.4) returns the number of full hours remaining until the expiration time of the row. In this example, this number is 8. So, the new TTL value is 8+3 = 11, and the expiration time of the row will be set to 2017-06-02:T08:00:00.0.

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

6 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 provide fast access to the rows of a table, when the information we are searching for is contained in the index. In section 6.1 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. In section 6.1 we will discuss indexes on strongly typed data only; indexes on JSON data will be discussed in section 6.2. In section 6.3 we describe the DROP INDEX statement. Finally, in section 6.4 we will see how indexes are used to optimize queries.

6.1 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 path_type? |
   keys_expr |
   values_expr path_type? |
   brackets_expr path_type?) ;

name_path : field_name (DOT field_name)* ;

field_name : id | DSTRING ;
```

```
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)? ;

path_type : AS (INTEGER | LONG | DOUBLE | STRING | BOOLEAN | NUMBER);
```

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, string, boolean, timestamp, or enum. Index fields may also store one of the ***special values***: NULL, json null, or EMPTY (EMPTY will be defined later in this section, and json null is possible only for indexes on JSON data). 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). The 3 special values are considered to be greater than any other values (i.e., they sort last in indexes). Among these 3 values, their relative order is EMPTY < json null < NULL.

Indexes can be characterized as ***simple indexes*** or ***multi-key indexes*** (which index arrays or maps). In both cases, for each table row, the index path expressions are evaluated and one or more index entries are created based on the values returned by the index path expressions. This is explained further in sections 6.1.2 and 6.1.3 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 allowed for index paths, because by default, the input is a table row that is being indexed.

- A single name_path is a list of one or more dot-separated field names. These field names may be ***schema fields***, i.e., field names that appear in the CREATE TABLE statement used to create the associated table. Using the example table from section 3.1.1, age and lastName are valid index paths for indexes over the Users table. However, name_paths may also contain field names of maps. For example, expenses.books is a valid index path; in this path books is not a schema field, but instead the name of an expense category that each user may or may not have.

- When a name_path is evaluated on a table row (or more generally, on a single input item), it must return at most one item. Furthermore, no arrays should be crossed during the evaluation. We will refer to these 2 restrictions as the ***name-path-restriction***. For example, otherNames.last cannot be used as an index path (because otherNames is an array field), even though it is a valid query path expression. However, the equivalent otherNames[].last path (which is not a single name_path) can be used instead.

- In the context of index creation, if a name_path returns an empty result, the special value **EMPTY** is used as the result instead. For example, the path expenses.books will return an empty result for a user that does not have a books expense category, but this empty result gets converted to EMPTY. So, in effect, each name_path returns exactly one item.

- A name_path may contain fields whose type is JSON. In this case, the index path is a ***json index path***, and the whole index is said to be a ***json index***. In the remainder of this section we will discuss non json indexes only. Json indexes will be discussed in section 6.2. Notice that the path_type production in the syntax of the CREATE INDEX statement is required for json index paths only; it must not be used for index paths on typed data.

- As shown by the syntax of the CREATE INDEX statement, filtering is not allowed, that is, .keys(), .values(), and [] steps cannot use a filtering condition. Furthermore, there can be at most one of such steps in each index path.

- .keys() steps can be applied to maps only (not records).

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

```
CREATE TABLE Users2 (
 id INTEGER,
 income INTEGER,
 address RECORD(street : STIRNG,
              city : STRING,
              state : STRING,
              phones : ARRAY(RECORD(area : INTEGER,
                                    number : INTEGER,
                                    kind : STRING))
             ),
```

```
  connections ARRAY(INTEGER),
  expenses MAP(INTEGER),
  PRIMARY KEY (id),
)
```

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

```
{
  "id":0,
  “income” : 1000
  "address":
  {
    “street” : “somewhere”
    "city": "Boston",
    "state"  : "MA",
    "phones" : [ { "area":408,  "number":50, "kind":"work" },
                 { "area":415,  "number":60, "kind":"work" },
                 { "area":NULL, "number":52, "kind":"home" },
               ]
  },
  "expenses" : { "housing" : 1000, "clothes" : 230, "books" : 20 },
  “connections” : [ 100, 20, 20, 10, 20]
}

