Oracle® Database
Developers Guide

Release 22.2
F57948-03
October 2022

Oracle Database Developers Guide, Release 22.2

F57948-03

Primary Author: Vandana Rajamani

Contents

4 Develop

5 Reference

Index

List of Tables

1

Get Started

The articles in this section focus on providing the quickest path to using SQL for Oracle NoSQL Database . It contains the schema used in the examples and sample data to run queries.

Getting started with SQL for Oracle NoSQL Database

Welcome to SQL for Oracle NoSQL Database. This language provides a SQL-like interface to Oracle NoSQL Database. The SQL for Oracle NoSQL Database data model supports flat relational data, hierarchical typed (schema-full) data, and schema-less JSON data. SQL for Oracle NoSQL Database is designed to handle all such data seamlessly without any impedance mismatch among the different sub-models. Impedance mismatch is the problem that occurs due to differences between the database model and the programming language model.

Pre-requisites: You already have an installation of the Oracle NoSQL Database. You could also use KVLite which is a simplified version of the Oracle NoSQL Database.

- Schemas used in the examples
- Starting the SQL shell
- Tables used in the examples
- Describe tables
- Sample data to run queries
- Table Hierarchies

Schemas used in the examples

You have two different schemas (with real-time scenarios) for learning various SQL concepts. These two schemas will include various data types that can be used in the Oracle NoSQL database.

Schema 1: BaggageInfo schema

Using this schema you can handle a use case wherein passengers traveling on a flight can track the progress of their checked-in bags or luggage along the route to the final destination. This functionality can be made available as part of the airline's mobile application. Once the passenger logs into the mobile application, the ticket number or reservation code of the current flight is displayed on the screen. Passengers can use this information to search for their baggage information. The mobile application is using NoSQL Database to store all the data related to the baggage. In the backend, the mobile application logic performs SQL queries to retrieve the required data.

Schema 2: Streaming Media Service - Persistent User Profile Store

Consider a TV streaming application. It streams various shows that are watched by customers across the globe. Every show has a number of seasons and every season has

multiple episodes. You need a persistent meta-data store that keeps track of the current activity of the customers using the TV streaming application. Using this schema you can provide useful information to the customer such as episodes they watched, the watch time per episode, the total number of seasons of the show they watched, etc. The data is stored in the NoSQL Database and the application performs SQL queries to retrieve the required data and make it available to the user.

Starting the SQL shell

You can run SQL queries and run DDL statements directly from the SQL shell. Here is the general usage to start the shell:

```
java -jar KVHOME/lib/sql.jar
       -helper-hosts <host:port[,host:port]*>
       -store <storeName>
       [-username <user>]
       [-security <security-file-path>]
       [-timeout <timeout ms>]
       [-consistency <NONE_REQUIRED(default) | ABSOLUTE |
NONE_REQUIRED_NO_MASTER>]
       [-durability <COMMIT_SYNC(default) | COMMIT_NO_SYNC |
COMMIT_WRITE_NO_SYNC>]
       [single command and arguments]
```

where:

`-consistency` Configures the read consistency used for this session.

`-durability` Configures the write durability used for this session.

`-helper-hosts` Specifies a comma-separated list of hosts and ports.

`-store` Specifies the name of the store.

`-timeout` Configures the request timeout used for this session.

`-username` Specifies the username to login as.

For example, you can start the shell like this:

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

This command assumes that a store `kvstore` is running at port `5000`. After the SQL starts successfully, you run queries.

Tables used in the examples

The table is the basic structure to hold user data.

Schema 1: BaggageInfo schema

The table used in this schema is `BaggageInfo`. This schema has a combination of fixed data types like LONG, STRING. It also has a schema-less JSON (`bagInfo`) as one of its columns. The schema-less JSON does not have a fixed data type. The bag

information of the passengers is a schema-less JSON. In contrast, the passenger's information like ticket number, full name, gender, contact details is all part of a fixed schema. You can add any number of fields to this non-fixed schemaless JSON field. .

The following code creates the table.

```
CREATE TABLE BaggageInfo (
ticketNo LONG,
fullName STRING,
gender STRING,
contactPhone STRING,
confNo STRING,
bagInfo JSON,
PRIMARY KEY (ticketNo)
)
```

Schema 2: Streaming Media Service - Persistent User Profile Store

The table used in this schema is `stream_acct`. This schema has a composite primary key column comprised of `acct_id` and `user_id`. The schema also includes a JSON column (`acct_data`), which is schema-less. The schema-less JSON does not have a fixed data type. You can add any number of fields to this non-fixed schemaless JSON field.

The following code creates the table.

```
CREATE TABLE stream_acct(
acct_id INTEGER,
user_id STRING,
acct_data JSON,
PRIMARY KEY(acct_id, user_id)
)
```

Describe tables

You use DESCRIBE or DESC command to view the description of a table.

```
(DESCRIBE | DESC) [AS JSON] TABLE table_name [ "(" field_name")"]
```

AS JSON can be specified if you want the output to be in JSON format. You could get information about a specific field in any table by providing the field name.

Example 1: Describe a table

```
DESCRIBE TABLE stream_acct
```

Output:

```
=== Information ===
 +-------------+-----+-------+----------+--------+----------+---------
+---------+-------------+
 |    name     | ttl | owner | sysTable | parent | children | regions |
indexes | description |
 +-------------+-----+-------+----------+--------+----------+---------
```

```
+---------+-------------+
 | stream_acct |     |       | N        |        |          |
|         |             |
 +-------------+-----+-------+----------+--------+----------+---------
+---------+-------------+
 === Fields ===
 +----+-----------+---------+----------+---------+----------
+------------+----------+
 | id |   name    |  type   | nullable | default | shardKey |
primaryKey | identity |
 +----+-----------+---------+----------+---------+----------
+------------+----------+
 |  1 | acct_id   | Integer | N        | NULL    | Y        |
Y          |          |
 +----+-----------+---------+----------+---------+----------
+------------+----------+
 |  2 | user_id   | String  | N        | NULL    | Y        |
Y          |          |
 +----+-----------+---------+----------+---------+----------
+------------+----------+
 |  3 | acct_data | Json    | Y        | NULL    |
|            |          |
 +----+-----------+---------+----------+---------+----------
+------------+----------+
```

Example 2: Describe a table and display the output as JSON

```
DESC AS JSON TABLE BaggageInfo
```

Output:

```
{
  "json_version" : 1,
  "type" : "table",
  "name" : "BaggageInfo",
  "fields" : [{
    "name" : "ticketNo",
    "type" : "LONG",
    "nullable" : false
  }, {
    "name" : "fullName",
    "type" : "STRING",
    "nullable" : true
  }, {
    "name" : "gender",
    "type" : "STRING",
    "nullable" : true
  }, {
    "name" : "contactPhone",
    "type" : "STRING",
    "nullable" : true
  }, {
    "name" : "confNo",
    "type" : "STRING",
```

```
    "nullable" : true
  }, {
    "name" : "bagInfo",
    "type" : "JSON",
    "nullable" : true
  }],
  "primaryKey" : ["ticketNo"],
  "shardKey" : ["ticketNo"]
}
```

Example 3: Describe one particular field of a table

```
DESCRIBE TABLE BaggageInfo (ticketNo)
```

Output:

```
 +----+----------+------+----------+---------+----------+------------
+----------+
 | id |   name   | type | nullable | default | shardKey | primaryKey |
identity |
 +----+----------+------+----------+---------+----------+------------
+----------+
 |  1 | ticketNo | Long | N        | NULL    | Y        | Y
|          |
 +----+----------+------+----------+---------+----------+------------
+----------+
```

Sample data to run queries

Schema 1: BaggageInfo schema

If you want to follow along with the examples, download the script baggageschema_loaddata.sql and run it as shown below. This script creates the table used in the example and loads data into the table. One sample row is shown below.

```
"ticketNo" : 1762344493810,
"fullName" : "Adam Phillips",
"gender" : "M",
"contactPhone" : "893-324-1064",
"confNo" : "LE6J4Z",
 [ {
    "id" : "79039899165297",
    "tagNum" : "17657806255240",
    "routing" : "MIA/LAX/MEL",
    "lastActionCode" : "OFFLOAD",
    "lastActionDesc" : "OFFLOAD",
    "lastSeenStation" : "MEL",
    "flightLegs" : [ {
      "flightNo" : "BM604",
      "flightDate" : "2019-02-01T01:00:00",
      "fltRouteSrc" : "MIA",
      "fltRouteDest" : "LAX",
      "estimatedArrival" : "2019-02-01T03:00:00",
```

```
      "actions" : [ {
        "actionAt" : "MIA",
        "actionCode" : "ONLOAD to LAX",
        "actionTime" : "2019-02-01T01:13:00"
      }, {
        "actionAt" : "MIA",
        "actionCode" : "BagTag Scan at MIA",
        "actionTime" : "2019-02-01T00:47:00"
      }, {
        "actionAt" : "MIA",
        "actionCode" : "Checkin at MIA",
        "actionTime" : "2019-02-01T23:38:00"
      } ]
    }, {
      "flightNo" : "BM667",
      "flightDate" : "2019-01-31T22:13:00",
      "fltRouteSrc" : "LAX",
      "fltRouteDest" : "MEL",
      "estimatedArrival" : "2019-02-02T03:15:00",
      "actions" : [ {
        "actionAt" : "MEL",
        "actionCode" : "Offload to Carousel at MEL",
        "actionTime" : "2019-02-02T03:15:00"
      }, {
        "actionAt" : "LAX",
        "actionCode" : "ONLOAD to MEL",
        "actionTime" : "2019-02-01T07:35:00"
      }, {
        "actionAt" : "LAX",
        "actionCode" : "OFFLOAD from LAX",
        "actionTime" : "2019-02-01T07:18:00"
      } ]
    } ],
    "lastSeenTimeGmt" : "2019-02-02T03:13:00",
    "bagArrivalDate" : "2019.02.02T03:13:00"
  } ]
```

Start your KVSTORE or KVLite and open the SQL.shell.

```
java -jar lib/kvstore.jar kvlite -secure-config disable
java -jar lib/sql.jar -helper-hosts localhost:5000 -store kvstore
```

Using the `load` command, run the script.

```
load -file baggageschema_loaddata.sql
```

Schema 2: Streaming Media Service - Persistent User Profile Store

Download the script acctstream_loaddata.sql and run it as shown below. This script creates the table used in the example and loads data into the table. One sample row is shown below.

```
1,
"user01",
{
   "firstName" : "John",
   "lastName" : "Sanders",
   "country" : "USA",
   "contentStreamed": [
      {
         "showName": "Call My Agent",
         "showId": 12,
         "showType": "tvseries",
         "numSeasons" : 2,
         "seriesInfo": [
            {
               "seasonNum" : 1,
               "numEpisodes" : 2,
               "episodes": [
                  { "episodeID" : 20, "lengthMin" : 40, "minWatched" : 40 },
                  { "episodeID" : 30, "lengthMin" : 42, "minWatched" : 42 }
               ]
            },
            {
               "seasonNum": 2,
               "numEpisodes" : 2,
               "episodes": [
                  { "episodeID" : 20, "lengthMin" : 50, "minWatched" : 50 },
                  { "episodeID" : 30, "lengthMin" : 46, "minWatched" : 46 }
               ]
            }
         ]
      },
      {
         "showName": "Rita",
         "showId": 16,
         "showType": "tvseries",
         "numSeasons" : 1,
         "seriesInfo": [
            {
               "seasonNum" : 1,
               "numEpisodes" : 2,
               "episodes": [
                   { "episodeID" : 20, "lengthMin" : 65, "minWatched" : 65 },
                   { "episodeID" : 30, "lengthMin" : 60, "minWatched" : 60 }
                ]
            }
         ]
      }
   ]
}
```

Start your KVSTORE or KVLite and open the SQL.shell.

```
java -jar lib/kvstore.jar kvlite -secure-config disable
java -jar lib/sql.jar -helper-hosts localhost:5000 -store kvstore
```

Using the `load` command, run the script.

```
load -file acctstream_loaddata.sql
```

Table Hierarchies

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

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

Characteristics of parent-child tables:

- A child table inherits the primary key columns of its parent table.
- All tables in the hierarchy have the same shard key columns, which are specified in the create table statement of the root table.
- A parent table cannot be dropped before its children are dropped.
- A referential integrity constraint is not enforced in a parent-child table.

You should consider using child tables when some form of data normalization is required. Child tables can also be a good choice when modeling 1 to N relationships and also provide ACID transaction semantics when writing multiple records in a parent-child hierarchy.

2
Create

The articles in this section include examples to create various database objects.

Create Database objects

A database object is any defined object in a database that is used to store or reference data. You use a CREATE command to create a Database object. You can use a database object to hold and manipulate the data.

- Creating a namespace
- Creating a table
- Creating a region

Creating a namespace

A namespace defines a group of tables, within which all of the table names must be uniquely identified. Namespaces permit you to do table privilege management as a group operation. You can grant authorization permissions to a namespace to determine who can access both the namespace and the tables within it. Namespaces permit tables with the same name to exist in your database store. To access such tables, you can use a fully qualified table name. A fully qualified table name is a table name preceded by its namespaces, followed with a colon (:), such as `ns1:table1`.

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

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

```
CREATE NAMESPACE [IF NOT EXISTS] namespace_name
```

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

The following statement defines a namespace named `ns1`.

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

Creating a table

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

Guidelines for creating a table:

- The table definition must include at least one field definition, and exactly one primary key definition.
- The field definition specifies the name of the column, its data type, whether the column is nullable or not, an optional default value, whether or not the column is an IDENTITY column , and an optional comment. All fields (other than the PRIMARY KEY) are nullable by default.
- The syntax for the primary key specification (key_definition) specifies the primary key columns of the table as an ordered list of field names.
- The Time-To-Live (TTL) value is used in computing the expiration time of a row. Expired rows are not included in query results and are eventually removed from the table automatically by Oracle NoSQL Database. If you specify a TTL value while creating the table, it applies as the default TTL for every row inserted into this table.
- You specify the REGIONS clause if the table being created is a Multi-Region table. The REGIONS clause lists all the regions that the table should span.

Example 1: The following CREATE TABLE statement defines a `BaggageInfo` table that holds baggage information of passengers in an airline system.

```
CREATE TABLE BaggageInfo (
ticketNo LONG,
fullName STRING,
gender STRING,
contactPhone STRING,
confNo STRING,
bagInfo JSON,
PRIMARY KEY (ticketNo)
)
```

Example 2: The following CREATE TABLE statement defines a `stream_acct` table that holds data from a TV streaming application.

```
CREATE TABLE stream_acct(
acct_id INTEGER,
acct_data JSON,
PRIMARY KEY(acct_id)
)
```

Example 3: The following CREATE TABLE statement defines a `stream_acct_new` table that holds data from a TV streaming application. The rows of the table expire in 2 days.

```
CREATE TABLE stream_acct_new(
acct_id INTEGER,
```

```
acct_data JSON,
PRIMARY KEY(acct_id)) USING TTL 2 days
```

Creating a region

Oracle NoSQL Database supports Multi-Region Architecture in which you can create tables in multiple KVStores and Oracle NoSQL Database will automatically replicate inserts, updates, and deletes in a multi-directional fashion across all regions for which the table spans. Each KVStore cluster in a Multi-Region NoSQL Database setup is called a Region.

Example 1: The following CREATE REGION statement creates a remote region named `my_region1`.

```
CREATE REGION my_region1
```

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

Example 2: Create a table in a region.

```
CREATE TABLE stream_acct_region(acct_id INTEGER,
acct_data JSON,
PRIMARY KEY(acct_id)) IN REGIONS my_region1
```

> **Note:**
>
> The region `my_region1` should be set as the local region before creating the table.

Create and Manage Indexes

An index is a database structure that enables you to retrieve data from database tables efficiently. Indexes provide fast access to the rows of a table when the key(s) you are searching for is contained in the index.

An index is an ordered map in which each row of the data is called an entry. An index can be created on atomic data types, arrays, maps, JSON, and GeoJSON data.. An index can store the following special values:

- NULL
- EMPTY
- json null (It is applicable only for JSON indexes)

If you want to follow along with the examples, download the scripts baggageschema_loaddata.sql and acctstream_loaddata.sql and run it as shown below. This script creates the table used in the example and loads data into the table.

Start your KVSTORE or KVLite and open the SQL.shell.

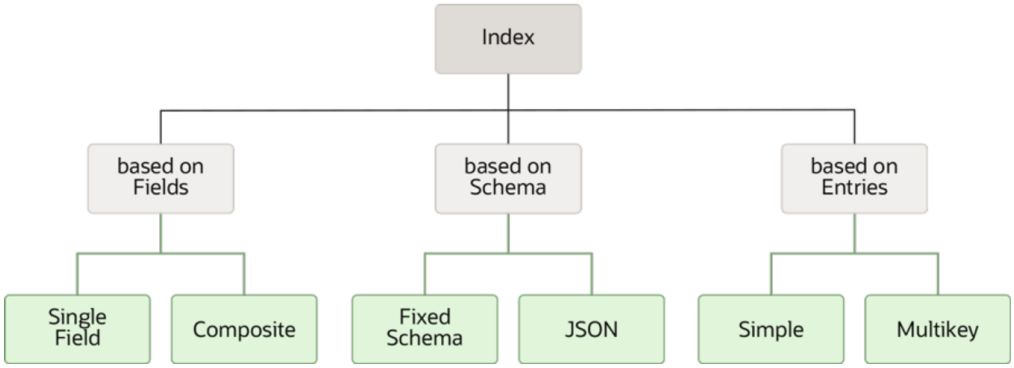
```
java -jar lib/kvstore.jar kvlite -secure-config disable
java -jar lib/sql.jar -helper-hosts localhost:5000 -store kvstore
```

Using the `load` command, run the scripts.

```
load -file baggageschema_loaddata.sql
load -file acctstream_loaddata.sql
```


Classification of Indexes

Indexes can be classified based on fields, schema, entries, or a combination of them.



Single Field Index: An index is called a single field index if it is created on only one field of a table.

Composite Index: An index is called a composite index if it is created on more than one field of a table

Fixed Schema Index: An index is called a fixed schema index if all the fields that are indexed are strongly typed data.

> **Note:**
>
> A data type is called precise if it is not one of the wild card types. Items that have precise types are said to be strongly typed.

Schema-less Index (JSON Index): An index is called a JSON index if at least one of the fields is JSON data or fields inside JSON data.

Simple Index: An index is called a simple index if for each row of data in the table, there is one entry created in the index.

Multikey Index: An index is called a multikey index if for each row of data in the table, there are multiple entries created in the index.

Creating Indexes

An index can be created using the `CREATE INDEX` command.

Create a single field index:

Example: Create an index on passengers reservation code.

```
CREATE INDEX fixedschema_conf ON baggageInfo(confNo)
```

The above is an example of a single-column fixed schema index. The index is created on the `confNo` field having `string` data type in the `baggageInfo` table.

Create a composite index:

Example : Create an index on the full name and phone number of passengers.

```
CREATE INDEX compindex_namephone ON baggageInfo(fullName,contactPhone)
```

The above is an example of a composite index. The index is created on two fields in the baggageInfo schema, on full name and the contact phone number.

> **Note:**
>
> You can have one or more fields of this index as fixed schema columns.

Create a JSON index:

An index is called a JSON index if at least one of the fields is inside JSON data. As JSON is schema-less, the data type of an indexed JSON field may be different across rows. When creating an index on JSON fields, if you are unsure what data type to expect for the JSON field, you may use the anyAtomic data type. Alternatively, you can specify one of the Oracle NoSQL Database atomic data types. You do that by declaring a data type using the AS keyword next to every index path into the JSON field.

Example 1: Create an index on the tag number of passengers bags.

```
CREATE INDEX jsonindex_tagnum ON baggageInfo(bagInfo[].tagnum as INTEGER)
```

The above is an example of a JSON index. The index is created on the `tagnum` field present in the `baginfo` JSON field in the `baggageInfo` table. Notice that you provide a data type for the `tagnum` field while creating the index.

The creation of a JSON index will fail if the associated table contains any rows with data that violate the declared data type. Similarly, after creating a JSON index, an insert/update

operation will fail if the new row does not conform to the declared data type in the JSON index.

Example 2: Create an index on the route of passengers.

```
CREATE INDEX jsonindex_routing ON baggageInfo(bagInfo[].routing as
ANYATOMIC)
```

Declaring a JSON index path as anyAtomic has the advantage of allowing the indexed JSON field to have values of various data types. The index entries are sorted in ascending order. When these values are stored in the index, they are sorted as follows:

- Numbers
- String
- boolean

However, this advantage is offset by space and CPU costs. It is because numeric values of any kind in the indexed field will be cast to Number before being stored in the index. This cast takes CPU time, and the resulting storage for the number will be larger than the original storage for the number.

Create a simple index:

An index is called a simple index if, for each row of data in the table, there is one entry created in the index. The index will return a single value that is of atomic data type or any special value (SQL NULL, JSON NULL, EMPTY). Essentially, the index paths of a simple index must not return an array or map or a nested data type.

Example: Create an index on three fields, when the bag was last seen, the last seen station and the arrival date and time.

```
CREATE INDEX simpleindex_arrival ON
baggageInfo(bagInfo[].lastSeenTimeGmt as ANYATOMIC,
bagInfo[].bagArrivalDate as ANYATOMIC, bagInfo[].lastSeenTimeStation
as ANYATOMIC)
```

The above is an example of a simple index created on a JSON document in a JSON field. The index is created on the `lastSeenTimeGmt` and `bagArrivalDate` and `lastSeenTimeStation`, all from the `bagInfo` JSON document in the info JSON field in the `baggageInfo` table. If the evaluation of a simple index path returns an empty result, the special value EMPTY is used as an index entry. In the above example, If there is no `lastSeenTimeGmt` or `bagArrivalDate` or `lastSeenTimeStation` entry in the `bagInfo` JSON document, or if there is no `bagInfo` JSON array, then the special value EMPTY is indexed.

Create a multikey index:

An index is called a multikey index if, for each row of data in the table, there are multiple entries created in the index. In a multikey index, there is at least one index path that uses an array or a nested data type. In a multikey index, for each table row, index entries are created on all the elements in arrays that are being indexed.

Example 1: Multikey index: Create an index on the series info array of the streaming account application.

```
CREATE INDEX multikeyindex1 ON stream_acct (acct_data.seriesInfo[] AS
ANYATOMIC);
```

The index is created on the `seriesInfo[]` array in the stream_acct table. Here, all the elements in the `seriesInfo[]` array in each row of the `stream_acct` table will be indexed.

Example 2: Nested multikey index: Create an index on the episode details array of the streaming account application.

An index is a nested multikey index if it is created on a field that is present inside an array which in turn is present inside another array.

```
CREATE INDEX multikeyindex2 ON stream_acct (
    acct_data.seriesInfo[].episodes[]  AS ANYATOMIC);
```

The above is an example of a nested multikey index where the field is present in an array that is present inside another array. The index is created on the `episodes[]` array in the `seriesInfo[]` array in the `acct_data` JSON of the `stream_acct` table.

Example 3: Composite multikey index:

An index is called a composite multikey index if it is created on more than one field, and at least one of those fields is multikey. A composite multikey index may have a combination of multikey index paths and simple index paths.

```
CREATE INDEX multikeyindex3 ON stream_acct (acct_data.country AS ANYATOMIC,
acct_data.seriesInfo[].episodes[]  AS ANYATOMIC);
```

The above is an example of a composite multikey index having one multikey index path and one simple index path. The index is created on the `country` field and `episodes[]`array in the `acct_data` JSON column of the `stream_acct` table.

See Specifications & Restrictions on Multikey index to learn about restrictions on multikey index.

Create an index with NO NULLS clause

You can create an index with the optional WITH NO NULLS clause. In that case, the rows with NULL and/or EMPTY values on the indexed fields will not be indexed.

```
CREATE INDEX nonull_phone ON baggageInfo (contactPhone) WITH NO NULLS;
```

- The above query creates an index on the phone number of the passengers. If some passengers do not have a phone number then those fields will not be part of the index.
- The indexes that are created with the WITH NO NULLS clause may be useful when the data contain a lot of NULL and/or EMPTY values on the indexed fields. It will reduce the time and space overhead during indexing.
- However, the use of such indexes by queries is restricted. If an index is created with the WITH NO NULLS clause, IS NULL, and NOT EXISTS predicates cannot be used as index predicates for that index.

- In fact, such an index can be used by a query only if the query has an index predicate for each of the indexed fields.

Create an index with unique keys per row

You can create an index with unique keys per row property.

```
CREATE INDEX idx_showid ON
stream_acct(acct_data.contentStreamed[].showId AS INTEGER)
WITH UNIQUE KEYS PER ROW;
```

In the above query, an index is created on `showId` and there cannot be duplicate `showId` for a single `contentStreamed` array. This informs the query processor that for any streaming user, the `contentStreamed` array cannot contain two or more shows with the same show id. The restriction is necessary because if duplicate show ids existed, they wouldn't be included in the index. If you insert a row with the same `showId` two or more items in a single `contentStreamed` array, an error is thrown and the insert operation is not successful.

Optimization in the query run time :

When you create an index with unique keys per row, the index would contain fewer entries than the number of elements in the `contentStreamed` array. You could write an efficient query to use this index. The use of such an index by the query would yield fewer results from the FROM clause than if the index was not used.

View Index

You can view the indexes in your database.

SHOW INDEXES

The SHOW INDEXES statement provides the list of indexes present in the specified table. If you want the output to be in JSON format, you can specify the optional AS JSON.

Example 1: List indexes on the `BaggageInfo` table.

```
SHOW INDEXES ON baggageInfo

indexes
  jsonindex_routing
  jsonindex_tagnum
  simpleindex_arrival
  nonull_phone
```

Example 2: List indexes on the `BaggageInfo` table in JSON format.

```
SHOW AS JSON INDEXES ON baggageInfo
{"indexes"  :
["jsonindex_routing","jsonindex_tagnum","simpleindex_arrival"]}
```

DESCRIBE INDEX

The DESCRIBE INDEX statement defines the specified index on a table. If you want the output to be in JSON format, you can specify the optional AS JSON.

The description for the index contains the following information:

- Name of the table on which the index is defined.
- Name of the index.
- Type of index. Whether the index is primary index or secondary index.
- Whether the index is multikey? If the index is multikey then 'Y' is displayed. Otherwise, 'N' is displayed.
- List of fields on which the index is defined.
- The declared type of the index.
- Description of the index.

Example 1: Describe the index `multikeyindex3`.

```
DESCRIBE INDEX multikeyindex3 ON stream_acct;
+------------+---------------+----------+----------
+-----------------------------------+-------------+-------------+
table       |  name          | type      | multiKey |
fields                            | declaredType | description
+------------+---------------+----------+----------
+-----------------------------------+-------------+-------------+
stream_acct | multikeyindex3 | SECONDARY | Y        |
acct_data.country                 | ANY_ATOMIC
|                                   | |           |
            |                |           |          |
acct_data.seriesInfo[].episodes[] | ANY_ATOMIC   |
+------------+---------------+----------+----------
+-----------------------------------+-------------+-------------+
```

Example 2: Describe the index `multikeyindex2` in JSON format.

```
DESCRIBE AS JSON INDEX multikeyindex2 ON stream_acct;
{
  "name" : "multikeyindex2",
  "type" :  "secondary",
  "fields" :["acct_data.seriesInfo[].episodes[]"],
  "types" : ["ANY_ATOMIC"],
  "withNoNulls" : false,
  "withUniqueKeysPerRow" : false
}
```

Drop Index

You can drop an index from your database when you no longer need it.

The DROP INDEX removes the specified index from the database. If an index with the given name does not exist, then the statement fails, and an error is reported. If the optional IF

EXISTS clause is used in the DROP INDEX statement, and if an index with the same name does not exist, then the statement will not execute, and no error is reported.

Example: Drop the index `multikeyindex1`.

```
DROP INDEX multikeyindex1 ON stream_acct
```

3
Manage

The articles in this section provide steps on how to manage various database objects.

Namespace Management

A namespace defines a group of tables, within which all of the table names must be uniquely identified. Namespaces permit you to do table privilege management as a group operation.

- Namespace Resolution
- Manage Namespaces
- Namespace scoped privileges

Namespace Resolution

You can grant authorization permissions to a namespace to determine who can access both the namespace and the tables within it.

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

- – If the `table_name` contains a namespace name, no resolution is needed, because a qualified table name uniquely identifies a table.
  – If you don't specify a namespace name explicitly, the namespace used is the one contained in the `ExecuteOptions` instance that is given as input to the `executeSync()`, `execute()`, or `prepare()` methods of TableAPI.
  – If `ExecuteOptions` doesn't specify a namespace, the default sysdefault namespace is used.

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

Manage Namespaces

SHOW NAMESPACES

The SHOW NAMESPACES statement provides the list of namespaces in the system. You can specify **AS JSON** if you want the output to be in JSON format.

Example 1: The following statement lists the namespaces present in the system.

```
SHOW NAMESPACES
```

Output:

```
namespaces
  sysdefault
```

Example 2: The following statement lists the namespaces present in the system in JSON format.

```
SHOW AS JSON NAMESPACES
```

Output:

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

DROP NAMESPACE

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

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

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

The following statement removes the namespace named **ns1**.

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

Namespace scoped privileges

You can add one or more namespaces to your store, create tables within them, and grant permission for users to access namespaces and tables. You can grant the following permissions to users.

System-scoped Privileges:

- CREATE_ANY_NAMESPACE
- DROP_ANY_NAMESPACE

Namespace-scoped privileges:

- CREATE_TABLE_IN_NAMESPACE
- DROP_TABLE_IN_NAMESPACE
- EVOLVE_TABLE_IN_NAMESPACE
- CREATE_INDEX_IN_NAMESPACE
- DROP_INDEX_IN_NAMESPACE

The following example creates a namespace, creates a table in the namespace, adds data to the table, and drops the namespace,. You also see the various privilege on the namespace being assigned to a role/user and revoked later.

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

> **Note:**
>
> You can save all of the above commands as a **sql** script and execute it in a single command. If you want to execute any of the above commands outside of a SQL prompt, remove the semi colon at then end.

Inserting, Modifying, and Deleting Data

You can perform various data manipulation operations in your table. You can add data, modify an existing data and remove data.

If you want to follow along with the examples, download the script baggageschema_loaddata.sql and execute it as shown below. This script creates the table used in the example and loads data into the table.

Start your KVSTORE or KVLite and open the SQL.shell.

```
java -jar lib/kvstore.jar kvlite -secure-config disable
java -jar lib/sql.jar -helper-hosts localhost:5000 -store kvstore
```

Using the `load` command, execute the script.

```
load -file baggageschema_loaddata.sql
```

- Insert data
- Update Data
- Modify JSON data
- Delete Data

Insert data

The INSERT statement is used to construct a new row and add it to a specified table. Optional column(s) may be specified after the table name. This list contains the column names for a subset of the table's columns. The subset must include all the primary key columns. If no columns list is present, the default columns list is the one containing all the columns of the table, in the order, they are specified in the CREATE TABLE statement.

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

Example 1: Inserting a row into `BaggageInfo` table providing all column values:

```
INSERT INTO BaggageInfo VALUES(
1762392196147,
"Birgit Naquin",
"M",
"165-742-5715",
"QD1L0T",
[ {
    "id" : "7903989918469",
    "tagNum" : "17657806240229",
    "routing" : "JFK/MAD",
    "lastActionCode" : "OFFLOAD",
    "lastActionDesc" : "OFFLOAD",
    "lastSeenStation" : "MAD",
    "flightLegs" : [ {
      "flightNo" : "BM495",
      "flightDate" : "2019-03-07T07:00:00Z",
      "fltRouteSrc" : "JFK",
      "fltRouteDest" : "MAD",
      "estimatedArrival" : "2019-03-07T14:00:00Z",
      "actions" : [ {
        "actionAt" : "MAD",
        "actionCode" : "Offload to Carousel at MAD",
        "actionTime" : "2019-03-07T13:54:00Z"
      }, {
        "actionAt" : "JFK",
        "actionCode" : "ONLOAD to MAD",
        "actionTime" : "2019-03-07T07:00:00Z"
      }, {
        "actionAt" : "JFK",
        "actionCode" : "BagTag Scan at JFK",
        "actionTime" : "2019-03-07T06:53:00Z"
      }, {
        "actionAt" : "JFK",
        "actionCode" : "Checkin at JFK",
        "actionTime" : "2019-03-07T05:03:00Z"
      } ]
    } ],
    "lastSeenTimeGmt" : "2019-03-07T13:51:00Z",
    "bagArrivalDate" : "2019-03-07T13:51:00Z"
  } ]
)
```

Example 2: Skipping some data while doing an INSERT statement by specifying the DEFAULT clause.

You can skip the data of some columns by specifying "DEFAULT".

```
INSERT INTO BaggageInfo VALUES(
1762397286805,
"Bonnie Williams",
DEFAULT,
DEFAULT,
"CZ1O5I",
[ {
    "id" : "79039899129693",
    "tagNum" : "17657806216554",
    "routing" : "SFO/ORD/FRA",
    "lastActionCode" : "OFFLOAD",
    "lastActionDesc" : "OFFLOAD",
    "lastSeenStation" : "FRA",
    "flightLegs" : [ {
      "flightNo" : "BM572",
      "flightDate" : "2019-03-02T05:00:00Z",
      "fltRouteSrc" : "SFO",
      "fltRouteDest" : "ORD",
      "estimatedArrival" : "2019-03-02T09:00:00Z",
      "actions" : [ {
        "actionAt" : "SFO",
        "actionCode" : "ONLOAD to ORD",
        "actionTime" : "2019-03-02T05:24:00Z"
      }, {
        "actionAt" : "SFO",
        "actionCode" : "BagTag Scan at SFO",
        "actionTime" : "2019-03-02T04:52:00Z"
      }, {
        "actionAt" : "SFO",
        "actionCode" : "Checkin at SFO",
        "actionTime" : "2019-03-02T03:28:00Z"
      } ]
    }, {
      "flightNo" : "BM582",
      "flightDate" : "2019-03-02T05:24:00Z",
      "fltRouteSrc" : "ORD",
      "fltRouteDest" : "FRA",
      "estimatedArrival" : "2019-03-02T13:24:00Z",
      "actions" : [ {
        "actionAt" : "FRA",
        "actionCode" : "Offload to Carousel at FRA",
        "actionTime" : "2019-03-02T13:20:00Z"
      }, {
        "actionAt" : "ORD",
        "actionCode" : "ONLOAD to FRA",
        "actionTime" : "2019-03-02T12:54:00Z"
      }, {
        "actionAt" : "ORD",
        "actionCode" : "OFFLOAD from ORD",
        "actionTime" : "2019-03-02T12:30:00Z"
```

```
      } ]
    } ],
    "lastSeenTimeGmt" : "2019-03-02T13:18:00Z",
    "bagArrivalDate" : "2019-03-02T13:18:00Z"
  } ]
)
```

Example 3: Specifying column names and skipping columns in the insert statement.

If you have data only for some columns of a table, you can specify the name of the columns in the INSERT clause and then specify the corresponding values in the "VALUES" clause.

```
INSERT INTO BaggageInfo(ticketNo, fullName,confNo,bagInfo) VALUES(
1762355349471,
"Bryant Weber",
"LI7N1W",
[ {
    "id" : "79039899149056",
    "tagNum" : "17657806234185",
    "routing" : "MEL/LAX/MIA",
    "lastActionCode" : "OFFLOAD",
    "lastActionDesc" : "OFFLOAD",
    "lastSeenStation" : "MIA",
    "flightLegs" : [ {
      "flightNo" : "BM114",
      "flightDate" : "2019-03-01T12:00:00Z",
      "fltRouteSrc" : "MEL",
      "fltRouteDest" : "LAX",
      "estimatedArrival" : "2019-03-02T02:00:00Z",
      "actions" : [ {
        "actionAt" : "MEL",
        "actionCode" : "ONLOAD to LAX",
        "actionTime" : "2019-03-01T12:20:00Z"
      }, {
        "actionAt" : "MEL",
        "actionCode" : "BagTag Scan at MEL",
        "actionTime" : "2019-03-01T11:52:00Z"
      }, {
        "actionAt" : "MEL",
        "actionCode" : "Checkin at MEL",
        "actionTime" : "2019-03-01T11:43:00Z"
      } ]
    }, {
      "flightNo" : "BM866",
      "flightDate" : "2019-03-01T12:20:00Z",
      "fltRouteSrc" : "LAX",
      "fltRouteDest" : "MIA",
      "estimatedArrival" : "2019-03-02T16:21:00Z",
      "actions" : [ {
        "actionAt" : "MIA",
        "actionCode" : "Offload to Carousel at MIA",
        "actionTime" : "2019-03-02T16:18:00Z"
      }, {
        "actionAt" : "LAX",
```

```
        "actionCode" : "ONLOAD to MIA",
        "actionTime" : "2019-03-02T16:12:00Z"
      }, {
        "actionAt" : "LAX",
        "actionCode" : "OFFLOAD from LAX",
        "actionTime" : "2019-03-02T16:02:00Z"
      } ]
    } ],
    "lastSeenTimeGmt" : "2019-03-02T16:09:00Z",
    "bagArrivalDate" : "2019-03-02T16:09:00Z"
  } ]
)
```

Update Data

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

- The `SET` clause consists of two expressions: the target expression and the new-value expression. The target expression returns the items to be updated. The new-value expression may return zero or more items. If it returns an empty result, the SET is a no-op. If it returns more than one item, the items are enclosed inside a newly constructed array (this is the same as the way the SELECT clause treats multi-valued expressions in the select list)) So, effectively, the result of the new-value expression contains at most one item.
- The `WHERE` clause specifies what row to update. In the current implementation, only single-row updates are allowed, so the `WHERE` clause must specify a complete primary key.
- There is an optional `RETURNING` clause which 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.

Example 1: Simple example to change the column values.

You are updating some column values for a given ticket number.

```
UPDATE BaggageInfo
SET contactPhone = "823-384-1964",
confNo = "LE6J4Y"
WHERE ticketNo = 1762344493810
```

Example 2: Update row data and fetch the values with a RETURNING clause.

You could use the RETURNING clause to fetch back the data after the UPDATE clause is executed.

```
UPDATE BaggageInfo
SET contactPhone = "823-384-1964",
confNo = "LE6J4Y"
WHERE ticketNo = 1762344493810 RETURNING *
```

Output:


```
{"ticketNo":1762344493810,"fullName":"Adam
Phillips","gender":"M","contactPhone":"823-384-1964",
"confNo":"LE6J4Y",
"bagInfo":{"bagInfo":[{"bagArrivalDate":"2019.02.02 at 03:13:00
AEDT","flightLegs":
[{"actions":[{"actionAt":"MIA","actionCode":"ONLOAD to
LAX","actionTime":"2019.02.01 at 01:13:00 EST"},
{"actionAt":"MIA","actionCode":"BagTag Scan at
MIA","actionTime":"2019.02.01 at 00:47:00 EST"},
{"actionAt":"MIA","actionCode":"Checkin at
MIA","actionTime":"2019.01.31 at 23:38:00 EST"}],
"estimatedArrival":"2019.02.01 at 03:00:00
PST","flightDate":"2019.02.01 at 01:00:00 EST",
"flightNo":"BM604","fltRouteDest":"LAX","fltRouteSrc":"MIA"},
{"actions":
[{"actionAt":"MEL","actionCode":"Offload to Carousel at
MEL","actionTime":"2019.02.02 at 03:15:00 AEDT"},
{"actionAt":"LAX","actionCode":"ONLOAD to
MEL","actionTime":"2019.02.01 at 07:35:00 PST"},
{"actionAt":"LAX","actionCode":"OFFLOAD from
LAX","actionTime":"2019.02.01 at 07:18:00 PST"}],
"estimatedArrival":"2019.02.02 at 03:15:00
AEDT","flightDate":"2019.01.31 at 22:13:00 PST",
"flightNo":"BM667","fltRouteDest":"MEL","fltRouteSrc":"LAX"}],"id":"790
39899165297",
"lastActionCode":"OFFLOAD","lastActionDesc":"OFFLOAD","lastSeenStation"
:"MEL",
"lastSeenTimeGmt":"2019.02.02 at 03:13:00 AEDT","routing":"MIA/LAX/
MEL","tagNum":"17657806255240"}]}}
```


Modify JSON data

While updating JSON data, in addition to `WHERE`, `SET` and `RETURNING` clause, the following clauses can be used..

- 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 `PUT` clause is used primarily to add new fields to a JSON document. It consists of a target expression, which should normally return one or more fields to be inserted into the target JSON document.
- The `REMOVE` clause consists of a single target expression, which computes the items to be removed.

Example 1: Update table and add data in a JSON object

Add elements to the action array (at a given array element) for a particular flight Leg of a passenger. By default, the element is added at the end. If a number is specified, it is

inserted in that position. In the example below, you want the new element to be added in the 2nd position.

```
UPDATE BaggageInfo bag
ADD bag.bagInfo[0].flightLegs[0].actions 2  {"actionAt" : "LAX",
         "actionCode" : "WAITING at LAX",
         "actionTime" : "2019-02-01T06:13:00Z"}
WHERE ticketNo=1762344493810
RETURNING *
```

Example 2: Update table and update data from a JSON object.

You could update the data from a JSON object using the SET clause. Here the second element of the actions array is updated with new values for a given ticket number.

```
UPDATE BaggageInfo bag
SET bag.bagInfo[0].flightLegs[0].actions[2]=
{"actionAt" : "LAX",
"actionCode" : "STILL WAITING at LAX",
"actionTime" : "2019-02-01T06:15:00Z"}
WHERE ticketNo=1762344493810 RETURNING *
```

Example 3: Update table and remove data in a JSON object.

You can use the `REMOVE` clause to remove a given element from an array. You need to specify which element of the array needs to be removed using the index of the array.

```
UPDATE BaggageInfo bag
REMOVE bag.bagInfo[0].flightLegs[0].actions[1]
WHERE ticketNo=1762344493810
RETURNING *
```

Delete Data

The DELETE statement is used to remove from a table a set of rows satisfying a condition. The condition is specified in a WHERE clause that behaves the same way as in the SELECT expression. The result of the DELETE statement depends on whether a RETURNING clause is present or not. Without a RETURNING clause the DELETE returns the number of rows deleted. Otherwise, for each deleted row the expressions following the RETURNING clause are computed the same way as in the SELECT clause and the result is returned to the application.

Example 1: Delete data from a table with a simple WHERE clause.

You delete the data corresponding to a user with a given fullname.

```
DELETE FROM BaggageInfo
WHERE fullName = "Bonnie Williams"
```

Example 2: Delete data from a table with a RETURNING clause.

The RETURNING clause fetches the details of the row to be deleted. In the example below, you are fetching the full name and conf number corresponding to a ticket number which will be deleted.

```
DELETE FROM BaggageInfo
WHERE ticketNo = 1762392196147
RETURNING fullName,confNo
```

Output:

```
{"fullName":"Birgit Naquin","confNo":"QD1L0T"}
```

> **Note:**
>
> If any error occurs during the execution of a DELETE statement, there is a possibility that some rows will be deleted and some not. The system does not keep track of what rows got deleted and what rows are not yet deleted. This is because Oracle NoSQL Database focuses on low latency operations. Long-running operations across shards are not coordinated using a two-phase commit and lock mechanism. In such cases, it is recommended that the application re-run the DELETE statement.

Managing Tables & Regions

You will first create a sample table. Then you will learn different ways to alter the table. At the end of the section, you will drop the table. You will also learn to view the existing regions and drop a particular region.

Start your KVSTORE or KVLite and open the SQL shell.

```
java -jar lib/kvstore.jar kvlite -secure-config disable
java -jar lib/sql.jar -helper-hosts localhost:5000 -store kvstore
```

- Alter Table
- Drop Table
- Manage regions

Alter Table

You can use the alter table command to perform the following operations.

- Add schema fields to the table schema
- Remove schema fields from the table schema
- Add a region
- Remove a region
- Modify the Time-To-Live value of the table

> **Note:**
>
> You can specify only one type of operation in a single command. For example, you cannot remove a schema field and set the TTL value together.

Create a sample table :

```
CREATE TABLE demo_acct(
acct_id INTEGER,
acct_data JSON,
PRIMARY KEY(acct_id)
)
```

Example 1: Add schema field to the table schema.

```
ALTER TABLE demo_acct(ADD acct_balance INTEGER)
```

Explanation: Adding a field does not affect the existing rows in the table. If a field is added, its default value or NULL will be used as the value of this field in existing rows that do not contain it. The field to add maybe 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.

Example 2: Remove schema fields in the table schema.

```
ALTER TABLE demo_acct(DROP acct_balance)
```

Explanation: You can drop any field in the schema other than the primary key. If you try removing the primary key field, you get an error as shown below.

```
ALTER TABLE demo_acct(DROP acct_id)
```

Output(showing error):

```
Error handling command ALTER TABLE demo_acct(DROP acct_id):
Error: at (1, 27) Cannot remove a primary key field: acct_id
```

Example 3: Add a region

The add regions clause lets you link an existing Multi-Region Table (MR Table) with new regions in a multi-region Oracle NoSQL Database environment. You use this clause to expand MR Tables to new regions.

Associate a new region with an existing MR Table using the DDL command shown below.

```
ALTER TABLE <table_name> ADD REGIONS <region_name>
```

Explanation: Here, table_name is an MR table and region_name is an existing region.

Example 4: Remove a region

The drop regions clause lets you disconnect an existing MR Table from a participating region in a multi-region Oracle NoSQL Database environment. You use this clause to contract MR Tables to fewer regions.

To remove an MR Table from a specific region in a Multi-Region NoSQL Database setup, you must run the following steps from all the other participating regions.

```
ALTER TABLE <table_name> DROP REGIONS <comma_separated_list_of_regions>
```

Here, table_name is a MR Table and comma_separated_list_of_regions is a list of regions to be dropped.

Example 5: Modify the Time-To-Live value of the table

Time-to-Live (TTL) is a mechanism that allows you to set a time frame on table rows, after which the rows expire automatically, and are no longer available. By default, every table that you create has a TTL value of zero, indicating that it has no expiration time.

You can use ALTER TABLE command to change this value for any table. You can specify the TTL with a number, followed by either `HOURS` or `DAYS`.

```
ALTER TABLE demo_acct USING TTL 5 days
```

> **Note:**
>
> Altering the TTL value for a table does not change the TTL value for existing rows in the table. Rather, it will only change the default TTL value placed in rows created subsequent to the alter table. To modify the TTL of every record in a table, you must iterate through each record of the table and update its TTL value.

Drop Table

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 then this statement fails. You don't get an error if the optional IF EXISTS clause is specified and the table does not exist.

```
DROP TABLE demo_acct
```

> **Note:**
>
> To drop a MR Table, first drop all of its child tables. Otherwise, the DROP statement results in an error.

Manage regions

The show regions statement provides the list of regions present in the Multi-Region Oracle NoSQL Database. You need to specify "AS JSON" if you want the output to be in JSON format.

Example 1: The following statement lists all the existing regions.

```
SHOW REGIONS
```

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

```
SHOW AS JSON REGIONS
```

In a Multi-Region Oracle NoSQL Database environment, the drop region statement removes the specified remote region from the local region. See Set up Multi-Region Environment for more details on the local regions and remote regions in a Multi-Region setup.

> **Note:**
>
> This region must be different from the local region where the command is executed.

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

```
DROP REGION my_region1
```

4
Develop

The articles in this section provide steps on how to use SQL and write queries. It covers information about different complex data types. It also covers how to use indexes for query optimization.

Simple SELECT queries

If you want to follow along with the examples, download the script baggageschema_loaddata.sql and run it as shown below. This script creates the table used in the example and loads data into the table.

Start your KVSTORE or KVLite and open the SQL.shell.

```
java -jar lib/kvstore.jar kvlite -secure-config disable
java -jar lib/sql.jar -helper-hosts localhost:5000 -store kvstore
```

Using the `load` command, run the script.

```
load -file baggageschema_loaddata.sql
```


Fetch column data

You can choose columns from a table. To do so, list the names of the desired table columns after SELECT in the statement. You give the name of the table after the FROM clause. To retrieve data from a child table, use dot notation, such as **parent.child**. To choose all table columns, use the asterisk (*) wildcard character. The SELECT statement can also contain computational expressions based on the values of existing columns.

Example 1: Choose all data from the table `BaggageInfo`.

```
SELECT * FROM BaggageInfo
```

Explanation: The `BaggageInfo` schema has some fixed static fields and a JSON column. The static fields are ticket number, full name, gender, contact phone, and confirmation number. The bag information is stored as JSON and is populated with an array of documents.