{
  "id":1,
  “income” : NULL
  "address":
  {
    “street” : “everywhere”
    "city": "San Fransisco",
    "state"  : "CA",
    "phones" : [ { "area":408,  "number":50, "kind":"work" },
                 { "area":408,  "number":60, "kind":"home" },
               ]
  },
  "expenses" : { "housing" : 1000, "travel" : 300, },
  “connections” : [ ]
}

{
```

```
  "id":2,
  "income" : 2000
  "address":
  {
    "street" : "nowhere"
    "city": "San Jose",
    "state"  : "CA",
    "phones" : [ ]
  },
  "expenses" : NULL,
  "connections" : NULL
}
```

6.1.1 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.

6.1.2 Simple index examples

- create index idx1 on Users2 (income)

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

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

- create index idx2 on Users2 (address.state, address.city, income)

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

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

- create index idx3 on Users2 (expenses.books)

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

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

- create index idx4 on users2 (expenses.housing, expenses.travel)

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

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

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

An index is a multi-key index if:

1. There is at least one index path that uses a ***multi-key step*** (.keys(), .values(), or []). Any such index path will be called a ***multi-key index path***, and the associated index field a ***multi-key field***. The index definition may contain more than one multi-key paths, but all multi-key paths must use the same name_path before their multi-key step. Let M be this common name_path. For example, we can index both the area codes and the phone kinds of users, that is, the paths

address.phones[].area and address.phones[].kind can both appear in the same CREATE INDEX statement, in which case M is the path address.phones. On the other hand, we cannot create an index on Users2 using both of these paths: connections[] and address.phones[].area, because this is indexing two different arrays in the same index, which is not allowed.

2. Any non-multi-key index paths must be simple paths, as defined in section 6.1.1.

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 result of this path expression is a sequence of arrays, not atomic items. Neither is complex2.matrix[][].a valid, because we cannot index arrays inside other arrays (in fact this path will raise a syntax error, because the syntax allows at most one [] per index path). On the other hand, the path complex1.mapField.someKey[].someOtherKey is valid. In this case, M is the path complex1.mapField.someKey, which specifies an array containing maps. Notice that in this index path, someKey and someOtherKey are map-entry keys. So, although we are indexing arrays that are contained inside maps, and the arrays being indexed contain maps, the path is valid, because it is selecting specific entries from the maps involved rather than indexing all the map entries in addition to all the array entries.

4. If M specifies an array (i.e., the index is indexing an array-valued field):

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: the simple index paths (if any) are computed on R. Then, M[] is computed (as if it were a query path expression), returning either NULL, or EMPTY, or all the elements of the array returned by M. Finally, for each value V returned by M[], an index entry is created whose field values are V and the values of the simple paths.

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

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.

6.1.4 Multi-key index examples

In this section we give some examples of multi-key indexes.

- create index midx1 on Users2 (connections[])

Creates an index on the elements of the connections array. The contents of this index for the sample rows in Users2 are:

```
[ 10,    0 ]
[ 20,    0 ]
[ 100,   0 ]
[ EMPTY, 1 ]
[ NULL,  2 ]
```

- create index midx2 on Users2 (address.phones[].area, income)

Creates an index on the area codes and income of users. The contents of this index for the sample rows in Users2 are:

```
[ 408,   1000, 0 ]
```

```
[ 408,   NULL, 1 ]
[ 415,   1000, 0 ]
[ EMPTY, 2000, 2 ]
[ NULL,  1000, 0 ]
```

- create index midx3 on Users2
 (address.phones[].area, address.phones[].kind, income)

  Creates an index on the area codes, the phone number kinds, and the income of users. The contents of this index for the sample rows in Users2 are:

```
[ 408,   “work”, 1000, 0 ]
[ 408,   “home”, NULL, 1 ]
[ 408,   “work”, NULL, 1 ]
[ 415,   “work”, 1000, 0 ]
[ EMPTY, EMPTY,  2000, 2 ]
[ NULL,  “home”, 1000, 0 ]
```

- create index midx4 on Users2 (expenses.keys(), expenses.values())

  Creates an index on the fields (both keys and values) of the expenses map. The contents of this index for the sample rows in Users2 are:

```
[ “books”,     50, 0 ]
[ “clothes”,  230, 0 ]
[ “housing”, 1000, 0 ]
[ “housing”, 1000, 1 ]
[ “travel”,   300, 1 ]
[ NULL,      NULL, 2 ]
```

6.2 Indexing JSON data

In the current implementation SQL for Oracle NoSQL supports ***typed json indexes***. As their name implies, such indexes place some type-related restrictions on the json data that is being indexed. Creation of a typed json index will fail if the associated table contains any rows with data that violate the restrictions imposed by the index. Similarly, an insert/update operation will be rejected if the new row does not conform to the restrictions imposed by one or more existing json indexes. However, as long as the type constraints are satisfied, typed json indexes are very similar to indexes on strongly typed data, both in the way their contents are evaluated as well as in the ways they are used by queries.

In general, an index is a json index if it indexes at least one field that is contained inside json data. Since json data is schemaless, the type of the indexed field may be different across table rows. Currently, such type variation is not allowed. Instead, an index path that specifies such a field must be

followed by a type declaration (using the AS keyword) in the CREATE INDEX statement. The type must be one of the json atomic types (numeric types, string, or boolean). Semantically, such a declaration means that the evaluation of the index path on a row must always result in a sequence of items, where each item in the sequence is NULL, or json null, of has a type that is the same or a subtype of the declared type. If the sequence is empty, the special value EMPTY is put in the index.

Like the non-json indexes, typed json indexes may be simple or multi-key. These are described further in sections 6.2.1 and 6.2.2, respectively. The examples shown there are based on a users table that stores all user info as json data. Specifically, we will use the table created like this:

```
CREATE TABLE Users3 (id integer, info json, primary key(id))
```

We will also assume that Users3 is populated with the following sample rows (shown in JSON format). Contrary to table Users2, the sub-documents under “info” remain as json data when these documents are inserted in Users3. This means that the null values inside “info” sub-documents will remain as json null values in the table. Notice however that for the row with id 3, the value of the (top-level) “info” column will be the SQL NULL. Notice also the variations in the data across different rows. For example, info.address.phones is usually an array, but in row with id 4 it is a single json object.

```
{
  "id":0,
  “info”: {
    “income” : 1000
    "address": {
      “street” : “somewhere”
      "city": "Boston",
      "state"  : "MA",
      "phones" : [ { "area":408,  "number":50, "kind":"work" },
                   { "area":415,  "number":60, "kind":"work" },
                   { "area":null, "number":52, "kind":"home" },
                 ]
    },
    "expenses" : { "housing" : 1000, "clothes" : 230, "books" : 20 },
    “connections” : [ 100, 20, 20, 10, 20]
  }
}

{
  "id":1,
  “info”: {
    “income” : null
    "address":{
      “street” : “everywhere”
      "city": "San Fransisco",
      "state"  : "CA",
      "phones" : [ { "area":408,  "number":50, "kind":"work" },
```

```
                { "area":408,  "number":60, "kind":"home" },
                “408-345-1232”
              ]
      },
      "expenses" : { "housing" : 1000, "travel" : 300, },
      “connections” : [ ]
   }
}

{
   "id":2,
   “info” : {
      “income” : 2000
      "address": {
         “street” : “nowhere”
         "city": "San Jose",
         "state"  : "CA",
         "phones" : [ ]
      },
      "expenses" : null,
      “connections” : null
   }
}

{
   “id”:3,
}


{
   "id":4,
   “info”: {
      "address": {
         “street” : “top of the hill”
         "city": "San Fransisco",
         "state"  : "CA",
         "phones" : { "area":408,  "number":50, "kind":"work" },
      },
      "expenses" : { "housing" : 1000, "travel" : 300, },
      “connections” : [ 30, 5, null ]
   }
}
```

```
{
  "id":5,
  "info": {
    "address": {
      "street" : "end of the road"
      "city": "Portland",
      "state"  : "OR"
    }
  }
}
```

6.2.1 Simple typed json indexes

A simple typed json index path has the form: name_path path_type. As with non-json index paths, when name_path is evaluated on any table row, it must return at most one atomic item and must not cross any arrays. For non-json indexes, the type of result item is always the same and can be deduced by the table schema. This is not possible for schemaless json data, and as a result, a path_type must be used to declare and enforce a type for the indexed field. The implication of such a declaration is that when name_path is evaluated on any table row, it must return an empty result, or NULL, or json null, or an instance of the declared type. As mentioned already, in case of an empty result, the special EMPTY value is used as the result.

If a CREATE INDEX statement consists of at least one simple typed json path and all the other paths are also simple (json or non-json), then the index is a simple typed json index. The contents of such an index are determined the same way as for non-json simple indexes.

The following examples are the json versions of the ones in 6.1.2.

- create index jidx1 on Users3(info.income as integer)

  It creates an index with one entry per user in the Users table. The entry contains the income and id (the primary key) of the user represented by the row. The contents of this index for the sample rows in Users3 are:

  ```
  [ 1000,  0 ]
  [ 2000,  2 ]
  [ EMPTY, 4 ]
  [ EMPTY, 5 ]
  [ JNULL, 1 ]
  [ NULL,  3 ]
  ```

- create index jidx2 on Users3 (info.address.state as string,

```
                 info.address.city as string,
                 info.income as integer)
```

It creates an index with one entry per user in the Users table. The entry contains the state, city, income and id (the primary key) of the user represented by the row. The contents of this index for the sample rows in Users3 are:

```
[ "CA", "San Fransisco", EMPTY, 4 ]
[ "CA", "San Fransisco", JNULL, 1 ]
[ "CA", "San Jose",      2000,  2 ]
[ "MA", "Boston",        1000,  0 ]
[ "OR", "Portland",      EMPTY, 5 ]
[ NULL,  NULL,           NULL,  3 ]
```

- create index jidx3 on Users3 (info.expenses.books as integer)

Creates an index entry for each user. The entry contains the user's spending on books, if the user does record spending on books, or EMPTY if there is no "books" entry in expenses or there is no expenses map at all, or NULL if there is no info at all (i.e. the value of the info column is NULL). The contents of this index for the sample rows in Users3 are:

```
[ 20,    0 ]
[ EMPTY, 1 ]
[ EMPTY, 2 ]
[ EMPTY, 4 ]
[ EMPTY, 5 ]
[ NULL,  3 ]
```

- create index jidx4 on users3 (info.expenses.housing as integer,
                 info.expenses.travel as integer)

Creates an index entry for each user. The entry contains 2 fields: (a) the user's housing expenses, or EMPTY if the user does not record housing expenses or there is no expenses field at all, and (b) the user's travel expenses, or EMPTY if the user does not record travel expenses or there is no expenses field at all. If info is NULL, both fields in the index entry will be NULL. The contents of this index for the sample rows in Users3 are:

```
[ 1000, 300,    1 ]
[ 1000, 300,    4 ]
[ 1000, EMPTY,  0 ]
[ EMPTY, EMPTY, 2 ]
[ EMPTY, EMPTY, 5 ]
[ NULL, NULL,   3 ]
```

6.2.2 Multi-key typed json indexes

An index is a multi-key typed json index if its definition includes at least one json index path that uses a multi-key step (keys(), values(), or []). Furthermore, if the index path uses values() or [], its declaration in the CREATE INDEX statement must be followed by a path_type. As mentioned already, when evaluated on any table row, such an index path must return zero or more items, and each item returned must be either NULL, or JNULL, or an instance of the declared type.

Multi-key typed json indexes are very similar to non-json multi-key indexes. In fact, the description in section 6.1.3 applies almost identically to json indexes as well. The only difference is that an actual row may not have an array or a map at the expected place. For example:

- Let a.b.c[].d be a json index path. The fact that [] appears after c implies that the user expects c to be an array (containing json objects having a d field). Furthermore, none of a, b, or d may be arrays. Oracle NoSQL enforces the later restriction, but it allows c to be any kind of item. If it's not an array, it will be treated as if it were an array containing that single item. In other words, for each table row, the path a.b.c[].d will be evaluated as if it were a DML path expression in a query. If c is an atomic, the result will be EMPTY; if c is a map, the value of the d field in that map (if any) will be indexed.

- Let a.b.c.values().d be a json index path. The fact that values() appears after c implies that the user expects c to be a json object (containing json objects having a d field). Furthermore, none of a, b, or d may be arrays. In this case, c may not be an array either (because then all the objects in the array would be indexed, as well as all the values in each such object). However, c may be an atomic, in which case the result will be EMPTY.

The following examples are the json versions of the ones in 6.1.4

- create index jmidx1 on Users3 (info.connections[] as integer)

  Creates an index on the elements of the connections array. The contents of this index for the sample rows in Users3 are:

  ```
  [  5,    4 ]
  [ 10,    0 ]
  [ 20,    0 ]
  [ 30,    4 ]
  [ 100,   0 ]
  [ EMPTY, 1 ]
  [ EMPTY, 5 ]
  [ JNULL, 2 ]
  [ JNULL, 4 ]
  [ NULL,  3 ]
  ```

- create index jmidx2 on Users3 (

info.address.phones[].area as integer,
info.income as integer)

Creates an index on the area codes and income of users. The contents of this index for the sample rows in Users3 are:

```
[ 408,   1000,  0 ]
[ 408,   EMPTY, 4 ]
[ 408,   JNULL, 1 ]
[ 415,   1000,  0 ]
[ EMPTY, 2000,  2 ]
[ EMPTY, EMPTY, 5 ]
[ EMPTY, JNULL, 1 ]
[ JNULL, 1000,  0 ]
[ NULL,  NULL,  3 ]
```

- create index jmidx3 on Users3 (
  info.address.phones[].area as integer,
  info.address.phones[].kind as string,
  info.income)

  Creates an index on the area codes, the phone number kinds, and the income of users. The contents of this index for the sample rows in Users3 are:

```
[ 408,   “home”, JNULL, 1 ]
[ 408,   “work”, 1000,  0 ]
[ 408,   “work”, EMPTY, 4 ]
[ 408,   “work”, JNULL, 1 ]
[ 415,   “work”, 1000,  0 ]
[ EMPTY, EMPTY,  2000,  2 ]
[ EMPTY, EMPTY,  EMPTY, 5 ]
[ EMPTY, EMPTY,  JNULL, 1 ]
[ JNULL, “home”, 1000,  0 ]
[ NULL,  NULL,    NULL, 3 ]
```

- create index jmidx4 on Users3 (
  info.expenses.keys(),
  info.expenses.values() as integer)

  Creates an index on the fields (both keys and values) of the expenses map. Notice that the keys() portion of the index definition must not declare a type. This is because the type will always be String. The contents of this index for the sample rows in Users2 are:

```
[ “books”,     50, 0 ]
```

```
[ “clothes”,  230, 0 ]
[ “housing”, 1000, 0 ]
[ “housing”, 1000, 1 ]
[ “housing”, 1000, 4 ]
[ “travel”,   300, 1 ]
[ “housing”, 1000, 4 ]
[ EMPTY,    EMPTY, 2 ]
[ EMPTY,    EMPTY, 5 ]
[ NULL,      NULL, 3 ]
```

6.3 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.

6.4 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.

6.4.1 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 on the entries being scanned. In the current implementation, the following kinds of predicates are considered as candidate start/stop predicates: (a) comparisons, using either the value or sequence (any) comparison operators, but not != or !=any, (b) IS NULL and IS NOT NULL operators and (c) EXISTS predicates.

If an index is used in a query, its index predicates are removed from the query because they are evaluated by the index scan. We say that index predicates are "pushed to the index". In the rest of this section we explain applicable indexes further via a number of example queries, and using the non-json indexes from sections 6.1.2 and 6.1.4. The algorithm for finding applicable json indexes is essentially the same as for non-json indexes.

Example 1

```
select *
from Users2
where 10 < income and income < 20
```

The query contains 2 index predicates. Indexes idx1, idx2, midx2, and midx3 are all applicable. For index idx1, 10 < income is a start predicate and income < 20 is a stop predicate. For the other indexes, both predicates are filtering predicates. If, say, idx2 were to be used, the subrange to scan is the whole index. Obviously, idx1 is better than the other indexes in this case. Notice however, that the

number of table rows retrieved would be the same whether idx1 or idx2 were used. If midx2 or midx3 were used, the number of **distinct** rows retrieved would be the same as for idx1 and idx2, but a row would be retrieved as many times as the number of elements in the phones array of that row. Such duplicates are eliminated from the final query result set.

Example 2

```
select *
from Users2
where 20 < income or income < 10
```

The query contains 1 index predicate, which is the whole WHERE expression. Indexes idx1, idx2, midx2, midx3 are all applicable. For all of them, the predicate is a filtering predicate.

Example 3