Output (displaying only a row of the result for brevity):

```
{"ticketNo":1762330498104,"fullName":"Michelle
Payne","gender":"F","contactPhone":"575-781-6240","confNo":"RL3J4Q",
"bagInfo":[{
```

```
   "bagArrivalDate":"2019-02-02T23:59:00Z",
   "flightLegs":[
      {"actions":[
           {"actionAt":"SFO","actionCode":"ONLOAD to
IST","actionTime":"2019-02-02T12:10:00Z"},
           {"actionAt":"SFO","actionCode":"BagTag Scan at
SFO","actionTime":"2019-02-02T11:47:00Z"},
           {"actionAt":"SFO","actionCode":"Checkin at
SFO","actionTime":"2019-02-02T10:01:00Z"}],
           "estimatedArrival":"2019-02-03T01:00:00Z",
           "flightDate":"2019-02-02T12:00:00Z",
           "flightNo":"BM318",
           "fltRouteDest":"IST",
           "fltRouteSrc":"SFO"},
      {"actions":[
           {"actionAt":"IST","actionCode":"ONLOAD to
ATH","actionTime":"2019-02-03T13:06:00Z"},
           {"actionAt":"IST","actionCode":"BagTag Scan at
IST","actionTime":"2019-02-03T12:48:00Z"},
           {"actionAt":"IST","actionCode":"OFFLOAD from
IST","actionTime":"2019-02-03T13:00:00Z"}],
           "estimatedArrival":"2019-02-03T12:12:00Z",
           "flightDate":"2019-02-02T13:10:00Z",
           "flightNo":"BM696",
           "fltRouteDest":"ATH",
           "fltRouteSrc":"IST"},
      {"actions":[
           {"actionAt":"JTR","actionCode":"Offload to Carousel at
JTR","actionTime":"2019-02-03T00:06:00Z"},
           {"actionAt":"ATH","actionCode":"ONLOAD to
JTR","actionTime":"2019-02-03T00:13:00Z"},
           {"actionAt":"ATH","actionCode":"OFFLOAD from
ATH","actionTime":"2019-02-03T00:10:00Z"}],
           "estimatedArrival":"2019-02-03T00:12:00Z",
           "flightDate":"2019-2-2T12:10:00Z",
           "flightNo":"BM665",
           "fltRouteDest":"JTR",
           "fltRouteSrc":"ATH"}],
     "id":"79039899186259",
     "lastActionCode":"OFFLOAD",
     "lastActionDesc":"OFFLOAD",
     "lastSeenStation":"JTR",
     "lastSeenTimeGmt":"2019-02-02T23:59:00Z",
     "routing":"SFO/IST/ATH/JTR",
     "tagNum":"17657806247861"}
]}
```


Example 2: To choose specific column(s) from the table `BaggageInfo`, include the column names as a comma-separated list in the SELECT statement.

```
SELECT fullName, contactPhone, gender FROM BaggageInfo
```

Explanation: You want to display the values of three static fields - full name, phone number, and gender.

Output:


```
{"fullName":"Lucinda Beckman","contactPhone":"364-610-4444","gender":"M"}
{"fullName":"Adelaide Willard","contactPhone":"421-272-8082","gender":"M"}
{"fullName":"Raymond Griffin","contactPhone":"567-710-9972","gender":"F"}
{"fullName":"Elane Lemons","contactPhone":"600-918-8404","gender":"F"}
{"fullName":"Zina Christenson","contactPhone":"987-210-3029","gender":"M"}
{"fullName":"Zulema Martindale","contactPhone":"666-302-0028","gender":"F"}
{"fullName":"Dierdre Amador","contactPhone":"165-742-5715","gender":"M"}
{"fullName":"Henry Jenkins","contactPhone":"960-428-3843","gender":"F"}
{"fullName":"Rosalia Triplett","contactPhone":"368-769-5636","gender":"F"}
{"fullName":"Lorenzo Phil","contactPhone":"364-610-4444","gender":"M"}
{"fullName":"Gerard Greene","contactPhone":"395-837-3772","gender":"M"}
{"fullName":"Adam Phillips","contactPhone":"893-324-1064","gender":"M"}
{"fullName":"Doris Martin","contactPhone":"289-564-3497","gender":"F"}
{"fullName":"Joanne Diaz","contactPhone":"334-679-5105","gender":"F"}
{"fullName":"Omar Harvey","contactPhone":"978-191-8550","gender":"F"}
{"fullName":"Fallon Clements","contactPhone":"849-731-1334","gender":"M"}
{"fullName":"Lisbeth Wampler","contactPhone":"796-709-9501","gender":"M"}
{"fullName":"Teena Colley","contactPhone":"539-097-5220","gender":"M"}
{"fullName":"Michelle Payne","contactPhone":"575-781-6240","gender":"F"}
{"fullName":"Mary Watson","contactPhone":"131-183-0560","gender":"F"}
{"fullName":"Kendal Biddle","contactPhone":"619-956-8760","gender":"F"}
```


Substituting column names in a query

You can use a different name for a column during a SELECT statement. Substituting a name in a query does not change the column name, but uses the substitute in the data returned.

Example: The following query returns the phone number as `CONTACTAT` in the result.

```
SELECT contactPhone AS CONTACTAT FROM BaggageInfo
```

Explanation: Here you want to fetch the contact phone of the passengers and display it as *CONTACTAT*.

Output:


```
{"CONTACTAT":"960-428-3843"}
{"CONTACTAT":"368-769-5636"}
{"CONTACTAT":"364-610-4444"}
{"CONTACTAT":"395-837-3772"}
{"CONTACTAT":"893-324-1064"}
{"CONTACTAT":"289-564-3497"}
{"CONTACTAT":"334-679-5105"}
{"CONTACTAT":"978-191-8550"}
{"CONTACTAT":"849-731-1334"}
{"CONTACTAT":"796-709-9501"}
{"CONTACTAT":"539-097-5220"}
{"CONTACTAT":"575-781-6240"}
{"CONTACTAT":"131-183-0560"}
{"CONTACTAT":"619-956-8760"}
{"CONTACTAT":"364-610-4444"}
{"CONTACTAT":"421-272-8082"}
```

```
{"CONTACTAT":"567-710-9972"}
{"CONTACTAT":"600-918-8404"}
{"CONTACTAT":"987-210-3029"}
{"CONTACTAT":"666-302-0028"}
{"CONTACTAT":"165-742-5715"}
```

You can combine columns using the concatenation operator "||" as shown below.

Example: For all customers, fetch the last place where the bag was seen and the time when it was seen.

Approach 1: Use the concatenation operator and fetch column names and static text as output of the SELECT command.

```
SELECT "The bag was last seen at " ||
bag.bagInfo[].lastSeenStation || " on " ||
bag.bagInfo[].bagArrivalDate AS Bag_Details FROM BaggageInfo bag
```

Output:

```
{"Bag_Details":"The bag was last seen at BZN on 2019-03-15T10:13:00Z"}
{"Bag_Details":"The bag was last seen at MEL on 2019-02-04T10:08:00Z"}
{"Bag_Details":"The bag was last seen at MEL on 2019-02-25T20:15:00Z"}
{"Bag_Details":"The bag was last seen at MAD on 2019-03-07T13:51:00Z"}
{"Bag_Details":"The bag was last seen at FRA on 2019-03-02T13:18:00Z"}
{"Bag_Details":"The bag was last seen at VIE on 2019-02-12T07:04:00Z"}
{"Bag_Details":"The bag was last seen at JTRJTR on
2019-03-12T15:05:00Z2019-03-12T16:25:00Z"}
{"Bag_Details":"The bag was last seen at JTR on 2019-03-07T16:01:00Z"}
{"Bag_Details":"The bag was last seen at MEL on 2019-02-01T16:13:00Z"}
{"Bag_Details":"The bag was last seen at MXP on 2019-03-22T10:17:00Z"}
{"Bag_Details":"The bag was last seen at MEL on 2019-02-16T16:13:00Z"}
{"Bag_Details":"The bag was last seen at MIA on 2019-03-02T16:09:00Z"}
{"Bag_Details":"The bag was last seen at BZN on 2019-02-21T14:08:00Z"}
{"Bag_Details":"The bag was last seen at SGN on 2019-02-10T10:01:00Z"}
{"Bag_Details":"The bag was last seen at JTR on 2019-02-02T23:59:00Z"}
{"Bag_Details":"The bag was last seen at BLR on 2019-03-14T06:22:00Z"}
{"Bag_Details":"The bag was last seen at VIE on 2019-03-05T12:00:00Z"}
{"Bag_Details":"The bag was last seen at JTR on 2019-03-12T15:05:00Z"}
{"Bag_Details":"The bag was last seen at SEA on 2019-02-15T21:21:00Z"}
{"Bag_Details":"The bag was last seen at HKG on 2019-02-03T08:09:00Z"}
{"Bag_Details":"The bag was last seen at HKG on 2019-02-13T11:15:00Z"}
```

The result is cluttered if there is more than one bag per customer/reservation number as shown above.

Approach 2: You can overcome this issue by printing as the value of elements of the `bagInfo` array as shown below.

```
SELECT "The bag was last seen at " || [bag.bagInfo[].lastSeenStation]
|| " on " ||
[bag.bagInfo[].bagArrivalDate] AS Bag_Details FROM BaggageInfo bag
```

> **Note:**
>
> Column names and static text can also be concatenated using the "||" operator.

Explanation: You are concatenating a part of the document in the `bagInfo` JSON with various static text and displaying it as elements of an array.

Output:

```
{"Bag_Details":"The bag was last seen at [\"MIA\"] on
[\"2019-03-02T16:09:00Z\"]"}
{"Bag_Details":"The bag was last seen at [\"BZN\"] on
[\"2019-02-21T14:08:00Z\"]"}
{"Bag_Details":"The bag was last seen at [\"SGN\"] on
[\"2019-02-10T10:01:00Z\"]"}
{"Bag_Details":"The bag was last seen at [\"HKG\"] on
[\"2019-02-13T11:15:00Z\"]"}
{"Bag_Details":"The bag was last seen at [\"JTR\"] on
[\"2019-02-02T23:59:00Z\"]"}
{"Bag_Details":"The bag was last seen at [\"BLR\"] on
[\"2019-03-14T06:22:00Z\"]"}
{"Bag_Details":"The bag was last seen at [\"VIE\"] on
[\"2019-03-05T12:00:00Z\"]"}
{"Bag_Details":"The bag was last seen at [\"JTR\"] on
[\"2019-03-12T15:05:00Z\"]"}
{"Bag_Details":"The bag was last seen at [\"SEA\"] on
[\"2019-02-15T21:21:00Z\"]"}
{"Bag_Details":"The bag was last seen at [\"HKG\"] on
[\"2019-02-03T08:09:00Z\"]"}
{"Bag_Details":"The bag was last seen at [\"BZN\"] on
[\"2019-03-15T10:13:00Z\"]"}
{"Bag_Details":"The bag was last seen at [\"MEL\"] on
[\"2019-02-04T10:08:00Z\"]"}
{"Bag_Details":"The bag was last seen at [\"MEL\"] on
[\"2019-02-25T20:15:00Z\"]"}
{"Bag_Details":"The bag was last seen at [\"MAD\"] on
[\"2019-03-07T13:51:00Z\"]"}
{"Bag_Details":"The bag was last seen at [\"FRA\"] on
[\"2019-03-02T13:18:00Z\"]"}
{"Bag_Details":"The bag was last seen at [\"VIE\"] on
[\"2019-02-12T07:04:00Z\"]"}
{"Bag_Details":"The bag was last seen at [\"JTR\",\"JTR\"] on
[\"2019-03-12T15:05:00Z\",
\"2019-03-12T16:25:00Z\"]"}
{"Bag_Details":"The bag was last seen at [\"JTR\"] on
[\"2019-03-07T16:01:00Z\"]"}
{"Bag_Details":"The bag was last seen at [\"MEL\"] on
[\"2019-02-01T16:13:00Z\"]"}
{"Bag_Details":"The bag was last seen at [\"MXP\"] on
[\"2019-03-22T10:17:00Z\"]"}
{"Bag_Details":"The bag was last seen at [\"MEL\"] on
[\"2019-02-16T16:13:00Z\"]"}
```

Filtering results in a query

You can filter query results by specifying a filter condition in the WHERE clause. Typically, a filter condition consists of one or more comparison expressions connected through logical operators AND or OR. The following comparison operators are also supported: =, !=, >, >=, <, and <= .

Example 1: Find the tag number of a passenger's baggage along with the passenger's full name for a given reservation number **FH7G1W**.

```
SELECT bag.fullName, bag.bagInfo[].tagNum FROM BaggageInfo bag
WHERE bag.confNo="FH7G1W"
```

Explanation: You fetch the tag number corresponding to a given reservation number.

Output:

```
{"fullName":"Rosalia Triplett","tagNum":"17657806215913"}
```

Note:

For better understanding, the row of data with all the static fields and the bagInfo JSON is shown below.


```
"ticketNo" : 1762344493810,
"fullName" : "Adam Phillips",
"gender" : "M",
"contactPhone" : "893-324-1064",
"confNo" : "LE6J4Z",
 [ {
    "id" : "79039899165297",
    "tagNum" : "17657806255240",
    "routing" : "MIA/LAX/MEL",
    "lastActionCode" : "OFFLOAD",
    "lastActionDesc" : "OFFLOAD",
    "lastSeenStation" : "MEL",
    "flightLegs" : [ {
      "flightNo" : "BM604",
      "flightDate" : "2019-02-01T01:00:00",
      "fltRouteSrc" : "MIA",
      "fltRouteDest" : "LAX",
      "estimatedArrival" : "2019-02-01T03:00:00",
      "actions" : [ {
        "actionAt" : "MIA",
        "actionCode" : "ONLOAD to LAX",
        "actionTime" : "2019-02-01T01:13:00"
      }, {
        "actionAt" : "MIA",
        "actionCode" : "BagTag Scan at MIA",
        "actionTime" : "2019-02-01T00:47:00"
      }, {
        "actionAt" : "MIA",
        "actionCode" : "Checkin at MIA",
        "actionTime" : "2019-02-01T23:38:00"
      } ]
    }, {
      "flightNo" : "BM667",
      "flightDate" : "2019-01-31T22:13:00",
      "fltRouteSrc" : "LAX",
      "fltRouteDest" : "MEL",
      "estimatedArrival" : "2019-02-02T03:15:00",
      "actions" : [ {
        "actionAt" : "MEL",
        "actionCode" : "Offload to Carousel at MEL",
        "actionTime" : "2019-02-02T03:15:00"
      }, {
        "actionAt" : "LAX",
        "actionCode" : "ONLOAD to MEL",
        "actionTime" : "2019-02-01T07:35:00"
      }, {
        "actionAt" : "LAX",
        "actionCode" : "OFFLOAD from LAX",
        "actionTime" : "2019-02-01T07:18:00"
```

```
          } ]
        } ],
        "lastSeenTimeGmt" : "2019-02-02T03:13:00",
        "bagArrivalDate" : "2019.02.02T03:13:00"
      } ]
```

Example 2: Where was the baggage with a given reservation number **FH7G1W** last seen? Also, fetch the tag number of the baggage.

```
SELECT bag.fullName, bag.bagInfo[].tagNum,bag.bagInfo[].lastSeenStation
FROM BaggageInfo bag WHERE bag.confNo="FH7G1W"
```

Explanation: The `bagInfo` is JSON and is populated with an array of documents. The full name and the last seen station can be fetched for a particular reservation number.

Output:

```
{"fullName":"Rosalia Triplett","tagNum":"17657806215913",
"lastSeenStation":"VIE"}
```

Example 3: Select details of the bags(tag and last seen time) for a passenger with ticket number **1762340579411**.

```
SELECT bag.ticketNo, bag.fullName,
bag.bagInfo[].tagNum,bag.bagInfo[].lastSeenStation
FROM BaggageInfo bag where bag.ticketNo=1762320369957
```

Explanation: The `bagInfo` is JSON and is populated with an array of documents. The full name, tag number, and last seen station can be fetched for a particular ticket number.

Output:

```
{"fullName":"Lorenzo Phil","tagNum":
["17657806240001","17657806340001"],
"lastSeenStation":["JTR","JTR"]}
```

Using Path expressions

Path expressions are used to navigate inside hierarchically structured data. Oracle NoSQL Database supports different complex data types like arrays and records. You will learn how to work with different complex data types using path expressions.

If you want to follow along with the examples, download the script baggageschema_loaddata.sql and execute it as shown below. This script creates the table used in the example and loads data into the table.

Start your KVSTORE or KVLite and open the SQL.shell.

```
java -jar lib/kvstore.jar kvlite -secure-config disable
java -jar lib/sql.jar -helper-hosts localhost:5000 -store kvstore
```

Using the `load` command, run the script.

```
load -file baggageschema_loaddata.sql
```


Using Internal variables and aliases

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

The table name in a query may be followed by a table alias. Table aliases are essentially variables ranging over the rows of the specified table. If no alias is specified, one is created internally, using the name of the table as it is spelled in the query.

Example 1: Find the ticket number and passenger details for a given reservation code:

```
SELECT bagDet.ticketNo, bagDet.fullName, bagDet.contactPhone FROM
BaggageInfo bagDet
WHERE confNo="QB1O0J"
```

Explanation: In this query, you fetch the values of static fields like fullname, ticket number, and contact phone for a particular reservation code. You use a table alias for the `BaggageInfo` table.

Output:

```
{"ticketNo":1762390789239,"fullName":"Zina
      Christenson","contactPhone":"987-210-3029"}
```

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.

Example 2: Fetch the full name and tag number for all customer baggage shipped after 2019.

```
SELECT fullName, bag.ticketNo FROM BaggageInfo bag WHERE
exists bag.bagInfo[$element.bagArrivalDate >="2019-01-01T00:00:00"]
```

Explanation: The bag arrival date value for every bag should be greater than the year 2019. Here the "$element" is bound to the context row (every baggage of the customer). The EXISTS operator checks whether the sequence returned by its input expression is empty or

not. The sequence returned by the comparison operator ">=" is non-empty for all bags which arrived after 2019.

Output:

```
{"fullName":"Lucinda Beckman","ticketNo":1762320569757}
{"fullName":"Adelaide Willard","ticketNo":1762392135540}
{"fullName":"Raymond Griffin","ticketNo":1762399766476}
{"fullName":"Elane Lemons","ticketNo":1762324912391}
{"fullName":"Zina Christenson","ticketNo":1762390789239}
{"fullName":"Zulema Martindale","ticketNo":1762340579411}
{"fullName":"Dierdre Amador","ticketNo":1762376407826}
{"fullName":"Henry Jenkins","ticketNo":176234463813}
{"fullName":"Rosalia Triplett","ticketNo":1762311547917}
{"fullName":"Lorenzo Phil","ticketNo":1762320369957}
{"fullName":"Gerard Greene","ticketNo":1762341772625}
{"fullName":"Adam Phillips","ticketNo":1762344493810}
{"fullName":"Doris Martin","ticketNo":1762355527825}
{"fullName":"Joanne Diaz","ticketNo":1762383911861}
{"fullName":"Omar Harvey","ticketNo":1762348904343}
{"fullName":"Fallon Clements","ticketNo":1762350390409}
{"fullName":"Lisbeth Wampler","ticketNo":1762355854464}
{"fullName":"Teena Colley","ticketNo":1762357254392}
{"fullName":"Michelle Payne","ticketNo":1762330498104}
{"fullName":"Mary Watson","ticketNo":1762340683564}
{"fullName":"Kendal Biddle","ticketNo":1762377974281}
```

Working with Arrays

An array is an ordered collection of zero or more items. The items of an array are called elements. Arrays cannot contain any NULL values.

The `BaggageInfo` schema has many arrays. A simple array from the schema is the `actions` array in every `flightLeg`. You can use path expressions to navigate a simple array or a nested array.

```
"actions" : [ {
"actionAt" : "SYD",
"actionCode" : "ONLOAD to SIN",
"actionTime" : "2019.02.28 at 22:09:00 AEDT"
}, {
"actionAt" : "SYD",
"actionCode" : "BagTag Scan at SYD",
"actionTime" : "2019.02.28 at 21:51:00 AEDT"
}, {
"actionAt" : "SYD",
"actionCode" : "Checkin at SYD",
"actionTime" : "2019.02.28 at 20:06:00 AEDT"
} ]
```

Example 1: Fetch the details of the first leg of every bag (including all the actions taken at the leg) for the passenger with ticket number **1762357254392**.

```
SELECT bagDet.fullName, bagDet.bagInfo[].flightLegs[0]
AS Details FROM BaggageInfo bagDet WHERE ticketNo=1762357254392
```

In the above query, `flightLegs` is an array. The slice step `[0]` is applied to the `flightLegs` array. Since array elements start with 0, this gives you the first record in the array. You get the first leg information of every bag for each passenger. You apply an additional filter with the `ticketNo` and so only one passenger information is fetched.

Output:

```
{"fullName":"Teena Colley",
"Details":[[
{"actionAt":"MSQ","actionCode":"ONLOAD to
FRA","actionTime":"2019-02-13T07:17:00Z"},
{"actionAt":"MSQ","actionCode":"BagTag Scan at
MSQ","actionTime":"2019-02-13T06:52:00Z"},
{"actionAt":"MSQ","actionCode":"Checkin at
MSQ","actionTime":"2019-02-13T06:11:00Z"}],
"2019-02-13T09:00:00Z","2019-02-13T07:00:00Z","BM365","FRA","MSQ"]}
```

> **Note:**
>
> You can also use a slice step to select all array elements whose positions are within a range: [low: high], where low and high are expressions to specify the range boundaries. You can omit low and high expressions if you do not require a low or high boundary.

Example: Fetch the details of all the legs (including all the actions taken at all the legs) for the passenger with ticket number **1762357254392**.

You'll be using the slice step to fetch the first 3 records of the `flightLegs` array.

```
SELECT bagDet.fullName, bagDet.bagInfo[].flightLegs[0:2] AS Details
FROM BaggageInfo bagDet WHERE ticketNo=1762357254392
```

Output:

```
{"fullName":"Teena Colley",
"Details":[
 [
   {"actionAt":"MSQ","actionCode":"ONLOAD to
FRA","actionTime":"2019-02-13T07:17:00Z"},
   {"actionAt":"MSQ","actionCode":"BagTag Scan at
MSQ","actionTime":"2019-02-13T06:52:00Z"},
   {"actionAt":"MSQ","actionCode":"Checkin at
MSQ","actionTime":"2019-02-13T06:11:00Z"}
 ],
 "2019-02-13T09:00:00Z","2019-02-13T07:00:00Z","BM365","FRA","MSQ",
```

```
 [
   {"actionAt":"HKG","actionCode":"Offload to Carousel at
HKG","actionTime":"2019-02-13T11:15:00Z"},
   {"actionAt":"FRA","actionCode":"ONLOAD to
HKG","actionTime":"2019-02-13T10:39:00Z"},
   {"actionAt":"FRA","actionCode":"OFFLOAD from
FRA","actionTime":"2019-02-13T10:37:00Z"}
 ],
 "2019-02-13T11:18:00Z","2019-02-13T07:17:00Z","BM313","HKG","FRA"
]}
```

Working with nested data type

Oracle NoSQL database supports nested data type. That means you can have one data type inside another data type. For example, records inside an array, an array inside an array, and so on. The sample `Baggageinfo` schema uses nested data type of an array of arrays.

Example 1: Fetch the various actions taken on the first leg for the passenger with the ticket number **1762330498104**.

```
SELECT bagDet.fullName,
bagDet.bagInfo[].flightLegs[0].values().values() AS Action
FROM BaggageInfo bagDet WHERE ticketNo=1762330498104
```

Explanation: In the above query, `flightLegs` is a nested data type. This in turn has an `actions` array, which is an array of records. The above query is executed in two steps.

1. `$bag.bagInfo[].flightLegs[0].values()` gives all the entries in the first record of the`flightLegs` array. This includes an `actions` array. You can iterate this (using `values()`) to get all the records of the `actions` array as shown below.
2. `$bag.bagInfo[].flightLegs[0].values().values()` gives all the records of the `actions` array.

Output:

```
{"fullName":"Michelle Payne",
"Action":["SFO","ONLOAD to IST","2019-02-02T12:10:00Z","SFO",
"BagTag Scan at SFO","2019-02-02T11:47:00Z","SFO",
"Checkin at SFO","2019-02-02T10:01:00Z"]}
```

Example 2: Display details of the last transit action update done on the first leg for the passenger with the ticket number **1762340683564**.

```
SELECT bagDet.fullName, (bagDet.bagInfo[].flightLegs[0].values())
[2].actionCode
AS lastTransit_Update FROM BaggageInfo bagDet WHERE
ticketNo=1762340683564
```

Explanation: The above query is processed using the following steps:

1. `$bagDet.bagInfo[].flightLegs[0].values()` gives all the entries in the first record of the `flightLegs` array.
2. `bagInfo[].flightLegs[0].values())[2]` points to the third (which is the last) record of the `actions` array inside the first element of the `flightLegs` array.
3. There are multiple records in the actions array. `bagInfo[].flightLegs[0].values())[2].actionCode` fetches the value corresponding to the `actionCode` element.

Output:

```
{"fullName":"Mary Watson","lastTransit_Update":"Checkin at YYZ"}
```

> **Note:**
>
> In a later section you will learn to write the same query in a generic way without hardcoding the array index by using the size function. See Finding the size of a complex data type.

Finding the size of a complex data type

The size function can be used to return the size (number of fields/entries) of a complex data type.

Example 1: Find out how many flight legs/hops are there for a passenger with ticket number **1762320569757**.

```
SELECT bagDet.fullName, size(bagDet.bagInfo.flightLegs) as Noof_Legs
FROM BaggageInfo bagDet WHERE ticketNo=1762320569757
```

Explanation: In the above query, you get the size of the `flightLegs` array using the `size` function.

Output:

```
{"fullName":"Lucinda Beckman","Noof_Legs":3}
```

Example 2: Find the number of action entries (for the bags) in the first leg for the passenger with ticket number **1762357254392**.

```
SELECT bagDet.fullName, size(bagDet.bagInfo[].flightLegs[0].actions) AS
FirstLeg_NoofActions
FROM BaggageInfo bagDet WHERE ticketNo=1762357254392
```

Output:

```
{"fullName":"Teena Colley","FirstLeg_NoofActions":3}
```

Example 3: Display details of the last transit action update done on the first leg for the passenger with the ticket number **1762340683564**.

```
SELECT bagDet.fullName,
(bagDet.bagInfo[].flightLegs[0].values())
[size(bagDet.bagInfo.flightLegs[0].actions)-1].actionCode
AS lastTransit_Update FROM BaggageInfo bagDet WHERE
ticketNo=1762340683564
```

Output:

```
{"fullName":"Mary Watson","lastTransit_Update":"Checkin at YYZ"}
```

Explanation:

The above query is processed using the following steps:

- 1. `$bagDet.bagInfo[].flightLegs[0].values()` gives all the entries in the first record of the `flightLegs` array.
  2. `size(bagDet.bagInfo.flightLegs[0].actions)` gives the size of the actions array in the first leg.
  3. There are multiple records in the actions array. You can use the result of the size function to get the last record in the action array and the corresponding `actionCode` can be fetched. You subtract the size by 1 as the index of an array starts with 0.

> **Note:**
>
> The same query has been written in the topic Working with nested data type by hard coding the index of the actions array. Using the `size` function, you have rewritten the same query in a generic way without hard coding the index.

Using Left Outer joins with parent-child tables

A JOIN is used to combine rows from two or more tables, based on a related column between them. In a hierarchical table, the child table inherits the primary key columns of its parent table. This is done implicitly, without including the parent columns in the `CREATE TABLE` statement of the child. All tables in the hierarchy have the same shard key columns.

A Left Outer Join (LOJ) is one of the join operations that allows you to specify a join clause.

- Overview of Left Outer Joins
- Examples using Left Outer Joins

Overview of Left Outer Joins

A Left Outer Join (LOJ) is one of the join operations that allows you to specify a join clause. It preserves the unmatched rows from the first (left) table, joining them with a NULL row in the second (right) table. This means all left rows that do not have a matching row in the right table will appear in the result, paired with a NULL value in place of a right row.

In an LOJ, the order of fields in the result-set is always in top-down order. That means the order of output in the result set is always from the ancestor table first and then the descendant table. This is true irrespective of the order of the joins.

Characteristics of LEFT OUTER JOIN:

- Queries multiple tables in the same hierarchy
- It is an ANSI-SQL Standard
- It does not support sibling table joins

If you want to follow along with the examples, download the script parentchildtbls_loaddata.sql and run it as shown below. This script creates the table used in the example and loads data into the table.

Start your KVSTORE or KVLite and open the SQL.shell.

```
java -jar lib/kvstore.jar kvlite -secure-config disable
java -jar lib/sql.jar -helper-hosts localhost:5000 -store kvstore
```

Using the `load` command, execute the script.

```
load -file parentchildtbls_loaddata.sql
```

Examples using Left Outer Joins

Various tables used in the examples :

- **`ticket`**

```
ticketNo LONG
confNo STRING
PRIMARY KEY(ticketNo)
```

- **`ticket.bagInfo`**

```
id LONG
tagNum LONG
routing STRING
lastActionCode STRING
lastActionDesc STRING
lastSeenStation STRING,
lastSeenTimeGmt TIMESTAMP(4)
bagArrivalDate TIMESTAMP(4)
PRIMARY KEY(id)
```

- **ticket.bagInfo.flightLegs**

```
flightNo STRING
flightDate TIMESTAMP(4)
fltRouteSrc STRING
fltRouteDest STRING
estimatedArrival TIMESTAMP(4),
actions JSON
PRIMARY KEY(flightNo)
```

- **ticket.passengerInfo**

```
contactPhone STRING
fullName STRING
gender STRING
PRIMARY KEY(contactPhone)
```

Example 1: Fetch the details of all passengers who have been issued a ticket.

```
SELECT fullname, contactPhone,gender
FROM ticket a
LEFT OUTER JOIN ticket.passengerInfo b
ON a.ticketNo=b.ticketNo
```

Explanation: This is an example of a join where the target table `ticket` is joined with its child table `passengerInfo`.

Output:

```
{"fullname":"Elane Lemons","contactPhone":"600-918-8404","gender":"F"}
{"fullname":"Adelaide
Willard","contactPhone":"421-272-8082","gender":"M"}
{"fullname":"Dierdre
Amador","contactPhone":"165-742-5715","gender":"M"}
{"fullname":"Doris Martin","contactPhone":"289-564-3497","gender":"F"}
{"fullname":"Adam Phillips","contactPhone":"893-324-1064","gender":"M"}
```

Example 1a: Fetch the details of the passenger with ticket number **1762324912391** .

```
SELECT fullname, contactPhone, gender
FROM ticket a
LEFT OUTER JOIN ticket.passengerInfo b
ON a.ticketNo=b.ticketNo
WHERE a.ticketNo=1762324912391
```

Explanation: This is an example of a join where the target table `ticket` is joined with its child table `passengerInfo` and a filter is applied to restrict the result. In this example, the result set is limited by applying a filter condition to the result of the join. You are limiting the result to a particular ticket number.

Output:


```
{"fullname":"Elane Lemons","contactPhone":"600-918-8404","gender":"F"}
```


Example 2: Fetch all the bag details for all passengers who have been issued a ticket.

```
SELECT * FROM ticket a
LEFT OUTER JOIN ticket.bagInfo b
ON a.ticketNo=b.ticketNo
```

Explanation: This is an example of a join where the target table`ticket` is joined with its child table `bagInfo`.

Output:


```
{"a":{"ticketNo":1762344493810,"confNo":"LE6J4Z"},
"b":
{"ticketNo":1762344493810,"id":79039899165297,"tagNum":17657806255240,"routin
g":"MIA/LAX/MEL",
"lastActionCode":"OFFLOAD","lastActionDesc":"OFFLOAD","lastSeenStation":"MEL"
,
"lastSeenTimeGmt":"2019-02-01T16:13:00.0000Z","bagArrivalDate":"2019-02-01T16
:13:00.0000Z"}}

{"a":{"ticketNo":1762324912391,"confNo":"LN0C8R"},
"b":
{"ticketNo":1762324912391,"id":79039899168383,"tagNum":1765780623244,"routing
":"MXP/CDG/SLC/BZN",
"lastActionCode":"OFFLOAD","lastActionDesc":"OFFLOAD","lastSeenStation":"BZN"
,
"lastSeenTimeGmt":"2019-03-15T10:13:00.0000Z","bagArrivalDate":"2019-03-15T10
:13:00.0000Z"}}

{"a":{"ticketNo":1762392135540,"confNo":"DN3I4Q"},
"b":
{"ticketNo":1762392135540,"id":79039899156435,"tagNum":17657806224224,"routin
g":"GRU/ORD/SEA",
"lastActionCode":"OFFLOAD","lastActionDesc":"OFFLOAD","lastSeenStation":"SEA"
,
"lastSeenTimeGmt":"2019-02-15T21:21:00.0000Z","bagArrivalDate":"2019-02-15T21
:21:00.0000Z"}}

{"a":{"ticketNo":1762376407826,"confNo":"ZG8Z5N"},
"b":
{"ticketNo":1762376407826,"id":7903989918469,"tagNum":17657806240229,"routing
":"JFK/MAD",
"lastActionCode":"OFFLOAD","lastActionDesc":"OFFLOAD","lastSeenStation":"MAD"
,
"lastSeenTimeGmt":"2019-03-07T13:51:00.0000Z","bagArrivalDate":"2019-03-07T13
:51:00.0000Z"}}

{"a":{"ticketNo":1762355527825,"confNo":"HJ4J4P"},
"b":
{"ticketNo":1762355527825,"id":79039899197492,"tagNum":17657806232501,"routin
```

g":"BZN/SEA/CDG/MXP",
"lastActionCode":"OFFLOAD","lastActionDesc":"OFFLOAD","lastSeenStation"
:"MXP",
"lastSeenTimeGmt":"2019-03-22T10:17:00.0000Z","bagArrivalDate":"2019-03
-22T10:17:00.0000Z"}}

Example 2a: Fetch all the bag details for a particular ticket number.

```
SELECT * FROM ticket a
LEFT OUTER JOIN ticket.bagInfo b
ON a.ticketNo=b.ticketNo
WHERE a.ticketNo=1762324912391
```

This is an example of a join where the target table `ticket` is joined with its child table `bagInfo` and a filter is applied to restrict the result. In this example, the result set is limited by applying a filter condition to the result of the join. You are limiting the result to a particular ticket number.

Output:

{"a":{"ticketNo":1762324912391,"confNo":"LN0C8R"},
"b":
{"ticketNo":1762324912391,"id":79039899168383,"tagNum":1765780623244,"r
outing":"MXP/CDG/SLC/BZN",
"lastActionCode":"OFFLOAD","lastActionDesc":"OFFLOAD","lastSeenStation"
:"BZN",
"lastSeenTimeGmt":"2019-03-15T10:13:00.0000Z","bagArrivalDate":"2019-03
-15T10:13:00.0000Z"}}

> **Note:**
>
> If you move the non-join predicate restriction to the ON clause, the result set includes all the rows that meet the ON clause condition. Rows from the right outer table that do not meet the ON condition are populated with NULL values as shown below.
>
> ```
> SELECT * FROM ticket a
> LEFT OUTER JOIN ticket.bagInfo b
> ON a.ticketNo=b.ticketNo AND
> a.ticketNo=1762324912391
> ```

Output:

{"a":{"ticketNo":1762355527825,"confNo":"HJ4J4P"},"b":null}
{"a":{"ticketNo":1762344493810,"confNo":"LE6J4Z"},"b":null}
{"a":{"ticketNo":1762324912391,"confNo":"LN0C8R"}, "b":
{"ticketNo":1762324912391,"id":79039899168383,"tagNum":1765780623244,"r
outing":"MXP/CDG/SLC/BZN",
"lastActionCode":"OFFLOAD","lastActionDesc":"OFFLOAD","lastSeenStation"
:"BZN",

"lastSeenTimeGmt":"2019-03-15T10:13:00.0000Z","bagArrivalDate":"2019-03-15T10
:13:00.0000Z"}}
{"a":{"ticketNo":1762392135540,"confNo":"DN3I4Q"},"b":null}
{"a":{"ticketNo":1762376407826,"confNo":"ZG8Z5N"},"b":null}

Example 3: Fetch all flight legs details for all passengers.

```
SELECT *FROM ticket a
LEFT OUTER JOIN ticket.bagInfo.flightLegs b
ON a.ticketNo=b.ticketNo;
```

Explanation: This is an example of a join where the target table `ticket` is joined with its descendant `ticketInfo`. A descendant table can be any level hierarchically below a table (For example `fightLegs` is the child of `bagInfo` which is the child of `ticket`, so `fightLegs` is a descendant of `ticket`).

Output:

{"a":{"ticketNo":1762344493810,"confNo":"LE6J4Z"},
"b":
{"ticketNo":1762344493810,"id":79039899165297,"flightNo":"BM604","flightDate"
:"2019-02-01T06:00:00.0000Z",
"fltRouteSrc":"MIA","fltRouteDest":"LAX","estimatedArrival":"2019-02-01T11:00
:00.0000Z",
"actions":[{"actionAt":"MIA","actionCode":"ONLOAD to
LAX","actionTime":"2019-02-01T06:13:00Z"},
{"actionAt":"MIA","actionCode":"BagTag Scan at
MIA","actionTime":"2019-02-01T05:47:00Z"},
{"actionAt":"MIA","actionCode":"Checkin at
MIA","actionTime":"2019-02-01T04:38:00Z"}]}}

{"a":{"ticketNo":1762344493810,"confNo":"LE6J4Z"},
"b":
{"ticketNo":1762344493810,"id":79039899165297,"flightNo":"BM667","flightDate"
:"2019-02-01T06:13:00.0000Z",
"fltRouteSrc":"LAX","fltRouteDest":"MEL","estimatedArrival":"2019-02-01T16:15
:00.0000Z",
"actions":[{"actionAt":"MEL","actionCode":"Offload to Carousel at
MEL","actionTime":"2019-02-01T16:15:00Z"},
{"actionAt":"LAX","actionCode":"ONLOAD to
MEL","actionTime":"2019-02-01T15:35:00Z"},
{"actionAt":"LAX","actionCode":"OFFLOAD from
LAX","actionTime":"2019-02-01T15:18:00Z"}]}}

{"a":{"ticketNo":1762324912391,"confNo":"LN0C8R"},
"b":
{"ticketNo":1762324912391,"id":79039899168383,"flightNo":"BM170","flightDate"
:"2019-03-15T08:13:00.0000Z",
"fltRouteSrc":"SLC","fltRouteDest":"BZN","estimatedArrival":"2019-03-15T10:14
:00.0000Z",
"actions":[{"actionAt":"BZN","actionCode":"Offload to Carousel at
BZN","actionTime":"2019-03-15T10:13:00Z"},
{"actionAt":"SLC","actionCode":"ONLOAD to
BZN","actionTime":"2019-03-15T10:06:00Z"},

```
{"actionAt":"SLC","actionCode":"OFFLOAD from
SLC","actionTime":"2019-03-15T09:59:00Z"}]}}

{"a":{"ticketNo":1762324912391,"confNo":"LN0C8R"},
"b":
{"ticketNo":1762324912391,"id":79039899168383,"flightNo":"BM490","fligh
tDate":"2019-03-15T08:13:00.0000Z",
"fltRouteSrc":"CDG","fltRouteDest":"SLC","estimatedArrival":"2019-03-15
T10:14:00.0000Z",
"actions":[{"actionAt":"CDG","actionCode":"ONLOAD to
SLC","actionTime":"2019-03-15T09:42:00Z"},
{"actionAt":"CDG","actionCode":"BagTag Scan at
CDG","actionTime":"2019-03-15T09:17:00Z"},
{"actionAt":"CDG","actionCode":"OFFLOAD from
CDG","actionTime":"2019-03-15T09:19:00Z"}]}}

{"a":{"ticketNo":1762324912391,"confNo":"LN0C8R"},
"b":
{"ticketNo":1762324912391,"id":79039899168383,"flightNo":"BM936","fligh
tDate":"2019-03-15T08:00:00.0000Z",
"fltRouteSrc":"MXP","fltRouteDest":"CDG","estimatedArrival":"2019-03-15
T09:00:00.0000Z",
"actions":[{"actionAt":"MXP","actionCode":"ONLOAD to
CDG","actionTime":"2019-03-15T08:13:00Z"},
{"actionAt":"MXP","actionCode":"BagTag Scan at
MXP","actionTime":"2019-03-15T07:48:00Z"},
{"actionAt":"MXP","actionCode":"Checkin at
MXP","actionTime":"2019-03-15T07:38:00Z"}]}}

{"a":{"ticketNo":1762392135540,"confNo":"DN3I4Q"},
"b":
{"ticketNo":1762392135540,"id":79039899156435,"flightNo":"BM79","flight
Date":"2019-02-15T01:00:00.0000Z",
"fltRouteSrc":"GRU","fltRouteDest":"ORD","estimatedArrival":"2019-02-15
T11:00:00.0000Z",
"actions":[{"actionAt":"GRU","actionCode":"ONLOAD to
ORD","actionTime":"2019-02-15T01:21:00Z"},
{"actionAt":"GRU","actionCode":"BagTag Scan at
GRU","actionTime":"2019-02-15T00:55:00Z"},
{"actionAt":"GRU","actionCode":"Checkin at
GRU","actionTime":"2019-02-14T23:49:00Z"}]}}

{"a":{"ticketNo":1762392135540,"confNo":"DN3I4Q"}
,"b":
{"ticketNo":1762392135540,"id":79039899156435,"flightNo":"BM907","fligh
tDate":"2019-02-15T01:21:00.0000Z",
"fltRouteSrc":"ORD","fltRouteDest":"SEA","estimatedArrival":"2019-02-15
T21:22:00.0000Z",
"actions":[{"actionAt":"SEA","actionCode":"Offload to Carousel at
SEA","actionTime":"2019-02-15T21:16:00Z"},
{"actionAt":"ORD","actionCode":"ONLOAD to
SEA","actionTime":"2019-02-15T20:52:00Z"},
{"actionAt":"ORD","actionCode":"OFFLOAD from
ORD","actionTime":"2019-02-15T20:44:00Z"}]}}
```

{"a":{"ticketNo":1762376407826,"confNo":"ZG8Z5N"},
"b":
{"ticketNo":1762376407826,"id":7903989918469,"flightNo":"BM495","flightDate":
"2019-03-07T07:00:00.0000Z",
"fltRouteSrc":"JFK","fltRouteDest":"MAD","estimatedArrival":"2019-03-07T14:00
:00.0000Z",
"actions":[{"actionAt":"MAD","actionCode":"Offload to Carousel at
MAD","actionTime":"2019-03-07T13:54:00Z"},
{"actionAt":"JFK","actionCode":"ONLOAD to
MAD","actionTime":"2019-03-07T07:00:00Z"},
{"actionAt":"JFK","actionCode":"BagTag Scan at
JFK","actionTime":"2019-03-07T06:53:00Z"},
{"actionAt":"JFK","actionCode":"Checkin at
JFK","actionTime":"2019-03-07T05:03:00Z"}]}}

{"a":{"ticketNo":1762355527825,"confNo":"HJ4J4P"},
"b":
{"ticketNo":1762355527825,"id":79039899197492,"flightNo":"BM386","flightDate"
:"2019-03-22T07:23:00.0000Z",
"fltRouteSrc":"CDG","fltRouteDest":"MXP","estimatedArrival":"2019-03-22T10:24
:00.0000Z",
"actions":[{"actionAt":"MXP","actionCode":"Offload to Carousel at
MXP","actionTime":"2019-03-22T10:15:00Z"},
{"actionAt":"CDG","actionCode":"ONLOAD to
MXP","actionTime":"2019-03-22T10:09:00Z"},
{"actionAt":"CDG","actionCode":"OFFLOAD from
CDG","actionTime":"2019-03-22T10:01:00Z"}]}}

{"a":{"ticketNo":1762355527825,"confNo":"HJ4J4P"},
"b":
{"ticketNo":1762355527825,"id":79039899197492,"flightNo":"BM578","flightDate"
:"2019-03-22T07:23:00.0000Z",
"fltRouteSrc":"SEA","fltRouteDest":"CDG","estimatedArrival":"2019-03-21T23:24
:00.0000Z",
"actions":[{"actionAt":"SEA","actionCode":"ONLOAD to
CDG","actionTime":"2019-03-22T11:26:00Z"},
{"actionAt":"SEA","actionCode":"BagTag Scan at
SEA","actionTime":"2019-03-22T10:57:00Z"},
{"actionAt":"SEA","actionCode":"OFFLOAD from
SEA","actionTime":"2019-03-22T11:07:00Z"}]}}

{"a":{"ticketNo":1762355527825,"confNo":"HJ4J4P"},
"b":
{"ticketNo":1762355527825,"id":79039899197492,"flightNo":"BM704","flightDate"
:"2019-03-22T07:00:00.0000Z",
"fltRouteSrc":"BZN","fltRouteDest":"SEA","estimatedArrival":"2019-03-22T09:00
:00.0000Z",
"actions":[{"actionAt":"BZN","actionCode":"ONLOAD to
SEA","actionTime":"2019-03-22T07:23:00Z"},
{"actionAt":"BZN","actionCode":"BagTag Scan at
BZN","actionTime":"2019-03-22T06:58:00Z"},
{"actionAt":"BZN","actionCode":"Checkin at
BZN","actionTime":"2019-03-22T05:20:00Z"}]}}

Example 3a: Fetch all the flight leg details for a particular ticket number.

```
SELECT * FROM ticket a
LEFT OUTER JOIN ticket.bagInfo.flightLegs b
ON a.ticketNo=b.ticketNo
WHERE a.ticketNo=1762344493810
```

This is an example of a join where the target table `ticket` is joined with its descendant `bagInfo` and a filter is applied to restrict the result. In this example, the result set is limited by applying a filter condition to the result of the join. You are limiting the result to a particular ticket number.

The result has two rows, implying there are two flight legs for this ticket number.

Output:

```
"a":{"ticketNo":1762344493810,"confNo":"LE6J4Z"},
"b":{"ticketNo":1762344493810,"id":79039899165297,"flightNo":"BM604",
"flightDate":"2019-02-01T06:00:00.0000Z","fltRouteSrc":"MIA","fltRouteD
est":"LAX",
"estimatedArrival":"2019-02-01T11:00:00.0000Z",
"actions":[{"actionAt":"MIA","actionCode":"ONLOAD to
LAX","actionTime":"2019-02-01T06:13:00Z"},
{"actionAt":"MIA","actionCode":"BagTag Scan at
MIA","actionTime":"2019-02-01T05:47:00Z"},
{"actionAt":"MIA","actionCode":"Checkin at
MIA","actionTime":"2019-02-01T04:38:00Z"}]}}

{"a":{"ticketNo":1762344493810,"confNo":"LE6J4Z"},
"b":{"ticketNo":1762344493810,"id":79039899165297,"flightNo":"BM667",
"flightDate":"2019-02-01T06:13:00.0000Z","fltRouteSrc":"LAX","fltRouteD
est":"MEL",
"estimatedArrival":"2019-02-01T16:15:00.0000Z",
"actions":[{"actionAt":"MEL","actionCode":"Offload to Carousel at
MEL","actionTime":"2019-02-01T16:15:00Z"},
{"actionAt":"LAX","actionCode":"ONLOAD to
MEL","actionTime":"2019-02-01T15:35:00Z"},
{"actionAt":"LAX","actionCode":"OFFLOAD from
LAX","actionTime":"2019-02-01T15:18:00Z"}]}}
```

Example 4: Fetch the bag id and number of hops for all bags of all passengers.

```
SELECT b.id,count(*) AS NUMBER_HOPS
FROM ticket a LEFT OUTER JOIN ticket.bagInfo.flightLegs b
ON a.ticketNo=b.ticketNo GROUP BY b.id
```

Explanation: You group the data based on the bag id (using GROUP BY) and get the count of flight legs (using count()) for every bag.

Output:

```
{"id":79039899168383,"NUMBER_HOPS":3}
{"id":79039899156435,"NUMBER_HOPS":2}
{"id":7903989918469,"NUMBER_HOPS":1}
```

```
{"id":79039899165297,"NUMBER_HOPS":2}
{"id":79039899197492,"NUMBER_HOPS":3}
```

Example 4a: Find the number of hops for all the bags of a given passenger.

```
SELECT b.id,count(*) AS NUMBER_HOPS
FROM ticket a LEFT OUTER JOIN ticket.bagInfo.flightLegs b
ON a.ticketNo=b.ticketNo
WHERE a.ticketNo=1762355527825 GROUP BY b.id
```

Explanation: You group the data based on the bag id (using GROUP BY) and get the count of flight legs (Using count())for every bag. Additionally, you filter the results for a particular ticket number.

Output:

```
{"id":79039899197492,"NUMBER_HOPS":3}
```

Example 5: Fetch bag id and routing details of all bags that arrived after 2019.

```
SELECT b.id, routing
FROM ticket a LEFT OUTER JOIN ticket.bagInfo b
ON a.ticketNo=b.ticketNo
WHERE CAST (b.bagArrivalDate AS Timestamp(0))
>= CAST ("2019-01-01T00:00:00" AS Timestamp(0))
```

Explanation: This is an example of a join where the target table`ticket` is joined with its child table `bagInfo`. The filter condition is applied on the `bagArrivalDate`. The `CAST` function is used to convert the string into Timestamp and then the values are compared.