```
select *
from Users2
where 20 < income or age > 70
```

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

Example 4

```
select *
from Users2 u
where u.address.state = “CA” and u.address.city = “San Jose”
```

Only idx2 is applicable. There are 2 index predicates, both of which serve as both start and stop predicates.

Example 5

```
select id, 2*income
from Users2 u
where u.address.state = “CA” and u.address.city = “San Jose”
```

Only idx2 is applicable. There are 2 index predicates, both of which serve as both start and stop predicates. In this case, the id and income information needed in the SELECT clause is available in the index. As a result, the whole query can be answered from the index only, with no access to the table. We say that index idx2 is a **covering index** for the query in Example 5. The query processor will apply this optimization.

Example 6

```
select *
from Users2 u
where u.address.state = “CA” and
    u.address.city = “San Jose” and
    u.income > 10
```

idx1, idx2, midx2, and midx3 are applicable. For idx2, there are 3 index predicates: the state and city predicates serve as both start and stop predicates; the income predicate is a start predicate. For idx1 only the income predicate is applicable, as a start predicate. For midx2 and midx3, the income predicate is a filtering one.

Example 7

```
select *
from Users2 u
where u.address.state = “CA” and
    u.income > 10
```

idx1, idx2, midx2, and midx3 are applicable. For idx2, there are 2 index predicates: the state predicate serves as both start and stop predicate; the income predicate is a filtering predicate. The income predicate is a start predicate for idx1 and a filtering predicate for midx2 and midx3.

Example 8

```
declare
$city string;
select *
from Users2 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).

Example 9

```
select id
```

from Users3 u
where exists u.info.income

In this example we use table Users3, which stores all information about users as json data. The query looks for users who record their income. Index jidx1 is applicable. The EXISTS condition is actually converted to 2 index start/stop conditions: u.info.income < EMPTY and u.info.income > EMPTY. As a result, two range scans are performed on the index.

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 (EXISTS, IS NULL and IS NOT NULL are converted to predicates of this form, as shown in Q9).
- <const expr> is an expression built from literals and external variables only (does not reference any tables or internal variables)
- <path expr> is a path expression that is "matches" an index path P appearing in the CREATE INDEX statement for IDX. So far we have seen examples of exact matches only. In the examples below we will see some non-exact matches as well.
- If P is not IDX's 1st index path, there are equality start/stop predicates for each index path appearing before P in IDX's definition.
- The comparison operator may be one of the "any" operators. Such operators are matched against the multi-key index paths of multi-key indexes. As shown in the examples below, additional restrictions apply for such predicates.

Example 10

```
select *
from users2 u
where u.connections[] =any 10
```

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

Example 11

```
select *
from users2 u
where u.connections[0:4] =any 10
```

midx1 is applicable. The predicate to push down to mdx1 is u.connections[] =any 10, in order to eliminate users who are not connected at all with user 10. However, the original predicate (u.connections[0:4] =any 10) must be retained in the query to eliminate users who do have a connection with user 10, but not among their 5 strongest connections. This is an example where the query path expression does not match exactly the corresponding index path.

Example 12

```
select *
from users2 u
where u.connections[] >any 10
```

midx1 is applicable and the predicate is a start predicate.

Example 13

```
select id
from users2 u
where 10 <any u.connections[] and u.connections[] <any 100
```

midx1 is applicable, but although each predicate by itself is an index predicate, only one of them can actually be used as such. To see why, first notice that the query asks for users that have a connection with id greater than 10 and **another** connection (which may or may not be the same as the 1st one) with id less than 100. Next, consider a Users2 table with only 2 users (say with ids 200 and 500) having the following connections arrays respectively: [1, 3, 110, 120] and [1, 50, 130]. Both of these arrays satisfy the predicates in the query, and both users should be returned as a result. Now, consider midx1; it contains the following 7 entries:

[1, 200], [1, 500], [3, 200], [50, 500], [110, 200], [120, 200], [130, 500]

By using only the 1st predicate as a start predicate to scan the index, and applying the 2nd predicate on the rows returned by the index scan, the result of the query is 500, 200, which is correct. If on the other hand both predicates were used for the index scan, only entry [50, 500] would qualify, and the query would return only user 500.