Output:

```
{"id":79039899197492,"routing":"BZN/SEA/CDG/MXP"}
{"id":79039899165297,"routing":"MIA/LAX/MEL"}
{"id":79039899168383,"routing":"MXP/CDG/SLC/BZN"}
{"id":79039899156435,"routing":"GRU/ORD/SEA"}
{"id":7903989918469,"routing":"JFK/MAD"}
```

Using NESTED TABLES to join parent-child tables

A JOIN is used to combine rows from two or more tables, based on a related column between them. In a hierarchical table, the child table inherits the primary key columns of its parent table. This is done implicitly, without including the parent columns in the `CREATE TABLE` statement of the child. All tables in the hierarchy have the same shard key columns.

You can use NESTED TABLES clause to join tables in Oracle NoSQL Database.

- Overview of NESTED TABLES
- Examples using NESTED TABLES

Overview of NESTED TABLES

The NESTED TABLES clause specifies the participating tables and separates them into 3 groups. First, the target table from where the data is fetched is specified. Then the ANCESTORS clause, if present, specifies the number of tables that must be ancestors of the target table in the table hierarchy. Finally, the DESCENDANTS clause, if present, specifies the number of tables that must be descendants of the target table in the table hierarchy.

> **Note:**
>
> Semantically, a NESTED TABLES clause is equivalent to a number of left-outer-join operations "centered" around the target table.

Characteristics of NESTED tables:

- Queries multiple tables in the same hierarchy
- It is not an ANSI-SQL Standard
- It supports sibling tables join

Table 4-1 Nested Tables Vs LOJ

Nested Tables	LOJ
Queries multiple tables in the same hierarchy	Queries multiple tables in the same hierarchy
Not an ANSI-SQL Standard	ANSI-SQL Standard
Supports sibling tables join	Does not support sibling table joins

If you want to follow along with the examples, download the script parentchildtbls_loaddata.sql and run it as shown below. This script creates the table used in the example and loads data into the table.

Start your KVSTORE or KVLite and open the SQL.shell.

```
java -jar lib/kvstore.jar kvlite -secure-config disable
java -jar lib/sql.jar -helper-hosts localhost:5000 -store kvstore
```

Using the `load` command, execute the script.

```
load -file parentchildtbls_loaddata.sql
```

Examples using NESTED TABLES

Various tables used in the examples :

- **ticket**

```
ticketNo LONG
confNo STRING
PRIMARY KEY(ticketNo)
```

- **ticket.bagInfo**

```
id LONG
tagNum LONG
routing STRING
lastActionCode STRING
lastActionDesc STRING
lastSeenStation STRING,
lastSeenTimeGmt TIMESTAMP(4)
bagArrivalDate TIMESTAMP(4)
PRIMARY KEY(id)
```

- **ticket.bagInfo.flightLegs**

```
flightNo STRING
flightDate TIMESTAMP(4)
fltRouteSrc STRING
fltRouteDest STRING
estimatedArrival TIMESTAMP(4),
actions JSON
PRIMARY KEY(flightNo)
```

- **ticket.passengerInfo**

```
contactPhone STRING
fullName STRING
gender STRING
PRIMARY KEY(contactPhone)
```

Example 1: Fetch the details of all passengers who have been issued a ticket.

```
SELECT fullname, contactPhone, gender
FROM NESTED TABLES
(ticket a descendants(ticket.passengerInfo b))
```

Explanation: This is an example of a join where the target table `ticket` is joined with its child table `passengerInfo`.

Output:

```
{"fullname":"Elane Lemons","contactPhone":"600-918-8404","gender":"F"}
{"fullname":"Adelaide Willard","contactPhone":"421-272-8082","gender":"M"}
{"fullname":"Dierdre Amador","contactPhone":"165-742-5715","gender":"M"}
{"fullname":"Doris Martin","contactPhone":"289-564-3497","gender":"F"}
{"fullname":"Adam Phillips","contactPhone":"893-324-1064","gender":"M"}
```

Example 1a: Fetch the details of the passenger with ticket number **1762324912391** .

```
SELECT fullname, contactPhone, gender
FROM NESTED TABLES
(ticket a descendants(ticket.passengerInfo b))
WHERE a.ticketNo=1762324912391
```

Explanation: This is an example of a join where the target table`ticket` is joined with its child table `passengerInfo`. Additionally, you can limit the result set by applying a filter condition to the result of the join. You are limiting the result to a particular ticket number.

Output:

```
{"fullname":"Elane Lemons","contactPhone":"600-918-8404","gender":"F"}
```

Example 2: Fetch all the bag details for all passengers who have been issued a ticket.

```
SELECT * FROM NESTED TABLES
(ticket a descendants(ticket.bagInfo b))
```

Explanation: This is an example of a join where the target table `ticket` is joined with its child table `bagInfo`.

Output:

```
{"a":{"ticketNo":1762344493810,"confNo":"LE6J4Z"},
"b":
{"ticketNo":1762344493810,"id":79039899165297,"tagNum":17657806255240,"
routing":"MIA/LAX/MEL",
"lastActionCode":"OFFLOAD","lastActionDesc":"OFFLOAD","lastSeenStation"
:"MEL",
"lastSeenTimeGmt":"2019-02-01T16:13:00.0000Z","bagArrivalDate":"2019-02
-01T16:13:00.0000Z"}}

{"a":{"ticketNo":1762324912391,"confNo":"LN0C8R"},
"b":
{"ticketNo":1762324912391,"id":79039899168383,"tagNum":1765780623244,"r
outing":"MXP/CDG/SLC/BZN",
"lastActionCode":"OFFLOAD","lastActionDesc":"OFFLOAD","lastSeenStation"
:"BZN",
"lastSeenTimeGmt":"2019-03-15T10:13:00.0000Z","bagArrivalDate":"2019-03
-15T10:13:00.0000Z"}}

{"a":{"ticketNo":1762392135540,"confNo":"DN3I4Q"},
"b":
{"ticketNo":1762392135540,"id":79039899156435,"tagNum":17657806224224,"
routing":"GRU/ORD/SEA",
"lastActionCode":"OFFLOAD","lastActionDesc":"OFFLOAD","lastSeenStation"
:"SEA",
"lastSeenTimeGmt":"2019-02-15T21:21:00.0000Z","bagArrivalDate":"2019-02
-15T21:21:00.0000Z"}}
```

```
{"a":{"ticketNo":1762376407826,"confNo":"ZG8Z5N"},
"b":
{"ticketNo":1762376407826,"id":7903989918469,"tagNum":17657806240229,"routing
":"JFK/MAD",
"lastActionCode":"OFFLOAD","lastActionDesc":"OFFLOAD","lastSeenStation":"MAD"
,
"lastSeenTimeGmt":"2019-03-07T13:51:00.0000Z","bagArrivalDate":"2019-03-07T13
:51:00.0000Z"}}

{"a":{"ticketNo":1762355527825,"confNo":"HJ4J4P"},
"b":
{"ticketNo":1762355527825,"id":79039899197492,"tagNum":17657806232501,"routin
g":"BZN/SEA/CDG/MXP",
"lastActionCode":"OFFLOAD","lastActionDesc":"OFFLOAD","lastSeenStation":"MXP"
,
"lastSeenTimeGmt":"2019-03-22T10:17:00.0000Z","bagArrivalDate":"2019-03-22T10
:17:00.0000Z"}}
```


Example 2a: Fetch all the bag details for a particular ticket number.

```
SELECT * FROM
NESTED TABLES (ticket a descendants(ticket.bagInfo b))
WHERE a.ticketNo=1762324912391
```

Explanation: This is an example of a join where the target table`ticket` is joined with its child table `bagInfo`. Additionally, you can limit the result set by applying a filter condition to the result of the join. You are limiting the result to a particular ticket number.

Output:


```
{"a":{"ticketNo":1762324912391,"confNo":"LN0C8R"},
"b":
{"ticketNo":1762324912391,"id":79039899168383,"tagNum":1765780623244,"routing
":"MXP/CDG/SLC/BZN",
"lastActionCode":"OFFLOAD","lastActionDesc":"OFFLOAD","lastSeenStation":"BZN"
,
"lastSeenTimeGmt":"2019-03-15T10:13:00.0000Z","bagArrivalDate":"2019-03-15T10
:13:00.0000Z"}}
```


Note:

If you move the non-join predicate restriction to the ON clause, the result set includes all the rows that meet the ON clause condition. Rows from the right outer table that do not meet the ON condition are populated with NULL values as shown below.

```
SELECT * FROM
NESTED TABLES(ticket a descendants(ticket.bagInfo b
ON a.ticketNo=b.ticketNo
AND a.ticketNo=1762324912391))
```

Output:

```
{"a":{"ticketNo":1762355527825,"confNo":"HJ4J4P"},"b":null}
{"a":{"ticketNo":1762344493810,"confNo":"LE6J4Z"},"b":null}
{"a":{"ticketNo":1762324912391,"confNo":"LN0C8R"}, "b":
{"ticketNo":1762324912391,"id":79039899168383,"tagNum":1765780623244,"r
outing":"MXP/CDG/SLC/BZN",
"lastActionCode":"OFFLOAD","lastActionDesc":"OFFLOAD","lastSeenStation"
:"BZN",
"lastSeenTimeGmt":"2019-03-15T10:13:00.0000Z","bagArrivalDate":"2019-03
-15T10:13:00.0000Z"}}
{"a":{"ticketNo":1762392135540,"confNo":"DN3I4Q"},"b":null}
{"a":{"ticketNo":1762376407826,"confNo":"ZG8Z5N"},"b":null}
```

Example 3: Fetch all flight leg details for all passengers.

```
SELECT * FROM
NESTED TABLES (ticket a descendants(ticket.bagInfo.flightLegs b))
```

Explanation: This is an example of a join where the target table `ticket` is joined with its descendant `bagInfo`. A descendant table can be any level hierarchically below a table (For example `fightLegs` is the child of `bagInfo` which is the child of `ticket`, so `flightLegs` is a descendant of `ticket`). All the rows from the `ticket` table will be fetched. If any row from the `ticket` table does not have a matching row in the `flightLegs` table, then NULL values will be displayed for those rows of the `flightLegs` table.

Output:

```
{"a":{"ticketNo":1762344493810,"confNo":"LE6J4Z"},
"b":
{"ticketNo":1762344493810,"id":79039899165297,"flightNo":"BM604","fligh
tDate":"2019-02-01T06:00:00.0000Z",
"fltRouteSrc":"MIA","fltRouteDest":"LAX","estimatedArrival":"2019-02-01
T11:00:00.0000Z",
"actions":[{"actionAt":"MIA","actionCode":"ONLOAD to
LAX","actionTime":"2019-02-01T06:13:00Z"},
{"actionAt":"MIA","actionCode":"BagTag Scan at
MIA","actionTime":"2019-02-01T05:47:00Z"},
{"actionAt":"MIA","actionCode":"Checkin at
MIA","actionTime":"2019-02-01T04:38:00Z"}]}}

{"a":{"ticketNo":1762344493810,"confNo":"LE6J4Z"},
"b":
{"ticketNo":1762344493810,"id":79039899165297,"flightNo":"BM667","fligh
tDate":"2019-02-01T06:13:00.0000Z",
"fltRouteSrc":"LAX","fltRouteDest":"MEL","estimatedArrival":"2019-02-01
T16:15:00.0000Z",
"actions":[{"actionAt":"MEL","actionCode":"Offload to Carousel at
MEL","actionTime":"2019-02-01T16:15:00Z"},
{"actionAt":"LAX","actionCode":"ONLOAD to
MEL","actionTime":"2019-02-01T15:35:00Z"},
{"actionAt":"LAX","actionCode":"OFFLOAD from
```

LAX","actionTime":"2019-02-01T15:18:00Z"}]}}

{"a":{"ticketNo":1762324912391,"confNo":"LN0C8R"},
"b":
{"ticketNo":1762324912391,"id":79039899168383,"flightNo":"BM170","flightDate"
:"2019-03-15T08:13:00.0000Z",
"fltRouteSrc":"SLC","fltRouteDest":"BZN","estimatedArrival":"2019-03-15T10:14
:00.0000Z",
"actions":[{"actionAt":"BZN","actionCode":"Offload to Carousel at
BZN","actionTime":"2019-03-15T10:13:00Z"},
{"actionAt":"SLC","actionCode":"ONLOAD to
BZN","actionTime":"2019-03-15T10:06:00Z"},
{"actionAt":"SLC","actionCode":"OFFLOAD from
SLC","actionTime":"2019-03-15T09:59:00Z"}]}}

{"a":{"ticketNo":1762324912391,"confNo":"LN0C8R"},
"b":
{"ticketNo":1762324912391,"id":79039899168383,"flightNo":"BM490","flightDate"
:"2019-03-15T08:13:00.0000Z",
"fltRouteSrc":"CDG","fltRouteDest":"SLC","estimatedArrival":"2019-03-15T10:14
:00.0000Z",
"actions":[{"actionAt":"CDG","actionCode":"ONLOAD to
SLC","actionTime":"2019-03-15T09:42:00Z"},
{"actionAt":"CDG","actionCode":"BagTag Scan at
CDG","actionTime":"2019-03-15T09:17:00Z"},
{"actionAt":"CDG","actionCode":"OFFLOAD from
CDG","actionTime":"2019-03-15T09:19:00Z"}]}}

{"a":{"ticketNo":1762324912391,"confNo":"LN0C8R"},
"b":
{"ticketNo":1762324912391,"id":79039899168383,"flightNo":"BM936","flightDate"
:"2019-03-15T08:00:00.0000Z",
"fltRouteSrc":"MXP","fltRouteDest":"CDG","estimatedArrival":"2019-03-15T09:00
:00.0000Z",
"actions":[{"actionAt":"MXP","actionCode":"ONLOAD to
CDG","actionTime":"2019-03-15T08:13:00Z"},
{"actionAt":"MXP","actionCode":"BagTag Scan at
MXP","actionTime":"2019-03-15T07:48:00Z"},
{"actionAt":"MXP","actionCode":"Checkin at
MXP","actionTime":"2019-03-15T07:38:00Z"}]}}

{"a":{"ticketNo":1762392135540,"confNo":"DN3I4Q"},
"b":
{"ticketNo":1762392135540,"id":79039899156435,"flightNo":"BM79","flightDate":
"2019-02-15T01:00:00.0000Z",
"fltRouteSrc":"GRU","fltRouteDest":"ORD","estimatedArrival":"2019-02-15T11:00
:00.0000Z",
"actions":[{"actionAt":"GRU","actionCode":"ONLOAD to
ORD","actionTime":"2019-02-15T01:21:00Z"},
{"actionAt":"GRU","actionCode":"BagTag Scan at
GRU","actionTime":"2019-02-15T00:55:00Z"},
{"actionAt":"GRU","actionCode":"Checkin at
GRU","actionTime":"2019-02-14T23:49:00Z"}]}}

{"a":{"ticketNo":1762392135540,"confNo":"DN3I4Q"}

,"b":
{"ticketNo":1762392135540,"id":79039899156435,"flightNo":"BM907","fligh
tDate":"2019-02-15T01:21:00.0000Z",
"fltRouteSrc":"ORD","fltRouteDest":"SEA","estimatedArrival":"2019-02-15
T21:22:00.0000Z",
"actions":[{"actionAt":"SEA","actionCode":"Offload to Carousel at
SEA","actionTime":"2019-02-15T21:16:00Z"},
{"actionAt":"ORD","actionCode":"ONLOAD to
SEA","actionTime":"2019-02-15T20:52:00Z"},
{"actionAt":"ORD","actionCode":"OFFLOAD from
ORD","actionTime":"2019-02-15T20:44:00Z"}]}}

{"a":{"ticketNo":1762376407826,"confNo":"ZG8Z5N"},
"b":
{"ticketNo":1762376407826,"id":7903989918469,"flightNo":"BM495","flight
Date":"2019-03-07T07:00:00.0000Z",
"fltRouteSrc":"JFK","fltRouteDest":"MAD","estimatedArrival":"2019-03-07
T14:00:00.0000Z",
"actions":[{"actionAt":"MAD","actionCode":"Offload to Carousel at
MAD","actionTime":"2019-03-07T13:54:00Z"},
{"actionAt":"JFK","actionCode":"ONLOAD to
MAD","actionTime":"2019-03-07T07:00:00Z"},
{"actionAt":"JFK","actionCode":"BagTag Scan at
JFK","actionTime":"2019-03-07T06:53:00Z"},
{"actionAt":"JFK","actionCode":"Checkin at
JFK","actionTime":"2019-03-07T05:03:00Z"}]}}

{"a":{"ticketNo":1762355527825,"confNo":"HJ4J4P"},
"b":
{"ticketNo":1762355527825,"id":79039899197492,"flightNo":"BM386","fligh
tDate":"2019-03-22T07:23:00.0000Z",
"fltRouteSrc":"CDG","fltRouteDest":"MXP","estimatedArrival":"2019-03-22
T10:24:00.0000Z",
"actions":[{"actionAt":"MXP","actionCode":"Offload to Carousel at
MXP","actionTime":"2019-03-22T10:15:00Z"},
{"actionAt":"CDG","actionCode":"ONLOAD to
MXP","actionTime":"2019-03-22T10:09:00Z"},
{"actionAt":"CDG","actionCode":"OFFLOAD from
CDG","actionTime":"2019-03-22T10:01:00Z"}]}}

{"a":{"ticketNo":1762355527825,"confNo":"HJ4J4P"},
"b":
{"ticketNo":1762355527825,"id":79039899197492,"flightNo":"BM578","fligh
tDate":"2019-03-22T07:23:00.0000Z",
"fltRouteSrc":"SEA","fltRouteDest":"CDG","estimatedArrival":"2019-03-21
T23:24:00.0000Z",
"actions":[{"actionAt":"SEA","actionCode":"ONLOAD to
CDG","actionTime":"2019-03-22T11:26:00Z"},
{"actionAt":"SEA","actionCode":"BagTag Scan at
SEA","actionTime":"2019-03-22T10:57:00Z"},
{"actionAt":"SEA","actionCode":"OFFLOAD from
SEA","actionTime":"2019-03-22T11:07:00Z"}]}}

{"a":{"ticketNo":1762355527825,"confNo":"HJ4J4P"},
"b":

```
{"ticketNo":1762355527825,"id":79039899197492,"flightNo":"BM704","flightDate"
:"2019-03-22T07:00:00.0000Z",
"fltRouteSrc":"BZN","fltRouteDest":"SEA","estimatedArrival":"2019-03-22T09:00
:00.0000Z",
"actions":[{"actionAt":"BZN","actionCode":"ONLOAD to
SEA","actionTime":"2019-03-22T07:23:00Z"},
{"actionAt":"BZN","actionCode":"BagTag Scan at
BZN","actionTime":"2019-03-22T06:58:00Z"},
{"actionAt":"BZN","actionCode":"Checkin at
BZN","actionTime":"2019-03-22T05:20:00Z"}]}}
```

Example 3a: Fetch all the flight leg details for a particular ticket number.

```
SELECT * FROM
NESTED TABLES (ticket.bagInfo.flightLegs b ancestors(ticket a))
WHERE a.ticketNo=1762344493810
```

Explanation: This is an example of a join where the target table`ticket` is joined with its descendant `bagInfo`. Additionally, you can limit the result set by applying a filter condition to the result of the join. You are limiting the result to a particular ticket number.

The result has two rows, implying there are two flight legs for this ticket number.

Output:

```
"a":{"ticketNo":1762344493810,"confNo":"LE6J4Z"},
"b":{"ticketNo":1762344493810,"id":79039899165297,"flightNo":"BM604",
"flightDate":"2019-02-01T06:00:00.0000Z","fltRouteSrc":"MIA","fltRouteDest":"
LAX",
"estimatedArrival":"2019-02-01T11:00:00.0000Z",
"actions":[{"actionAt":"MIA","actionCode":"ONLOAD to
LAX","actionTime":"2019-02-01T06:13:00Z"},
{"actionAt":"MIA","actionCode":"BagTag Scan at
MIA","actionTime":"2019-02-01T05:47:00Z"},
{"actionAt":"MIA","actionCode":"Checkin at
MIA","actionTime":"2019-02-01T04:38:00Z"}]}}

{"a":{"ticketNo":1762344493810,"confNo":"LE6J4Z"},
"b":{"ticketNo":1762344493810,"id":79039899165297,"flightNo":"BM667",
"flightDate":"2019-02-01T06:13:00.0000Z","fltRouteSrc":"LAX","fltRouteDest":"
MEL",
"estimatedArrival":"2019-02-01T16:15:00.0000Z",
"actions":[{"actionAt":"MEL","actionCode":"Offload to Carousel at
MEL","actionTime":"2019-02-01T16:15:00Z"},
{"actionAt":"LAX","actionCode":"ONLOAD to
MEL","actionTime":"2019-02-01T15:35:00Z"},
{"actionAt":"LAX","actionCode":"OFFLOAD from
LAX","actionTime":"2019-02-01T15:18:00Z"}]}}
```

Example 4: Fetch the bag id and number of hops for all bags of all passengers.

```
SELECT b.id,count(*) AS NUMBER_HOPS
FROM NESTED TABLES (ticket a descendants(ticket.bagInfo.flightLegs b))
GROUP BY b.id
```

Explanation: You group the data based on the bag id (using GROUP BY) and get the count of flight legs (using count()) for every bag.

Output:

```
{"id":79039899168383,"NUMBER_HOPS":3}
{"id":79039899156435,"NUMBER_HOPS":2}
{"id":7903989918469,"NUMBER_HOPS":1}
{"id":79039899165297,"NUMBER_HOPS":2}
{"id":79039899197492,"NUMBER_HOPS":3}
```

Example 4a: Find the number of hops for all bags of a particular passenger.

```
SELECT b.id,count(*) AS NUMBER_HOPS FROM
NESTED TABLES (ticket a descendants(ticket.bagInfo.flightLegs b))
WHERE a.ticketNo=1762355527825
GROUP BY b.id
```

Explanation: You group the data based on the bag id (using GROUP BY) and get the count of flight legs (Using count()) for every bag. Additionally, you filter the results for a particular ticket number.

Output:

```
{"id":79039899197492,"NUMBER_HOPS":3}
```

Example 5: Fetch bag id and routing details of all bags that arrived after 2019.

```
SELECT b.id, routing FROM
NESTED TABLES(ticket a descendants(ticket.bagInfo b))
WHERE CAST (b.bagArrivalDate AS Timestamp(0))>=
CAST ("2019-01-01T00:00:00" AS Timestamp(0))
```

Explanation: This is an example of a join where the target table`ticket` is joined with its child table `bagInfo`. The filter condition is applied on the `bagArrivalDate`. The `CAST` function is used to convert the string into Timestamp and then the values are compared.

Output:

```
{"id":79039899197492,"routing":"BZN/SEA/CDG/MXP"}
{"id":79039899165297,"routing":"MIA/LAX/MEL"}
{"id":79039899168383,"routing":"MXP/CDG/SLC/BZN"}
{"id":79039899156435,"routing":"GRU/ORD/SEA"}
{"id":7903989918469,"routing":"JFK/MAD"}
```

Tuning and Optimizing SQL queries

Query optimization is the overall process of choosing the most efficient means of executing a SQL statement.

You optimize a SQL query to get accurate and fast database results.

Using Indexes for query optimization

Indexing is a way to optimize the performance of a database by minimizing the number of disk accesses required when a query is processed.

In Oracle NoSQL Database, the query processor can identify which of the available indexes are beneficial for a query and rewrite the query to make use of such an index. "Using" an index means scanning a contiguous subrange of its entries, potentially applying further filtering conditions on the entries within this subrange, and using the primary keys stored in the surviving index entries to extract and return the associated table rows. The subrange of the index entries to scan is determined by the conditions appearing in the WHERE clause, some of which may be converted to search conditions for the index. Given that only a (hopefully small) subset of the index entries will satisfy the search conditions, the query can be evaluated without accessing each individual table row, thus saving a potentially large number of disk accesses.

Notice that in Oracle NoSQL Database, a primary-key index is always created by default. This index maps the primary key columns of a table to the physical location of the table rows. Furthermore, if no other index is available, the primary index will be used. In other words, there is no pure "table scan" mechanism; a table scan is equivalent to a scan via the primary-key index. When it comes to indexes and queries, the query processor must answer two questions:

1. Is an index applicable to a query? That is, will accessing the table via this index be more efficient than doing a full table scan (via the primary index).
2. Among the applicable indexes, which index or combination of indexes is the best to use?

There are no statistics on the number and distribution of values in a table column. As a result, the query processor has to rely on some simple heuristics in choosing among the applicable indexes. In addition, SQL for Oracle NoSQL Database allows for the inclusion of index hints in the queries. You can use index hints to force the use of a particular index in queries.

Examples of queries using index

You can write simple queries to understand how an index is used.

Query 1:

Fetch the bag details of passengers for ticket numbers satisfying 2 range of values.

```
SELECT fullname, ticketNo,bag.bagInfo[].tagNum,
bag.bagInfo[].routing
```

```
FROM BaggageInfo bag WHERE 1762340000000 < ticketNo
AND ticketNo < 1762352000000
```

In the above example, the query contains 2 index predicates. The primary key index is used as `ticketNo` is the primary key here. For the primary key index, 1762340000000 < ticketNo is a start predicate and ticketNo < 1762352000000 is a stop predicate.

A portion of the query plan is shown below. You can see the primary index being used.

```
"iterator kind" : "TABLE",
"target table" : "BaggageInfo",
"row variable" : "$$bag",
"index used" : "primary index",
"covering index" : false,
"index scans" :
[
  {
    "equality conditions" : {},
    "range conditions" : { "ticketNo" : { "start value" :
1762340000000,
       "start inclusive" : false,
       "end value" : 1762352000000,
       "end inclusive" : false } }
  }
]
```

For more information on how a query is executed, see Query execution plan.

Query 2:

Fetch the bag details of passengers for ticket numbers satisfying one of the two ranges of values.

```
SELECT fullname, ticketNo,bag.bagInfo[].tagNum,
bag.bagInfo[].routing
FROM BaggageInfo bag
WHERE ticketNo > 1762340000000 OR
ticketNo < 1762352000000
```

In the above example, the query contains 1 index predicate, which is the whole WHERE expression. The primary key index is used as `ticketNo` is the primary key here. The predicate is a filtering predicate.

A portion of the query plan is shown below. You can see the primary index and the index filtering predicates being used.

```
"iterator kind" : "TABLE",
"target table" : "BaggageInfo",
"row variable" : "$$bag",
"index used" : "primary index",
"covering index" : false,
"index scans" :
[
  {
```

```
    "equality conditions" : {},
    "range conditions" : {}
  }
],
"index filtering predicate" :
{
  "iterator kind" : "OR",
  "input iterators" :
  [
    {
      "iterator kind" : "GREATER_THAN",
      "left operand" :
        {
          ---
        },
      "right operand" :
        {
          ---
        }
    },
    {
      "iterator kind" : "LESS_THAN",
      "left operand" :
        {
          ---
        },
      "right operand" :
        {
          ---
        }
    }
  ]
}
```

For more information on how a query is executed, see Query execution plan.

Query 3:

Fetch the bag details for a particular reservation code.

```
SELECT fullName,bag.ticketNo, bag.confNo,
bag.bagInfo[].tagNum,bag.bagInfo[].routing
FROM BaggageInfo bag WHERE bag.confNo="FH7G1W"
```

In the above example, two indexes are applicable `compindex_tckNoconfNo` and `fixedschema_conf` .

A portion of the query plan is shown below. The `fixedschema_conf` is used as that is a single index on `ticketNo`. An index scan is performed with the equality condition.

```
"iterator kind" : "TABLE",
"target table" : "BaggageInfo",
"row variable" : "$$bag",
"index used" : "fixedschema_conf",
```

```
"covering index" : false,
"index scans" :
[
  {
    "equality conditions" : {"confNo":"FH7G1W"},
    "range conditions" : {}
  }
]
```

For more information on how a query is executed, see Query execution plan.

Query 4:

Fetch the name and routing details of all male passengers.

```
SELECT fullname,bag.bagInfo[].routing FROM BaggageInfo bag
WHERE gender!="F"
```

In the above example, there is no index predicate, because no index has information about gender.

A portion of the query plan is shown below. As there are no available indexes to be used, only the primary key index is used.

```
"iterator kind" : "TABLE",
"target table" : "BaggageInfo",
"row variable" : "$$bag",
"index used" : "primary index",
"covering index" : false,
"index scans" :
[
  {
    "equality conditions" : {},
    "range conditions" : {}
  }
]
```

For more information on how a query is executed, see Query execution plan.

Query 5:

Fetch the name and phone number for all passengers.

```
SELECT bag.contactPhone, bag.fullName FROM BaggageInfo bag
ORDER BY bag.fullName
```

In the above example, only the index `compindex_namephone` is applicable. The sort (for the order by clause) will be index-based because the order-by expression matches the 1st field of the index used by the query. In this case, the full name and contact phone 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. So the index `compindex_namephone` is a covering index in this example. The query processor will apply this optimization.

A portion of the query plan is shown below. You can see the index `compindex_namephone` is used and it is a covering index.

```
"iterator kind" : "TABLE",
"target table" : "BaggageInfo",
"row variable" : "$$bag",
"index used" : "compindex_namephone",
"covering index" : true,
"index row variable" : "$$bag_idx",
"index scans" :
[
  {
    "equality conditions" : {},
    "range conditions" : {}
  }
]
```

For more information on how a query is executed, see Query execution plan.

Query 6:

Fetch the name, ticket number, and arrival date of passengers whose arrival date is greater than a given value.

```
SELECT fullName, bag.ticketNo, bag.bagInfo[].bagArrivalDate
FROM BaggageInfo bag WHERE EXISTS
bag.bagInfo[$element.bagArrivalDate >="2019-01-01T00:00:00"]
```

In the above example, the EXISTS condition is actually converted to a filtering predicate. There is one filtering predicate which is the whole WHERE expression.

A portion of the query plan is shown below. The index `simpleindex_arrival` is used in this example.

```
"iterator kind" : "TABLE",
"target table" : "BaggageInfo",
"row variable" : "$$bag",
"index used" : "simpleindex_arrival",
"covering index" : false,
"index row variable" : "$$bag_idx",
"index scans" :
[
  {
    "equality conditions" : {},
    "range conditions" : {}
  }
],
"index filtering predicate" :
{
  "iterator kind" : "GREATER_OR_EQUAL",
  "left operand" :
    {
      ---
    },
```

```
  "right operand" :
    {
      ---
    }
}
```

For more information on how a query is executed, see Query execution plan.

Query 7:

Fetch the reservation code and count of bags for all passengers.

```
SELECT bag.confNo, count(bag.bagInfo) AS TOTAL_BAGS
FROM BaggageInfo bag GROUP BY bag.confNo
```

In the above example, two indexes `fixedschema_conf` and `compindex_tckNoconfNo` are applicable.

A portion of the query plan is shown below. The index `fixedschema_conf` is used as that is a single index with only one column `confNo`. For this query, the group-by is index-based. As you need the entire bagInfo details to determine the number of bags using the aggregate **count** function, the index here is not covering.

```
"iterator kind" : "TABLE",
"target table" : "BaggageInfo",
"row variable" : "$$bag",
"index used" : "fixedschema_conf",
"covering index" : false,
"index scans" :
[
  {
    "equality conditions" : {},
    "range conditions" : {}
  }
]
```

For more information on how a query is executed, see Query execution plan.

Query 8:

Fetch the full name and tag number of passengers who are in the given list of names.

```
SELECT bagdet.fullName, bagdet.bagInfo[].tagNum
FROM BaggageInfo bagdet
WHERE bagdet.fullName IN
("Lucinda Beckman", "Adam Phillips",
"Zina Christenson","Fallon Clements")
```

In the above example, only the index `compindex_namephone` is applicable.

A portion of the query plan is shown below. The index `compindex_namephone` is used. An index scan is performed on `compindex_namephone` evaluating four equality predicates.

```
"iterator kind" : "TABLE",
"target table" : "BaggageInfo",
"row variable" : "$$bagdet",
"index used" : "compindex_namephone",
"covering index" : false,
"index scans" :
[
  {
    "equality conditions" : {"fullName":"Lucinda Beckman"},
    "range conditions" : {}
  },
  {
    "equality conditions" : {"fullName":"Adam Phillips"},
    "range conditions" : {}
  },
  {
    "equality conditions" : {"fullName":"Zina Christenson"},
    "range conditions" : {}
  },
  {
    "equality conditions" : {"fullName":"Fallon Clements"},
    "range conditions" : {}
  }
]
```

For more information on how a query is executed, see Query execution plan.

Query 9:

Select the ticket details(ticket number, reservation code, tag number, and routing) for a passenger with a specific ticket number and reservation code.

```
SELECT fullName,bag.ticketNo, bag.confNo,
bag.bagInfo[].tagNum,bag.bagInfo[].routing
FROM BaggageInfo bag WHERE
bag.ticketNo=1762311547917
AND bag.confNo="FH7G1W"
```

In the above example, though the index `compindex_tckNoconfNo` is available, only the primary index (for `ticketNo`) gets used. An index scan is performed on the primary index and the WHERE expression is evaluated.

A portion of the query plan is shown below.

```
"iterator kind" : "TABLE",
"target table" : "BaggageInfo",
"row variable" : "$$bag",
"index used" : "primary index",
"covering index" : false,
"index scans" :
[
```

```
  {
    "equality conditions" : {"ticketNo":1762311547917},
    "range conditions" : {}
  }
]
```

For more information on how a query is executed, see Query execution plan.

Query 10:

Fetch the source of passenger bags and the count of bags for all passengers and group the data by the source.

```
SELECT $flt_src as SOURCE, count(*) as COUNT
FROM BaggageInfo $bag,
$bag.bagInfo.flightLegs[0].fltRouteSrc $flt_src
GROUP BY $flt_src
```

In the above example, there is no index on the `fltRouteSrc` field. So the grouping is done in a generic way. An internal variable is created that iterates over the records produced by the SELECT statement.

A portion of the query plan is shown below. The primary index is being used.

```
"iterator kind" : "TABLE",
"target table" : "BaggageInfo",
"row variable" : "$bag",
"index used" : "primary index",
"covering index" : false,
"index scans" :
[
  {
    "equality conditions" : {},
    "range conditions" : {}
  }
]
```

For more information on how a query is executed, see Query execution plan.

Managing GeoJSON data

The GeoJson specification defines the structure and content of json objects that are supposed to represent geographical shapes on earth (called geometries).

According to the GeoJson specification, for a JSON object to be a geometry object it must have two fields called ***type*** and ***coordinates***, where the value of the **type** field specifies the kind of geometry and the value of **coordinates** must be an array whose elements define the geometrical shape. See About GeoJSON Data for more details on the various types of geometry objects. All kinds of geometries are specified in terms of a set of positions. However, for line strings and polygons, the actual geometrical shape is formed by the lines connecting their positions. The GeoJson specification defines a line between two points as the straight line that connects the points in the (flat)

cartesian coordinate system whose horizontal and vertical axes are the longitude and latitude, respectively. See Lines and Coordinate System for more details.

If you want to follow along with the examples, download the script geojsonschema_loaddata.sql and run it as shown below. This script creates the table used in the example and loads data into the table.

Start your KVSTORE or KVLite and open the SQL.shell.

```
java -jar lib/kvstore.jar kvlite -secure-config disable
java -jar lib/sql.jar -helper-hosts localhost:5000 -store kvstore
```

Using the `load` command, run the script.

```
load -file geojsonschema_loaddata.sql
```

Oracle NoSQL Database implements a number of functions that interpret JSON objects as geometries and allow for the search for rows containing geometries that satisfy certain conditions.

- geo_inside
- geo_intersect
- geo_distance
- geo_within_distance
- geo_near
- geo_is_geometry

geo_inside

Determines geometries within a bounding GeoJSON geometry.

```
boolean geo_inside(any*, any*)
```

- The first parameter `any*` can be any geometric object.
- The second parameter `any*` needs to be a polygon.

The function determines if the geometry pointed by the first parameter is completely contained inside the polygon pointed by the second parameter.

If any of the two parameters does not return a single valid geometry object, and if it can be detected at compile time then the function raises an error.

The runtime behavior is as follows:

- Returns false if any parameter returns 0 or more than 1 item.
- Returns NULL if any parameter returns NULL.
- Returns false if any parameter (at runtime) returns an item that is not a valid geometry object.
- Returns false if the second parameter returns a geometry object that is not a polygon.

- If both parameters return a single geometry object each and the second geometry is a polygon.
  - It returns true if the first geometry is completely contained inside the second polygon, i.e., all its points belong to the interior of the polygon.
  - Else it returns false.

> **Note:**
>
> The interior of a polygon is all the points in the polygon area except the points on the linear ring that define the polygon's boundary.

Example: Look for nature parks in Northern California.

```
SELECT t.poi.name AS park_name,
t.poi.address.street AS park_location
FROM PointsOfInterest t
WHERE t.poi.kind = "nature park"
AND geo_inside(t.poi.location,
              { "type" : "polygon",
                "coordinates": [[
                  [-120.1135253906249, 36.99816565700228],
                  [-119.0972900390625, 37.391981943533544],
                  [-119.2840576171875, 37.97451499202459],
                  [-120.2069091796874, 38.035112420612975],
                  [-122.3822021484375, 37.74031329210266],
                  [-122.2283935546875, 37.15156050223665],
                  [-121.5362548828124, 36.85325222344018],
                  [-120.1135253906249, 36.99816565700228]
                ]]
             });
```

Explanation:

- You query the `PointsOfInterest` table to filter the rows for **nature park**.
- You specify a polygon as the second parameter to the `geo_inside` function.
- The coordinates of the polygon you specify correspond to the coordinates of the northern portion of the state of California in the U.S.
- The `geo_inside` function only returns rows when the location of the nature park is completely contained inside the location points specified.

Result:

```
{"park_name":"portola redwoods state park",
"park_location":"15000 Skyline Blvd"}
```

geo_intersect

Determines geometries that intersect with a GeoJSON geometry.

```
boolean geo_intersect(any*, any*)
```

The first and the second parameters `any*` can be any geometric object.

The function determines if two geometries that are specified as parameters have any points in common. If any of the two parameters does not return a single valid geometry object, and if it can be detected at compile time then the function raises an error.

The runtime behavior is as follows:

- Returns false if any parameter returns 0 or more than 1 item.
- Returns NULL if any parameter returns NULL.
- Returns false if any parameter (at runtime) returns an item that is not a valid geometry object.

If both parameters return a single geometry object each, the function returns true if the 2 geometries have any points in common; otherwise false.

Example: Texas is considering regulating access to the underground water supply. An aquifer is an underground layer of water-bearing permeable rock, rock fractures, or unconsolidated materials. The government wants to impose new regulations for locations that are very close to an aquifer.

The coordinates of the aquifer have already been mapped. You want to know all counties in the Texas state that intersect with that aquifer so that you can notify the county government for each affected county to participate in talks for the new regulations.

```
SELECT t.poi.county AS County_needs_regulation,
t.poi.contact AS Contact_phone
FROM PointsOfInterest t WHERE
geo_intersect(
    t.poi.location,
    {

     "type" : "polygon",
      "coordinates": [
          [
            [-97.668457031249, 29.34387539941801],
            [-95.207519531258, 29.19053283229458],
            [-92.900390625653, 30.37287518811801],
            [-94.636230468752, 32.21280106801518],
            [-97.778320312522, 32.45415593941475],
            [-99.799804687541, 31.18460913574325],
            [-97.668457031249, 29.34387539941801]
          ]
        ]
    }
);
```

Explanation:

- The above query fetches the locations which intersect with the location of the aquifer. That is if the location coordinates have any points in common with the location of the aquifer.
- You use `geo_intersect` to see if the coordinates of the location have any points common with the coordinates of the aquifer that are specified.

Result:

```
{"County_needs_regulation":"Tarrant","Contact_phone":"469 745 5687"}
{"County_needs_regulation":"Kinga","Contact_phone":"469 384 7612"}
```

geo_distance

Determines distance between two geospatial objects.

```
double geo_distance(any*, any*)
```

The first and the second parameters `any*` can be any geometric object.

The function returns the geodetic distance between the two input geometries. The returned distance is the minimum among the distances of any pair of points where the first point belongs to the first geometry and the second point to the second geometry. Between two such points, their distance is the length of the geodetic line that connects the points.

Overview of Geodetic Line

A geodetic line between 2 points is the shortest line that can be drawn between the 2 points on the ellipsoidal surface of the earth. For a simplified, but more illustrative definition, assume for a moment that the earth's surface is a sphere. Then, the geodetic line between two points on the earth is the minor arc between the two points on the great circle corresponding to the points, i.e., the circle that is formed by the intersection of the sphere and the plane defined by the center of the earth and the two points.

The following figure shows the difference between the geodetic and straight lines between Los Angeles and London.

If any of the two parameters does not return a single valid geometry object, and if it can be detected at compile time then the function raises an error.

The runtime behavior is as follows:

- Returns -1 if any parameter returns zero or more than 1 item.
- Returns NULL if any parameter returns NULL.
- Returns -1 if any of the parameters is not a geometry object.

Otherwise, the function returns the geodetic distance in meters between the 2 input geometries.

> **Note:**
>
> The results are sorted ascending by distance(displaying the shortest distance first).

Example: How far is the nearest restaurant from the given location?

```
SELECT
t.poi.name AS restaurant_name,
t.poi.address.street AS street_name,
geo_distance(
    t.poi.location,
    {
       "type" : "point",
       "coordinates": [-121.94034576416016,37.2812239247177]
    }
) AS distance_in_meters
FROM PointsOfInterest t
WHERE t.poi.kind = "restaurant" ;
```

Explanation:

- You query the `PointsOfInterest` table to filter the rows for **restaurant**.
- You provide the correct location point and determine the distance using the `geo_distance` function.

Result:

```
{"restaurant_name":"Coach Sports Bar & Grill","street_name":"80 Edward
St","distance_in_meters":799.2645323337218}
{"restaurant_name":"Ricos Taco","street_name":"80 East Boulevard
St","distance_in_meters":976.5361117138553}
{"restaurant_name":"Effie's Restaurant and Bar","street_name":"80 Woodeard
St","distance_in_meters":2891.0508307646282}
```

The distance between the current location and the nearest restaurant is 799 meters.

geo_within_distance

Determines geospatial objects in proximity to a point.

```
boolean geo_within_distance(any*, any*,double)
```

The first and the second parameters `any*` can be any geometric object.

The function determines if the first geometry is within a distance of N meters from the second geometry.

If any of the two parameters does not return a single valid geometry object, and if it can be detected at compile time then the function raises an error.

The runtime behavior is as follows:

- Returns false if any parameter returns 0 or more than 1 item.
- Returns NULL if any of the first two parameters returns NULL.
- Returns false if any of the first two parameters returns an item that is not a valid geometry object.

Finally, if both the parameters return a single geometry object each, it returns true if the first geometry is within a distance of N meters from the second geometry, where N is the number returned by the third parameter; otherwise false. The distance between 2 geometries is defined as the minimum among the distances of any pair of points where the first point belongs to the first geometry and the second point to the second geometry. If N is a negative number, it is set to 0.

Example: Is a city hall there in the next 5 km? How far is it?

```
SELECT t.poi.address.street AS city_hall_address,
geo_distance(
    t.poi.location,
    {
        "type" : "point",
        "coordinates" : [-120.653828125,38.85682013474361]
    }

) AS distance_in_meters
FROM PointsOfInterest t
WHERE t.poi.kind = "city hall" AND
geo_within_distance(
    t.poi.location,
    {
        "type" : "point",
        "coordinates" : [-120.653828125,38.85682013474361]
    },
    5000
);
```

Explanation:

- You query the `PointsOfInterest` table to filter the rows for **city hall**.
- You use the `geo_within_distance` function to filter city hall within 5 km (5000m) of the given location.
- You also fetch the actual distance between your location and the city hall using the `geo_distance` function.

Result:

```
{"city_hall_address":"70 North 1st
street","distance_in_meters":1736.0144040331768}
```

The city hall is 1736 m(1.73 km) from the current location.

geo_near

Determines geospatial objects in proximity to a point.

```
boolean geo_near(any*, any*, double)
```

The first and the second parameters `any*` can be any geometric object.

The function determines if the first geometry is within a distance of N meters from the second geometry.

If any of the two parameters does not return a single valid geometry object, and if it can be detected at compile time then the function raises an error.

The runtime behavior is as follows:

- Returns false if any parameter returns 0 or more than 1 item.
- Returns NULL if any of the first two parameters returns NULL.
- Returns false if any of the first two parameters returns an item that is not a valid geometry object.

Finally, if both of the first two parameters return a single geometry object each, it returns true if the first geometry is within a distance of N meters from the second geometry, where N is the number returned by the third parameter; otherwise false.

> **Note:**
>
> geo_near is converted internally to geo_within_distance plus an (implicit) order by the distance between the two geometries. However, if the query has an (explicit) order-by already, no ordering by distance is performed. The geo_near function can appear in the WHERE clause only, where it must be a top-level predicate, i.e, not nested under an OR or NOT operator.

Example 1: Is there a hospital within 3km of the given location?

```
SELECT
t.poi.name AS hospital_name,
t.poi.address.street AS hospital_address
FROM PointsOfInterest t
WHERE t.poi.kind = "hospital"
AND
geo_near(
    t.poi.location,
    {"type" : "point",
     "coordinates" : [-122.03493933105469,37.32949164059004]
```

```
    },
    3000
);
```

Explanation:

- You query the `PointsOfInterest` table to filter the rows for **hospital**.
- You use the `geo_near` function to filter hospitals within 3000m of the given location.

Result:

```
{"hospital_name":"St. Marthas hospital","hospital_address":"18000 West
Blvd"}
{"hospital_name":"Memorial hospital","hospital_address":"10500 South
St"}
```

Example 2: How far is a gas station within the next one mile from the given location?

```
SELECT
t.poi.address.street AS gas_station_address,
geo_distance(
    t.poi.location,
    {
        "type" : "point",
        "coordinates" : [-121.90768646240233,37.292081740702365]
    }
) AS distance_in_meters
FROM PointsOfInterest t
WHERE t.poi.kind = "gas station" AND
geo_near(
    t.poi.location,
    {
        "type" : "point",
        "coordinates" : [-121.90768646240233,37.292081740702365]
    },
    1600
);
```

Explanation:

- You query the `PointsOfInterest` table to filter the rows for **gas station**.
- You use the `geo_near` function to filter gas stations within one mile(1600m) of the given location.
- You also fetch the actual distance between your location and the gas station using the `geo_distance` function.

Result:

```
{"gas_station_address":"33 North
Avenue","distance_in_meters":886.7004173859665}
```

The actual distance to the nearest gas station within the next mile is 886m.

geo_is_geometry

Validates a geospatial object.

```
boolean geo_is_geometry(any*)
```

The parameter `any*` can be any geometric object.

The function determines if the given input is a valid geometry object.

- Returns false if the parameter returns zero or more than 1 item.
- Returns NULL if the parameter returns NULL.
- Returns true if the input is a single valid geometry object. Otherwise, false.

Example: Determine if the location pointing to the **city hall** is a valid geometric object.

```
SELECT geo_is_geometry(t.poi.location) AS city_hall
FROM PointsOfInterest t
WHERE t.poi.kind = "city hall"
```

Explanation: You use the function geo_is_geometry to determine if a given location is a valid geometric object or not.

Result:

```
{ "city_hall" : true}
```

5
Reference

The articles in this section contain reference information related to various operators, constructs and expressions used in SQL.

Operators in SQL

If you want to follow along with the examples, download the script baggageschema_loaddata.sql and run it as shown below. This script creates the table used in the example and loads data into the table.