Example 14

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

```
select id
from users2 u
where exist u.connections[10 < $element and $element < 100]
```

Assuming the same 2 users as in Example 13, the result of this query is user 500 only and both predicates can be used as index predicates (start and stop), because both predicates apply to the same array element. The query processor will indeed push both predicates to mdx1.

Example 15

```
select *
from Users2 u
where u.address.phones.area =any 650 and
    u.address.phones.kind =any work and
    u.income > 10
```

This query looks for users whose income is greater than 10, and have a phone number with area code 650, and also have a work phone number (whose area code may not be 650). Index midx3 is applicable, but the address.phones.kind predicate cannot be used as an index predicate (for the same reason as in Example 13). Only the area code predicate can be used as a start/stop predicate and the income predicate as a filtering one. Indexes idx1, idx2, and midx2 are also applicable in Example 15.

Example 16

```
select *
from Users2 u
where u.expenses.housing = 10000
```

idx4 is applicable and the predicate is both a start and a stop predicate. midx4 is also applicable. To use midx4, two predicates must be pushed to it, even though only one appears in the query. The 1st predicate is on the "keys" index field and the second on the "values" field. Specifically, the predicates key = "price" and value = 10000 are pushed as start/stop predicates. This is another example where the match between the query path expression and an index path is not exact: we match expenses.housing with the expenses.values() index path, and additionally, generate an index predicate for the properties.keys() index path.

Example 17

```
select *
from Users2 u
where u.expenses.travel = 1000 and u.expenses.clothes > 500
```

midx4 is applicable. Each of the query predicates is by itself an index predicate and can be pushed to midx4 the same way as the expenses.housing predicate in the previous example. However, the query predicates cannot be both pushed (at least not in the current implementation). The query processor has to choose one of them to push and the other will remain in the query. Because the expenses.travel predicate is an equality one, it's more selective than the greater-than predicate and the query processor will use that.

6.4.2 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 shards 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.

7 Built-in functions

7.1 Functions on sequences

any* **seq_concat**(any*, ...)
seq_concat is a variadic function: it can have any number of arguments. It simply evaluates its arguments (if any) in the order they are listed in the argument list, and concatenates the sequences returned by these arguments.

7.2 Functions on complex values.

integer? **size**(any?)
Returns the size of a complex item (array, map, record). Although the parameter type appears as ANY?, the function will actually raise an error if the given item is not complex. The function accepts an empty sequence as argument, in which case it will return the empty sequence. The function will return NULL if its input is NULL.

7.3 Functions on timestamps

integer? **year**(timestamp?)
Returns the year for the given timestamp. The returned value is in the range -6383 to 9999. If the argument is NULL or empty, the result is also NULL or empty.

integer? **month**(timestamp?)
Returns the month for the given timestamp, in the range 1 ~ 12. If the argument is NULL or empty, the result is also NULL or empty.

integer? **day**(timestamp?)
Returns the day of month for the timestamp, in the range 1 ~ 31. If the argument is NULL or empty, the result is also NULL or empty.

integer? **hour**(timestamp?)
Returns the hour of day for the timestamp, in the range 0 ~ 23. If the argument is NULL or empty, the result is also NULL or empty.

integer? **minute**(timestamp?)
Returns the minute for the timestamp, in the range 0 ~ 59. If the argument is NULL or empty, the result is also NULL or empty.

integer? **second**(timestamp?)
Returns the second for the timestamp, in the range 0 ~ 59. If the argument is NULL or empty, the result is also NULL or empty.

integer? **millisecond**(timestamp?)
Returns the fractional second in millisecond for the timestamp, in the range 0 ~ 999. If the argument is NULL or empty, the result is also NULL or empty.

integer? **microsecond**(timestamp?)
Returns the fractional second in microsecond for the timestamp, in the range 0 ~ 999999. If the argument is NULL or empty, the result is also NULL or empty.

integer? **nanosecond**(timestamp?)
Returns the fractional second in nanosecond for the timestamp, in the range 0 ~ 999999999. If the argument is NULL or empty, the result is also NULL or empty.

integer? **week**(timestamp?)
Returns the week number within the year where a week starts on Sunday and the first week has a minimum of 1 day in this year, in the range 1 ~ 54. If the argument is NULL or empty, the result is also NULL or empty.

integer? **isoweek**(timestamp?)
Returns the week number within the year based on IS0-8601, where a week starts on Monday and the first week has a minimum of 4 days in this year, in range 0 ~ 53. If the argument is NULL or empty, the result is also NULL or empty.