Start your KVSTORE or KVLite and open the SQL.shell.

```
java -jar lib/kvstore.jar kvlite -secure-config disable
java -jar lib/sql.jar -helper-hosts localhost:5000 -store kvstore
```

Using the `load` command, run the script.

```
load -file baggageschema_loaddata.sql
```


Sequence Comparison Operators

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

Example 1: Find passenger name and tag number for all bags where the estimated arrival time is greater than **2019-03-01T13:00:00Z**.

```
SELECT fullname, bag.bagInfo[].tagNum,
bag.bagInfo[].flightLegs[].estimatedArrival
```

```
FROM BaggageInfo bag
WHERE bag.bagInfo[].flightLegs[].estimatedArrival >any
"2019-03-01T13:00:00Z"
```

Explanation: You fetch the full name, and tag number of all passenger bags whose estimated arrival time is greater than the given value. Here the operand on the left hand of the ">" operator (`bag.bagInfo[].flightLegs[].estimatedArrival`) is a sequence of values. If you try using the regular comparison operator instead of the sequence operator, you get an error as shown below. That is the reason you need a sequence operator here.

```
SELECT fullname, bag.bagInfo[].tagNum,
bag.bagInfo[].flightLegs[].estimatedArrival
FROM BaggageInfo bag
WHERE bag.bagInfo[].flightLegs[].estimatedArrival >
"2019-03-01T13:00:00Z"
```

Output showing error:

```
Error handling command SELECT fullname,
bag.bagInfo[].tagNum,bag.bagInfo[].flightLegs[].estimatedArrival
FROM BaggageInfo bag WHERE bag.bagInfo[].flightLegs[].estimatedArrival
> "2019-03-01T13:00:00Z":
Error: at (1, 107) The left operand of comparison operator > is a
sequence with more than one items.
Comparison operators cannot operate on sequences of more than one
items.
```

Output (after using sequence operator):


```
{"fullname":"Lucinda
Beckman","tagNum":"17657806240001","estimatedArrival":
["2019-03-12T16:00:00Z","2019-03-13T03:14:00Z","2019-03-12T15:12:00Z"]}
{"fullname":"Elane Lemons","tagNum":"1765780623244","estimatedArrival":
["2019-03-15T09:00:00Z","2019-03-15T10:14:00Z","2019-03-15T10:14:00Z"]}
{"fullname":"Dierdre
Amador","tagNum":"17657806240229","estimatedArrival":"2019-03-07T14:00:
00Z"}
{"fullname":"Henry
Jenkins","tagNum":"17657806216554","estimatedArrival":
["2019-03-02T09:00:00Z","2019-03-02T13:24:00Z"]}
{"fullname":"Lorenzo Phil","tagNum":
["17657806240001","17657806340001"],"estimatedArrival":
["2019-03-12T16:00:00Z","2019-03-13T03:14:00Z",
"2019-03-12T15:12:00Z","2019-03-12T16:40:00Z","2019-03-13T03:18:00Z","2
019-03-12T15:12:00Z"]}
{"fullname":"Gerard
Greene","tagNum":"1765780626568","estimatedArrival":
["2019-03-07T17:00:00Z","2019-03-08T04:10:00Z","2019-03-07T16:10:00Z"]}
{"fullname":"Doris
Martin","tagNum":"17657806232501","estimatedArrival":
["2019-03-22T09:00:00Z","2019-03-21T23:24:00Z","2019-03-22T10:24:00Z"]}
{"fullname":"Omar Harvey","tagNum":"17657806234185","estimatedArrival":
```

```
["2019-03-02T02:00:00Z","2019-03-02T16:21:00Z"]}
{"fullname":"Mary Watson","tagNum":"17657806299833","estimatedArrival":
["2019-03-13T15:00:00Z","2019-03-14T06:22:00Z"]}
{"fullname":"Kendal Biddle","tagNum":"17657806296887","estimatedArrival":
["2019-03-04T22:00:00Z","2019-03-05T12:02:00Z"]}
```

Example 2: Find the tag number of passengers who fly from JFK/through JFK to any other location.

```
SELECT bag.bagInfo[].tagNum,bag.bagInfo[].flightLegs[].fltRouteSrc
FROM BaggageInfo bag
WHERE bag.bagInfo[].flightLegs[].fltRouteSrc=any "JFK"
```

Explanation: You fetch the tag number of passengers whose flight source is JFK or the passengers who travel through JFK. The destination can be anything.

Output:

```
{"tagNum":"17657806240229","fltRouteSrc":"JFK"}
{"tagNum":"17657806215913","fltRouteSrc":["JFK","IST"]}
{"tagNum":"17657806296887","fltRouteSrc":["JFK","IST"]}
```

Logical operators

The operators AND and OR are binary and the NOT operator is unary. The operands of the logical operators are conditional expressions, which must have a type `BOOLEAN`. An empty result from an operand is treated as a false value. If an operand returns NULL(either SQL NULL or JSON NULL), then:

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

Example 1: Select the details of the passenger and their bags for a trip with ticket number **1762311547917** or confirmation number **KN4D1L**.

```
SELECT fullName,bag.ticketNo, bag.confNo,
bag.bagInfo[].tagNum, bag.bagInfo[].routing
FROM BaggageInfo bag
WHERE bag.ticketNo=1762311547917 OR bag.confNo="KN4D1L"
```

Explanation: You fetch the details of passengers satisfying one of the two filter criteria. You do this with the **OR** clause. You fetch the full name, tag number, ticket number, reservation code, and routing details of passengers satisfying a particular ticket number or a particular reservation code (`confNo`).

Output:

```
{"fullName":"Rosalia
Triplett","ticketNo":1762311547917,"confNo":"FH7G1W","tagNum":"17657806215913
```

```
","routing":"JFK/IST/VIE"}
{"fullName":"Mary
Watson","ticketNo":1762340683564,"confNo":"KN4D1L","tagNum":"1765780629
9833","routing":"YYZ/HKG/BLR"}
```

Example 2: Select baggage details of passengers traveling between **MIA** and **MEL**.

```
SELECT fullName, bag.bagInfo[].tagNum, bag.bagInfo[].routing
FROM BaggageInfo bag
WHERE bag.bagInfo[].flightLegs[].fltRouteSrc =any "MIA" AND
bag.bagInfo[].flightLegs[].fltRouteDest=any "MEL"
```

Explanation: You fetch the details of the passengers traveling between MIA and MEL. Since you need to match 2 conditions here, the flight source and the flight destination, you are using an **AND** operator. Here the flight source could be the starting point of the flight or any transit airport. Similarly, the flight destination could be a transit airport or a final destination.

Output:

```
{"fullName":"Zulema
Martindale","tagNum":"17657806288937","routing":"MIA/LAX/MEL"}
{"fullName":"Adam
Phillips","tagNum":"17657806255240","routing":"MIA/LAX/MEL"}
{"fullName":"Joanne Diaz","tagNum":"17657806292518","routing":"MIA/LAX/
MEL"}
{"fullName":"Zina
Christenson","tagNum":"17657806228676","routing":"MIA/LAX/MEL"}
```

Example 3: Select details of those bags which does not originate from MIA/pass through MIA.

```
SELECT fullName, bag.bagInfo[].tagNum, bag.bagInfo[].routing,
bag.bagInfo[].flightLegs[].fltRouteSrc
FROM BaggageInfo bag
WHERE NOT bag.bagInfo[].flightLegs[].fltRouteSrc=any "MIA"
```

Explanation: You fetch the details of passengers not originating from a particular source. To fetch these details, you are using the **NOT** operator here. You want to fetch details of bags which did not start/go through **MIA**.

Output:

```
{"fullName":"Kendal
Biddle","tagNum":"17657806296887","routing":"JFK/IST/
VIE","fltRouteSrc":"JFK"}
{"fullName":"Lucinda
Beckman","tagNum":"17657806240001","routing":"SFO/IST/ATH/
JTR","fltRouteSrc":"SFO"}
{"fullName":"Adelaide
Willard","tagNum":"17657806224224","routing":"GRU/ORD/
SEA","fltRouteSrc":"GRU"}
{"fullName":"Raymond
```

```
Griffin","tagNum":"17657806243578","routing":"MSQ/FRA/
HKG","fltRouteSrc":"MSQ"}
{"fullName":"Elane Lemons","tagNum":"1765780623244","routing":"MXP/CDG/SLC/
BZN","fltRouteSrc":"MXP"}
{"fullName":"Dierdre Amador","tagNum":"17657806240229","routing":"JFK/
MAD","fltRouteSrc":"JFK"}
{"fullName":"Henry Jenkins","tagNum":"17657806216554","routing":"SFO/ORD/
FRA","fltRouteSrc":"SFO"}
{"fullName":"Rosalia Triplett","tagNum":"17657806215913","routing":"JFK/IST/
VIE","fltRouteSrc":"JFK"}
{"fullName":"Lorenzo Phil","tagNum":
["17657806240001","17657806340001"],"routing":["SFO/IST/ATH/
JTR","SFO/IST/ATH/JTR"],"fltRouteSrc":["SFO","SFO"]}
{"fullName":"Gerard Greene","tagNum":"1765780626568","routing":"SFO/IST/ATH/
JTR","fltRouteSrc":"SFO"}
{"fullName":"Doris Martin","tagNum":"17657806232501","routing":"BZN/SEA/CDG/
MXP","fltRouteSrc":"BZN"}
{"fullName":"Omar Harvey","tagNum":"17657806234185","routing":"MEL/LAX/
MIA","fltRouteSrc":"MEL"}
{"fullName":"Fallon
Clements","tagNum":"17657806255507","routing":"MXP/CDG/SLC/
BZN","fltRouteSrc":"MXP"}
{"fullName":"Lisbeth Wampler","tagNum":"17657806292229","routing":"LAX/TPE/
SGN","fltRouteSrc":"LAX"}
{"fullName":"Teena Colley","tagNum":"17657806255823","routing":"MSQ/FRA/
HKG","fltRouteSrc":"MSQ"}
{"fullName":"Michelle
Payne","tagNum":"17657806247861","routing":"SFO/IST/ATH/
JTR","fltRouteSrc":"SFO"}
{"fullName":"Mary Watson","tagNum":"17657806299833","routing":"YYZ/HKG/
BLR","fltRouteSrc":"YYZ"}
```


NULL operators

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

Example 1: Fetch ticket number of passengers whose baggage details are available and is NOT NULL.

```
SELECT ticketNo,fullname FROM BaggageInfo bagdet
WHERE bagdet.bagInfo is NOT NULL
```

Explanation: You fetch the details of passengers who have baggage, which means `bagInfo` JSON is not null.

Output:


```
{"ticketNo":1762357254392,"fullname":"Teena Colley"}
{"ticketNo":1762330498104,"fullname":"Michelle Payne"}
{"ticketNo":1762340683564,"fullname":"Mary Watson"}
```

```
{"ticketNo":1762377974281,"fullname":"Kendal Biddle"}
{"ticketNo":1762320569757,"fullname":"Lucinda Beckman"}
{"ticketNo":1762392135540,"fullname":"Adelaide Willard"}
{"ticketNo":1762399766476,"fullname":"Raymond Griffin"}
{"ticketNo":1762324912391,"fullname":"Elane Lemons"}
{"ticketNo":1762390789239,"fullname":"Zina Christenson"}
{"ticketNo":1762340579411,"fullname":"Zulema Martindale"}
{"ticketNo":1762376407826,"fullname":"Dierdre Amador"}
{"ticketNo":176234463813,"fullname":"Henry Jenkins"}
{"ticketNo":1762311547917,"fullname":"Rosalia Triplett"}
{"ticketNo":1762320369957,"fullname":"Lorenzo Phil"}
{"ticketNo":1762341772625,"fullname":"Gerard Greene"}
{"ticketNo":1762344493810,"fullname":"Adam Phillips"}
{"ticketNo":1762355527825,"fullname":"Doris Martin"}
{"ticketNo":1762383911861,"fullname":"Joanne Diaz"}
{"ticketNo":1762348904343,"fullname":"Omar Harvey"}
{"ticketNo":1762350390409,"fullname":"Fallon Clements"}
{"ticketNo":1762355854464,"fullname":"Lisbeth Wampler"}
```

Example 2: Fetch ticket number of passengers whose baggage details are not available or IS NULL

```
SELECT ticketNo, fullname FROM BaggageInfo bagdet
WHERE bagdet.bagInfo is NULL
0 row returned
```

Value Comparison Operators

Value comparison operators are primarily used to compare 2 values, one produced by the left operand and another from the right operand. If any operand returns more than one item, an error is raised. If both operands return the empty sequence, the operands are considered equal (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 !=. If an operand returns NULL, the result of the comparison expression is also NULL. Otherwise, the result is a boolean value.

Example 1: Select the full name and routing of all male passengers.

```
SELECT fullname, bag.bagInfo[].routing
FROM BaggageInfo bag
WHERE gender="M"
```

Explanation: Here the data is filtered based on gender. The value comparison operator "=" is used to filter the data.

Output:

```
{"fullname":"Lucinda Beckman","routing":"SFO/IST/ATH/JTR"}
{"fullname":"Adelaide Willard","routing":"GRU/ORD/SEA"}
{"fullname":"Raymond Griffin","routing":"MSQ/FRA/HKG"}
{"fullname":"Zina Christenson","routing":"MIA/LAX/MEL"}
{"fullname":"Dierdre Amador","routing":"JFK/MAD"}
{"fullname":"Birgit Naquin","routing":"JFK/MAD"}
```

```
{"fullname":"Lorenzo Phil","routing":["SFO/IST/ATH/JTR","SFO/IST/ATH/JTR"]}
{"fullname":"Gerard Greene","routing":"SFO/IST/ATH/JTR"}
{"fullname":"Adam Phillips","routing":"MIA/LAX/MEL"}
{"fullname":"Fallon Clements","routing":"MXP/CDG/SLC/BZN"}
{"fullname":"Lisbeth Wampler","routing":"LAX/TPE/SGN"}
{"fullname":"Teena Colley","routing":"MSQ/FRA/HKG"}
```

You can rewrite this query with a "!=" comparison operator. To get the details of all male passengers, your query can filter data where gender is not "F". This is valid only with the assumption that there can only be two values in the column gender which is "F" and "M".

```
SELECT fullname,bag.bagInfo[].routing
FROM BaggageInfo bag
WHERE gender!="F";
```

Example 2: Fetch the passenger name and routing details of passengers with ticket numbers greater than 1762360000000.

```
SELECT fullname, ticketNo,
bag.bagInfo[].tagNum,bag.bagInfo[].routing
FROM BaggageInfo bag
WHERE ticketNo > 1762360000000
```

Explanation: You need the details of passengers whose ticket number is greater than the given value. You use the ">" operator to filter the data.

Output:

```
{"fullname":"Adelaide
Willard","ticketNo":1762392135540,"tagNum":"17657806224224","routing":"GRU/OR
D/SEA"}
{"fullname":"Raymond
Griffin","ticketNo":1762399766476,"tagNum":17657806243578,"routing":"MSQ/FRA/
HKG"}
{"fullname":"Zina
Christenson","ticketNo":1762390789239,"tagNum":"17657806228676","routing":"MI
A/LAX/MEL"}
{"fullname":"Bonnie
Williams","ticketNo":1762397286805,"tagNum":"17657806216554","routing":"SFO/O
RD/FRA"}
{"fullname":"Joanne
Diaz","ticketNo":1762383911861,"tagNum":"17657806292518","routing":"MIA/LAX/
MEL"}
{"fullname":"Kendal
Biddle","ticketNo":1762377974281,"tagNum":"17657806296887","routing":"JFK/IST
/VIE"}
{"fullname":"Dierdre
Amador","ticketNo":1762376407826,"tagNum":"17657806240229","routing":"JFK/
MAD"}
{"fullname":"Birgit
Naquin","ticketNo":1762392196147,"tagNum":"17657806240229","routing":"JFK/
MAD"}
```

Example 3: Select all bag tag numbers originating from SFO/transit through SFO.

```
SELECT bag.bagInfo[].tagNum,
bag.bagInfo[].flightLegs[].fltRouteSrc
FROM BaggageInfo bag
WHERE bag.bagInfo[].flightLegs[].fltRouteSrc=any "SFO"
```

Explanation: You fetch the tag number of bags that either originate from SFO or pass through SFO. Though you are using the value comparison operator **=**, since the `flightLegs` is an array, the left operand of comparison operator **=** is a sequence with more than one item. That is the reason to use the sequence operator **any** in addition to the value comparison operator **=**. Else you get the following error.

```
Error handling command SELECT
bag.bagInfo[].tagNum,bag.bagInfo[].flightLegs[].fltRouteSrc
FROM BaggageInfo bag WHERE bag.bagInfo[].flightLegs[].fltRouteSrc=
"SFO":
Error: at (3, 6) The left operand of comparison operator = is a
sequence with more than one items.
Comparison operators cannot operate on sequences of more than one
items.
```

Output:

```
{"tagNum":"17657806240001","fltRouteSrc":"SFO"}
{"tagNum":"17657806216554","fltRouteSrc":"SFO"}
{"tagNum":["17657806240001","17657806340001"],"fltRouteSrc":
["SFO","SFO"]}
{"tagNum":"1765780626568","fltRouteSrc":"SFO"}
{"tagNum":"17657806247861","fltRouteSrc":"SFO"}
```

Example 4: Select all bag tag numbers which did not originate from JFK.

```
SELECT bag.bagInfo[].tagNum,
bag.bagInfo[].flightLegs[0].fltRouteSrc
FROM BaggageInfo bag
WHERE bag.bagInfo.flightLegs[0].fltRouteSrc!=ANY "JFK"
```

Explanation: The assumption here is that the first record of the `flightLegs` array has the details of the source location. You fetch the tag number of bags that did not originate from JFK and so using a **!=** operator here. Though you are using the value comparison operator **!=**, since the `flightLegs` is an array, the left operand of the comparison operator **!=** is a sequence with more than one item. That is the reason to use the sequence operator **any** in addition to the value comparison operator **!=**. Else you get the following error.

```
Error handling command SELECT
bag.bagInfo[].tagNum,bag.bagInfo[].flightLegs[0].fltRouteSrc
FROM BaggageInfo bag WHERE bag.bagInfo.flightLegs[0].fltRouteSrc!
="JFK":
Failed to display result set: Error: at (2, 0) The left operand of
comparison operator != is a sequence with
```

more than one items. Comparison operators cannot operate on sequences of more than one items.

Output:

```
{"tagNum":"17657806240001","fltRouteSrc":["SFO","IST","ATH"]}
{"tagNum":"17657806224224","fltRouteSrc":["GRU","ORD"]}
{"tagNum":"17657806243578","fltRouteSrc":["MSQ","FRA"]}
{"tagNum":"1765780623244","fltRouteSrc":["MXP","CDG","SLC"]}
{"tagNum":"17657806228676","fltRouteSrc":["MIA","LAX"]}
{"tagNum":"17657806234185","fltRouteSrc":["MEL","LAX"]}
{"tagNum":"17657806255507","fltRouteSrc":["MXP","CDG","SLC"]}
{"tagNum":"17657806292229","fltRouteSrc":["LAX","TPE"]}
{"tagNum":"17657806255823","fltRouteSrc":["MSQ","FRA"]}
{"tagNum":"17657806247861","fltRouteSrc":["SFO","IST","ATH"]}
{"tagNum":"17657806299833","fltRouteSrc":["YYZ","HKG"]}
{"tagNum":"17657806288937","fltRouteSrc":["MIA","LAX"]}
{"tagNum":"17657806216554","fltRouteSrc":["SFO","ORD"]}
{"tagNum":["17657806240001","17657806340001"],"fltRouteSrc":
["SFO","IST","ATH","SFO","IST","ATH"]}
{"tagNum":"1765780626568","fltRouteSrc":["SFO","IST","ATH"]}
{"tagNum":"17657806255240","fltRouteSrc":["MIA","LAX"]}
{"tagNum":"17657806232501","fltRouteSrc":["BZN","SEA","CDG"]}
{"tagNum":"17657806292518","fltRouteSrc":["MIA","LAX"]}
```

IN Operator

The IN operator is essentially a compact alternative to a number of OR-ed equality conditions. This operator allows you to specify multiple values in a WHERE clause.

Example: Fetch tag number for the customers `"Lucinda Beckman", "Adam Phillips","Zina Christenson","Fallon Clements"`.

```
SELECT bagdet.fullName, bagdet.bagInfo[].tagNum
FROM BaggageInfo bagdet
WHERE bagdet.fullName IN
("Lucinda Beckman", "Adam Phillips","Zina Christenson","Fallon Clements")
```

Explanation: You fetch the tag numbers of a list of passengers. The list of passengers to be fetched can be given inside an **IN** clause.

Output:

```
{"fullName":"Lucinda Beckman","tagNum":"17657806240001"}
{"fullName":"Zina Christenson","tagNum":"17657806228676"}
{"fullName":"Adam Phillips","tagNum":"17657806255240"}
{"fullName":"Fallon Clements","tagNum":"17657806255507"}
```

Regular Expression Conditions

A regular expression is a pattern that the regular expression engine attempts to match with an input string. The `regex_like` function performs regular expression matching. The

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

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

Example 1: Fetch baggage information of passengers whose names start with 'Z'.

```
SELECT bag.fullname,bag.bagInfo[].tagNum
FROM BaggageInfo bag
WHERE regex_like(fullName, "Z.*")
```

Explanation: You fetch the full name and tag numbers of passengers whose full name starts with Z. You use a regular expression and specify that the first character in the full name should be "**Z**" and the rest can be anything else.

Output:

```
{"fullname":"Zina Christenson","tagNum":"17657806228676"}
{"fullname":"Zulema Martindale","tagNum":"17657806288937"}
```

Example 2: Fetch baggage information of passengers whose flight source location has an "**M**" in it.

Option 1:

```
SELECT bag.fullname,bag.bagInfo[].tagNum,
bag.bagInfo[].flightLegs[0].fltRouteSrc
FROM BaggageInfo bag
WHERE regex_like(bag.bagInfo.flightLegs[0].fltRouteSrc, ".*M.*")
```

Explanation: The assumption here is that the first record of the `flightLegs` array has the details of the source location. You fetch the full name and tag numbers of passengers whose flight source has an "**M**" in it. You use a regular expression and specify that one of the characters in the source field should be "**M**" and the rest can be anything else.

You can also use different approaches to write queries to solve the above problem.

Option 2: Instead of hard coding the index of the `flightLegs` array, you use the `regex_like` function to determine the correct index.

```
SELECT bag.fullname,bag.bagInfo[].tagNum,
bag.bagInfo[].flightLegs[].fltRouteSrc
FROM BaggageInfo bag
WHERE EXISTS (bag.bagInfo.flightLegs[regex_like($element.fltRouteSrc,
".*M.*")])
```

Option 3: You use the substring of the "routing" field to extract the source and then use `regex_like` function to search the letter **M** in the source.

```
SELECT bag.fullname,bag.bagInfo[].tagNum,
substring(bag.bagInfo[].routing,0,3)
FROM BaggageInfo bag WHERE
regex_like(substring(bag.bagInfo[].routing,0,3), ".*M.*")
```

Output:

```
{"fullname":"Raymond Griffin","tagNum":"17657806243578","fltRouteSrc":"MSQ"}
{"fullname":"Elane Lemons","tagNum":"1765780623244","fltRouteSrc":"MXP"}
{"fullname":"Zina Christenson","tagNum":"17657806228676","fltRouteSrc":"MIA"}
{"fullname":"Zulema
Martindale","tagNum":"17657806288937","fltRouteSrc":"MIA"}
{"fullname":"Adam Phillips","tagNum":"17657806255240","fltRouteSrc":"MIA"}
{"fullname":"Joanne Diaz","tagNum":"17657806292518","fltRouteSrc":"MIA"}
{"fullname":"Teena Colley","tagNum":"17657806255823","fltRouteSrc":"MSQ"}
{"fullname":"Omar Harvey","tagNum":"17657806234185","fltRouteSrc":"MEL"}
{"fullname":"Fallon Clements","tagNum":"17657806255507","fltRouteSrc":"MXP"}
```

EXISTS Operator

The `EXISTS` operator checks whether the sequence returned by its input expression is empty or not, and returns false or true, respectively. A special case is when the input expression returns NULL. In this case, `EXISTS` will also return NULL.

Example 1: Select passenger details and baggage information for those passengers who have three flight segments.

```
SELECT fullName, bag.bagInfo[].tagNum,
bag.bagInfo[].routing
FROM BaggageInfo bag
WHERE EXISTS bag.bagInfo[].flightLegs[2]
```

Explanation: You fetch the details of the passengers who have three flight segments. You determine this by evaluating if the third element of the flight legs array is present using the `EXISTS` operator.

Output:

```
{"fullName":"Lorenzo Phil","tagNum":
["17657806240001","17657806340001"],"routing":["SFO/IST/ATH/
JTR","SFO/IST/ATH/JTR"]}
{"fullName":"Gerard Greene","tagNum":"1765780626568","routing":"SFO/IST/ATH/
JTR"}
{"fullName":"Doris Martin","tagNum":"17657806232501","routing":"BZN/SEA/CDG/
MXP"}
{"fullName":"Fallon
Clements","tagNum":"17657806255507","routing":"MXP/CDG/SLC/BZN"}
{"fullName":"Michelle
Payne","tagNum":"17657806247861","routing":"SFO/IST/ATH/JTR"}
{"fullName":"Lucinda
```

```
Beckman","tagNum":"17657806240001","routing":"SFO/IST/ATH/JTR"}
{"fullName":"Elane
Lemons","tagNum":"1765780623244","routing":"MXP/CDG/SLC/BZN"}
```

Example 2: Fetch the full name and tag number for all customer baggage shipped after 2019.

```
SELECT fullName, bag.ticketNo
FROM BaggageInfo bag WHERE
EXISTS bag.bagInfo[$element.bagArrivalDate >="2019-01-01T00:00:00"]
```

Explanation: The bag arrival date value for every bag should be greater than the year 2019. Here the **"$element"** is bound to the context row (every bag of the customer). The `EXISTS` operator checks whether the sequence returned by its input expression is empty or not. The sequence returned by the comparison operator ">=" is non-empty for all bags which arrived after 2019.