7.4 Functions on rows

As described in section 2.7, table rows are record values conforming to the table schema. However, table rows have some additional properties that are not part of the tables schema (i.e., they are not stored as top-level columns or nested fields). To extract the value of such properties, the functions listed in this sub-section must be used.

Although the signature of these functions specifies AnyRecord as the type of the input parameter, the functions actually require a row as input. The only expression that returns a row, is a row variable, that is a table alias whose name starts with '$'. Section 5.6.3 shows an example of using the remaining_hours() function, which is one of the row available functions.

integer **remaining_hours**(AnyRecord)
Returns the number of full hours remaining until the row expires. If the row has no expiration time, it returns a negative number.

integer **remaining_days**(AnyRecord)
Returns the number of full days remaining until the row expires. If the row has no expiration time, it returns a negative number.

timestap(0) **expiration_time**(AnyRecord)
Returns the expiration time of the row, as a timestamp value of precision zero. If the row has no expiration time, it returns zero.

long **expiration_time_millis**(AnyRecord)
Returns the expiration time of the row, as the number of milliseconds since January 1, 1970 UTC. If the
row has no expiration time, it returns zero.

7.5 Miscellaneous Functions

long **current_time_millis**()
Returns the current time in UTC, as the number of milliseconds since January 1, 1970 UTC.

timestamp(3) **current_time**()
Returns the current time in UTC, as a timestamp value with millisecond precision.

8 Appendix : The full SQL grammar

```
program :
    (
    query
  | update_statement
  | 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 ;

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 select_list ;

select_list :
  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 ;

or_expr : and_expr | or_expr OR and_expr ;

and_expr : not_expr | and_expr AND not_expr ;
```

```
not_expr : NOT? is_null_expr ;

is_null_expr : cond_expr (IS NOT? NULL)? ;

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 : DOT (map_filter_step | map_field_step) ;

map_field_step :
    id | string | var_ref | parenthesized_expr | func_call ;

map_filter_step : (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 :
  parenthesized_expr |
  const_expr |
  column_ref |
  var_ref |
  array_constructor |
  map_constructor |
  case_expr |
  cast_expr |
  extract_expr |
  func_call ;

parenthesized_expr : LPAREN expr RPAREN;

const_expr : number | 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 ;

extract _expr : EXTRACT LP id FROM expr RP ;

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

update_statement :
  prolog?
  UPDATE table_name AS? tab_alias?
        update_clause (COMMA update_clause)*
  WHERE expr
```

```
  returning_clause? ;

returning_clause : RETURNING select_list ;

update_clause :
  (SET set_clause (COMMA (update_clause | set_clause))*) |
  (ADD add_clause (COMMA (update_clause | add_clause))*) |
  (PUT put_clause (COMMA (update_clause | put_clause))*) |
  (REMOVE remove_clause (COMMA remove_clause)*)
  (SET TTL ttl_clause (COMMA update_clause)*) ;

set_clause : target_expr EQ expr ;

add_clause : target_expr pos_expr? expr ;

put_clause : target_expr expr ;

remove_clause : target_expr ;

ttl_clause : (add_expr (HOURS | DAYS)) | (USING TABLE DEFAULT) ;

target_expr : path_expr ;

pos_expr : add_expr ;

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 path_type? |
     keys_expr |
     values_expr path_type? |
     brackets_expr path_type?) ;

name_path : field_name (DOT field_name)* ;

field_name : id | DSTRING ;

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)? ;

path_type : AS (INTEGER | LONG | DOUBLE | STRING | BOOLEAN | NUMBER);

type_def :
  INTEGER |
  LONG |
  FLOAT |
  DOUBLE |
  NUMBER |
  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 | NUMBER_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 | DAYS |
    DECLARE | DEFAULT | DESC | DOUBLE | DROP |
    ELSE | END | ENUM | EXISTS | EXTRACT | FIRST | FLOAT | FROM |
    HOURS | 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 | TTL | TYPE | USING | 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+ ) ;

NUMBER_CONST : (INT_CONST | FLOAT_CONST) [Nn]

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] ;
```