Output:

```
{"fullName":"Lucinda Beckman","ticketNo":1762320569757}
{"fullName":"Adelaide Willard","ticketNo":1762392135540}
{"fullName":"Raymond Griffin","ticketNo":1762399766476}
{"fullName":"Elane Lemons","ticketNo":1762324912391}
{"fullName":"Zina Christenson","ticketNo":1762390789239}
{"fullName":"Zulema Martindale","ticketNo":1762340579411}
{"fullName":"Dierdre Amador","ticketNo":1762376407826}
{"fullName":"Henry Jenkins","ticketNo":176234463813}
{"fullName":"Rosalia Triplett","ticketNo":1762311547917}
{"fullName":"Lorenzo Phil","ticketNo":1762320369957}
{"fullName":"Gerard Greene","ticketNo":1762341772625}
{"fullName":"Adam Phillips","ticketNo":1762344493810}
{"fullName":"Doris Martin","ticketNo":1762355527825}
{"fullName":"Joanne Diaz","ticketNo":1762383911861}
{"fullName":"Omar Harvey","ticketNo":1762348904343}
{"fullName":"Fallon Clements","ticketNo":1762350390409}
{"fullName":"Lisbeth Wampler","ticketNo":1762355854464}
{"fullName":"Teena Colley","ticketNo":1762357254392}
{"fullName":"Michelle Payne","ticketNo":1762330498104}
{"fullName":"Mary Watson","ticketNo":1762340683564}
{"fullName":"Kendal Biddle","ticketNo":1762377974281}
```

Is-Of-Type Operator

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.

Example: Fetch the names of the passengers whose baggage tags contain only numbers and not a STRING.

```
SELECT fullname,bag.bagInfo.tagNum
FROM  BaggageInfo bag
WHERE bag.bagInfo.tagNum is of type (NUMBER)
```

Explanation: The `tagNum` in the `bagInfo` schema is a STRING data type. But the application could take in a NUMBER value as `tagNum` by mistake. The query captures the passengers for whom the `tagNum` column has only numbers.

Output:

```
{"fullname":"Raymond Griffin","tagNum":17657806243578}
```

If you query the `bagInfo` schema for the above `tagNum` as STRING, no rows are displayed.

```
SELECT * FROM BaggageInfo bag WHERE tagnum = "17657806232501"
0 row returned
```

You can also fetch the names of the passengers whose baggage tags contain only STRING.

```
SELECT fullname,bag.bagInfo.tagNum
FROM BaggageInfo bag
WHERE  bag.bagInfo.tagNum is of type (STRING)
```

Sorting, Grouping & Limiting results

If you want to follow along with the examples, download the script baggageschema_loaddata.sql and execute it as shown below. This script creates the table used in the example and loads data into the table.

Start your KVSTORE or KVLite and open the SQL.shell.

```
java -jar lib/kvstore.jar kvlite -secure-config disable
java -jar lib/sql.jar -helper-hosts localhost:5000 -store kvstore
```

Using the `load` command, execute the script.

```
load -file baggageschema_loaddata.sql
```


Ordering results

Use the ORDER BY clause to order the results by any column, primary key or non-primary key.

Example 1: Sort the ticket number of all passengers by their full name.

```
SELECT bag.ticketNo, bag.fullName
FROM BaggageInfo bag
ORDER BY bag.fullName
```

Explanation: You are sorting the ticket number of passengers in the `BaggageInfo` schema based on the full name of the passengers in ascending order.

Output:

```
{"ticketNo":1762344493810,"fullName":"Adam Phillips"}
{"ticketNo":1762392135540,"fullName":"Adelaide Willard"}
{"ticketNo":1762376407826,"fullName":"Dierdre Amador"}
{"ticketNo":1762355527825,"fullName":"Doris Martin"}
{"ticketNo":1762324912391,"fullName":"Elane Lemons"}
{"ticketNo":1762350390409,"fullName":"Fallon Clements"}
{"ticketNo":1762341772625,"fullName":"Gerard Greene"}
{"ticketNo":176234463813,"fullName":"Henry Jenkins"}
{"ticketNo":1762383911861,"fullName":"Joanne Diaz"}
{"ticketNo":1762377974281,"fullName":"Kendal Biddle"}
{"ticketNo":1762355854464,"fullName":"Lisbeth Wampler"}
{"ticketNo":1762320369957,"fullName":"Lorenzo Phil"}
{"ticketNo":1762320569757,"fullName":"Lucinda Beckman"}
{"ticketNo":1762340683564,"fullName":"Mary Watson"}
{"ticketNo":1762330498104,"fullName":"Michelle Payne"}
{"ticketNo":1762348904343,"fullName":"Omar Harvey"}
{"ticketNo":1762399766476,"fullName":"Raymond Griffin"}
{"ticketNo":1762311547917,"fullName":"Rosalia Triplett"}
{"ticketNo":1762357254392,"fullName":"Teena Colley"}
{"ticketNo":1762390789239,"fullName":"Zina Christenson"}
{"ticketNo":1762340579411,"fullName":"Zulema Martindale"}
```

Example 2: Fetch the passenger details(full name, tag number) by the last seen time (latest first) for passengers (sorted by their name) whose last seen station is **MEL**.

```
SELECT bag.fullName, bag.bagInfo[].tagNum,
bag.bagInfo[].lastSeenTimeGmt
FROM BaggageInfo bag
WHERE bag.bagInfo[].lastSeenStation=any "MEL"
ORDER BY bag.bagInfo[].lastSeenTimeGmt DESC
```

Explanation: You first filter the data in the `BaggageInfo` table based on the last seen station and you sort the filtered results based on the last seen time and the full name of the passengers in descending order. You do this using the ORDER BY clause.

> **Note:**
>
> You can use more than one column to sort the output of the query.

Output:

```
{"fullName":"Adam
Phillips","tagNum":"17657806255240","lastSeenTimeGmt":"2019-02-01T16:13:00Z"}
{"fullName":"Zina
Christenson","tagNum":"17657806228676","lastSeenTimeGmt":"2019-02-04T10:08:00
Z"}
{"fullName":"Joanne
Diaz","tagNum":"17657806292518","lastSeenTimeGmt":"2019-02-16T16:13:00Z"}
{"fullName":"Zulema
Martindale","tagNum":"17657806288937","lastSeenTimeGmt":"2019-02-25T20:15:00Z
"}
```

Limit and offset results

Use the `LIMIT` clause to limit the number of results returned from a SELECT statement. For example, if there are 1000 rows in a table, limit the number of rows to return by specifying a LIMIT value. It is recommended to use LIMIT and OFFSET with an ORDER BY clause. Otherwise, the results are returned in a random order, producing unpredictable results.

A good use-case/example of using LIMIT and OFFSET is the application paging of results. Say for example your application wants to show 4 results per page. You can use limit and offset to implement stateless paging in the application. If you are showing n (say 4) results per page, then the results for page m (say 2) are being displayed, then offset would be (n*m-1) which is 4 in this example and the limit would be n(which is 4 here).

Example 1: Your application can show 4 results on a page. Fetch the details fetched by your application in the first page for passengers whose last seen station is **JTR**.

```
SELECT $bag.fullName, $bag.bagInfo.tagNum, $flt_time
FROM BaggageInfo $bag,
$bag.bagInfo[].lastSeenTimeGmt $flt_time
WHERE $bag.bagInfo[].lastSeenStation=any "JTR"
ORDER BY $flt_time LIMIT 4
```

Explanation: You filter the data in the `BaggageInfo` table based on the last seen station and you sort the result based on the last seen time. You use an unnest array to flatten your data. That is the `bagInfo` array is flattened and the last seen time is fetched. You need to just display the first 4 rows from the result set.

Output:

```
{"fullName":"Michelle
Payne","tagNum":"17657806247861","flt_time":"2019-02-02T23:59:00Z"}
{"fullName":"Gerard
Greene","tagNum":"1765780626568","flt_time":"2019-03-07T16:01:00Z"}
{"fullName":"Lorenzo Phil","tagNum":
["17657806240001","17657806340001"],"flt_time":"2019-03-12T15:05:00Z"}
{"fullName":"Lucinda
Beckman","tagNum":"17657806240001","flt_time":"2019-03-12T15:05:00Z"}
```

Example 2: Your application can show 4 results on a page. Fetch the details fetched by your application in the second page for passengers whose last seen station is **JTR**.

```
SELECT $bag.fullName, $bag.bagInfo.tagNum, $flt_time
FROM BaggageInfo $bag,
$bag.bagInfo[].lastSeenTimeGmt $flt_time
WHERE $bag.bagInfo[].lastSeenStation=any "JTR"
ORDER BY $flt_time LIMIT 4 OFFSET 4
```

Explanation: You filter the data in the `BaggageInfo` table based on the last seen station and you sort the result based on the last seen time. You use an unnest array to flatten your data. You need to display the contents of the second page, so you set an OFFSET 4. Though you LIMIT to 4 rows, only one row is displayed as the total result set is only 5. The first few are skipped and the fifth one is displayed.

Output:

```
{"fullName":"Lorenzo Phil","tagNum":
["17657806240001","17657806340001"],
"flt_time":"2019-03-12T16:05:00Z"}
```

Grouping results

Use the GROUP BY clause to group the results by one or more table columns. Typically, a GROUP BY clause is used in conjunction with an aggregate expression such as COUNT, SUM, and AVG.

Example 1: Display the number of bags for each reservation made.

```
SELECT bag.confNo,
count(bag.bagInfo) AS TOTAL_BAGS
FROM BaggageInfo bag
GROUP BY bag.confNo
```

Explanation: Every passenger has one reservation code (`confNo`). A passenger can have more than one baggage. Here you group the data based on the reservation code and you get the count of the `bagInfo` array which gives the number of bags per reservation.

Output:

```
{"confNo":"FH7G1W","TOTAL_BAGS":1}
{"confNo":"PQ1M8N","TOTAL_BAGS":1}
{"confNo":"XT6K7M","TOTAL_BAGS":1}
{"confNo":"DN3I4Q","TOTAL_BAGS":1}
{"confNo":"QB1O0J","TOTAL_BAGS":1}
{"confNo":"TX1P7E","TOTAL_BAGS":1}
{"confNo":"CG6O1M","TOTAL_BAGS":1}
{"confNo":"OH2F8U","TOTAL_BAGS":1}
{"confNo":"BO5G3H","TOTAL_BAGS":1}
{"confNo":"ZG8Z5N","TOTAL_BAGS":1}
{"confNo":"LE6J4Z","TOTAL_BAGS":1}
{"confNo":"XT1O7T","TOTAL_BAGS":1}
```

```
{"confNo":"QI3V6Q","TOTAL_BAGS":2}
{"confNo":"RL3J4Q","TOTAL_BAGS":1}
{"confNo":"HJ4J4P","TOTAL_BAGS":1}
{"confNo":"CR2C8MY","TOTAL_BAGS":1}
{"confNo":"LN0C8R","TOTAL_BAGS":1}
{"confNo":"MZ2S5R","TOTAL_BAGS":1}
{"confNo":"KN4D1L","TOTAL_BAGS":1}
{"confNo":"MC0E7R","TOTAL_BAGS":1}
```

Example 2: Select the total baggage originating from each airport (excluding the transit baggage).

```
SELECT $flt_src as SOURCE,
count(*) as COUNT
FROM BaggageInfo $bag,
$bag.bagInfo.flightLegs[0].fltRouteSrc $flt_src
GROUP BY $flt_src
```

Explanation: You want to get the total count of baggage originating from each airport. However, you don't want to consider the airports that are part of the transit. So you group the data with the flight source values of the first record of the `flightLegs` array(as the first record is the source). You then determine the count of baggage.

Output:

```
{"SOURCE":"SFO","COUNT":6}
{"SOURCE":"BZN","COUNT":1}
{"SOURCE":"GRU","COUNT":1}
{"SOURCE":"LAX","COUNT":1}
{"SOURCE":"YYZ","COUNT":1}
{"SOURCE":"MEL","COUNT":1}
{"SOURCE":"MIA","COUNT":4}
{"SOURCE":"MSQ","COUNT":2}
{"SOURCE":"MXP","COUNT":2}
{"SOURCE":"JFK","COUNT":3}
```

Primary Expressions in SQL

If you want to follow along with the examples, download the script baggageschema_loaddata.sql and execute it as shown below. This script creates the table used in the example and loads data into the table.

Start your KVSTORE or KVLite and open the SQL.shell.

```
java -jar lib/kvstore.jar kvlite -secure-config disable
java -jar lib/sql.jar -helper-hosts localhost:5000 -store kvstore
```

Using the `load` command, execute the script.

```
load -file baggageschema_loaddata.sql
```

- Parenthesized Expressions

Parenthesized Expressions

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.

Example: Fetch the full name, tag number, and routing details of passengers either boarding at JFK /traversing through JFK and their destination is either MAD or VIE.

```
SELECT fullName, bag.bagInfo.tagNum,
bag.bagInfo.routing,
bag.bagInfo[].flightLegs[].fltRouteDest
FROM BaggageInfo bag
WHERE bag.bagInfo.flightLegs[].fltRouteSrc=any "JFK" AND
(bag.bagInfo[].flightLegs[].fltRouteDest=any "MAD" OR
bag.bagInfo[].flightLegs[].fltRouteDest=any "VIE" )
```

Explanation: You want to fetch the full name, tag number, and routing details of passengers. The first filter condition is that the boarding point/transit is JFK. Once this is satisfied the second filter condition is that destination is either MAD or VIE. You use an OR condition to filter the destination value.

Output:

```
{"fullName":"Dierdre Amador","tagNum":"17657806240229","routing":"JFK/
MAD","fltRouteDest":"MAD"}
{"fullName":"Rosalia
Triplett","tagNum":"17657806215913","routing":"JFK/IST/
VIE","fltRouteDest":["IST","VIE"]}
{"fullName":"Kendal
Biddle","tagNum":"17657806296887","routing":"JFK/IST/
VIE","fltRouteDest":["IST","VIE"]}
```

Case Expressions

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

Example:

```
SELECT
    fullName,
    CASE
        WHEN NOT exists bag.bagInfo.flightLegs[0]
        THEN "you have no bag info"
        WHEN NOT exists bag.bagInfo.flightLegs[1]
        THEN "you have one hop"
        WHEN NOT exists bag.bagInfo.flightLegs[2]
        THEN "you have two hops."
        ELSE "you have three hops."
    END AS NUMBER_HOPS
FROM BaggageInfo bag WHERE ticketNo=1762340683564
```

Explanation: You want to determine how many transits are there for the passenger `bagInfo` using a CASE statement. If the `flightLegs` array has no elements, then the passenger has no bag data. When the `flightLegs` array has only one element, then there is only one transit point. Similarly, if the `flightLegs` array has two elements, then there is two hops. Else there is three transit points. Here you assume that a bag can have at the most three transit points/ hops.

Output:

```
{"fullName":"Mary Watson","NUMBER_HOPS":"you have two hops."}
```

Example 2: Write a query to alert the system to update the `tagNum` of passengers if the existing value is not a string.

```
SELECT bag.bagInfo[].tagNum,
CASE
   WHEN bag.bagInfo[0].tagNum is of type (NUMBER)
   THEN "Tagnumber is not a STRING. Update the data"
   ELSE "Tagnumber has correct datatype"
   END AS tag_NUM_TYPE
FROM BaggageInfo bag
```

Explanation: The `tagNum` of passengers in the `bagInfo` schema is a STRING data type. But the application could take in a NUMBER value as the value of `tagNum` by mistake. The query uses "is of type" operator to capture this and prompts the system to update the `tagNum` if the existing value is not a string.

Output (only few rows are shown for brevity).

```
{"tagNum":"17657806240001","tag_NUM_TYPE":"Tagnumber has correct datatype"}
{"tagNum":"17657806224224","tag_NUM_TYPE":"Tagnumber has correct datatype"}
{"tagNum":17657806243578,"tag_NUM_TYPE":"Tagnumber is not a STRING. Update
the data"}
{"tagNum":"1765780623244","tag_NUM_TYPE":"Tagnumber has correct datatype"}
```

Cast Expression

The cast expression creates, if possible, new items of a given target type from the items of its input sequence. For example, a STRING can be converted to TIMESTAMP(0) using CAST expression.

Rules followed in a CAST expression:

- If the type of the input item is equal to the target item type, the cast is a no-op: the input item itself is returned.
- If the target type is a wildcard type other than JSON and the type of the input item is a subtype of the wild card type, the cast is a no-op.
- If the target type is JSON, then an error is raised if the input item is a non-json atomic type.
- If the target type is an array type, an error is raised if the input item is not an array.
- If the target type is string, the input item may be of any type. That means every item can be cast to a string. For timestamps, their string value is in UTC and has the format `uuuu-MM-dd['T'HH:mm:ss]`.
- If the target type is an atomic type other than string, the input item must also be atomic.
  - * Integers and longs can be cast to timestamps. The input value is interpreted as the number of milliseconds since January 1, 1970, 00:00:00 GMT.
  - * String items may be castable to all other atomic types. Whether the cast succeeds or not depends on whether the actual string value can be parsed into a value that belongs to the domain of the target type.
  - * Timestamp items are castable to all the timestamp types. If the target type has a smaller precision that the input item, the resulting timestamp is the one closest to the input timestamp in the target precision.
- To cast a STRING to TIMESTAMP, if the input has STRING values in ISO-8601 format, then it will be automatically converted by the SQL runtime into TIMESTAMP data type.

> **Note:**
>
> ISO8601 describes an internationally accepted way to represent dates, times, and durations.
> **Syntax:** Date with time: YYYY-MM-DDThh:mm:ss[.s[s[s[s[s]]]]][Z|(+|-)hh:mm]
>
> where
>
> - YYYY specifies the year, as four decimal digits
> - MM specifies the month, as two decimal digits, 00 to 12
> - DD specifies the day, as two decimal digits, 00 to 31
> - hh specifies the hour, as two decimal digits, 00 to 23
> - mm specifies the minutes, as two decimal digits, 00 to 59
> - ss[.s[s[s[s[s]]]]] specifies the seconds, as two decimal digits, 00 to 59, optionally followed by a decimal point and 1 to 6 decimal digits (representing the fractional part of a second).
> - Z specifies UTC time (time zone 0). (It can also be specified by +00:00, but not by –00:00.)
> - (+|-)hh:mm specifies the time-zone as difference from UTC. (One of + or – is required.)

Example 1: Fetch the bag arrival date for the passenger with a reservation code **DN3I4Q** in TIMESTAMP(3) format.

```
SELECT CAST (bag.bagInfo.bagArrivalDate AS Timestamp(3))
AS BAG_ARRIVING_DATE
FROM BaggageInfo bag WHERE bag.confNo=DN3I4Q
```

Explanation: The `bagArrivalDate` is a STRING. Using CAST you are converting this field into a TIMESTAMP format.

Output:

```
{"BAG_ARRIVING_DATE":"2019-02-15T21:21:00.000Z"}
```

Example 2: Fetch the full name and tag number for all customer baggage shipped after 2019.

```
SELECT fullName, bag.ticketNo,
bag.bagInfo[].bagArrivalDate
FROM BaggageInfo bag WHERE
exists bag.bagInfo[$element.bagArrivalDate >="2019-01-01T00:00:00"]
```

Explanation: You want to filter and display details of the baggage that are shipped after 2019. The bag arrival date for every element in the `flightLegs` array is compared with the given timestamp (`2019-01-01T00:00:00`). Here the casting is implicit as `bagArrivalDate` is a STRING and is directly compared with a static Timestamp value. An explicit CAST function is not needed when an implicit casting can be done. However, your data should be in the format

`YYYY-MM-DDTHH:MI:SS` . You then use the EXISTS condition to check if the bagInfo is present for this timestamp condition.

Output:


```
{"fullName":"Kendal
Biddle","ticketNo":1762377974281,"bagArrivalDate":"2019-03-05T12:00:00Z
"}
{"fullName":"Lucinda
Beckman","ticketNo":1762320569757,"bagArrivalDate":"2019-03-12T15:05:00
Z"}
{"fullName":"Adelaide
Willard","ticketNo":1762392135540,"bagArrivalDate":"2019-02-15T21:21:00
Z"}
{"fullName":"Raymond
Griffin","ticketNo":1762399766476,"bagArrivalDate":"2019-02-03T08:09:00
Z"}
{"fullName":"Elane
Lemons","ticketNo":1762324912391,"bagArrivalDate":"2019-03-15T10:13:00Z
"}
{"fullName":"Zina
Christenson","ticketNo":1762390789239,"bagArrivalDate":"2019-02-04T10:0
8:00Z"}
{"fullName":"Zulema
Martindale","ticketNo":1762340579411,"bagArrivalDate":"2019-02-25T20:15
:00Z"}
{"fullName":"Dierdre
Amador","ticketNo":1762376407826,"bagArrivalDate":"2019-03-07T13:51:00Z
"}
{"fullName":"Henry
Jenkins","ticketNo":176234463813,"bagArrivalDate":"2019-03-02T13:18:00Z
"}
{"fullName":"Rosalia
Triplett","ticketNo":1762311547917,"bagArrivalDate":"2019-02-12T07:04:0
0Z"}
{"fullName":"Lorenzo Phil","ticketNo":1762320369957,"bagArrivalDate":
["2019-03-12T15:05:00Z","2019-03-12T16:25:00Z"]}
{"fullName":"Gerard
Greene","ticketNo":1762341772625,"bagArrivalDate":"2019-03-07T16:01:00Z
"}
{"fullName":"Adam
Phillips","ticketNo":1762344493810,"bagArrivalDate":"2019-02-01T16:13:0
0Z"}
{"fullName":"Doris
Martin","ticketNo":1762355527825,"bagArrivalDate":"2019-03-22T10:17:00Z
"}
{"fullName":"Joanne
Diaz","ticketNo":1762383911861,"bagArrivalDate":"2019-02-16T16:13:00Z"}
{"fullName":"Teena
Colley","ticketNo":1762357254392,"bagArrivalDate":"2019-02-13T11:15:00Z
"}
{"fullName":"Michelle
Payne","ticketNo":1762330498104,"bagArrivalDate":"2019-02-02T23:59:00Z"
}
{"fullName":"Mary
```

```
Watson","ticketNo":1762340683564,"bagArrivalDate":"2019-03-14T06:22:00Z"}
{"fullName":"Omar
Harvey","ticketNo":1762348904343,"bagArrivalDate":"2019-03-02T16:09:00Z"}
{"fullName":"Fallon
Clements","ticketNo":1762350390409,"bagArrivalDate":"2019-02-21T14:08:00Z"}
{"fullName":"Lisbeth
Wampler","ticketNo":1762355854464,"bagArrivalDate":"2019-02-10T10:01:00Z"}
```

Sequence Transform Expressions

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

Example: For each `ticketNo`, fetch a flat array containing all the actions performed on the luggage of that `ticketNo`.

```
SELECT seq_transform(l.bagInfo[],
      seq_transform(
          $sq1.flightLegs[],
          seq_transform(
              $sq2.actions[],
              {
                "at" : $sq3.actionAt,
                "action" : $sq3.actionCode,
                "flightNo" : $sq2.flightNo,
                "tagNum" : $sq1.tagNum
              }
          )
      )
  ) AS actions
FROM baggageInfo l WHERE ticketNo=1762340683564
```

Explanation: You can use the sequence transform expression for transforming JSON documents stored in table rows. In such cases, you often use multiple sequence transform expressions nested inside each other. Here the mapper expression of an inner sequence transform may need to access the current item of an outer sequence transform. To allow this, each sequence transform expression 'S' declares a variable with name `$sqN`, where N is the level of nesting of the expression `S` within the outer sequence transform expressions. `$sqN` is basically a synonym for `$`, that is, it is bound to the items returned by the input expression `S`. However, `$sqN` can be accessed by other sequence transform expressions that may be nested inside the expression `S`.

Output:

```
{
 "actions":[
   {"action":"ONLOAD to
HKG","at":"YYZ","flightNo":"BM267","tagNum":"17657806299833"},
```

```
   {"action":"BagTag Scan at
YYZ","at":"YYZ","flightNo":"BM267","tagNum":"17657806299833"},
   {"action":"Checkin at
YYZ","at":"YYZ","flightNo":"BM267","tagNum":"17657806299833"},
   {"action":"Offload to Carousel at
BLR","at":"BLR","flightNo":"BM115","tagNum":"17657806299833"},
   {"action":"ONLOAD to
BLR","at":"HKG","flightNo":"BM115","tagNum":"17657806299833"},
   {"action":"OFFLOAD from
HKG","at":"HKG","flightNo":"BM115","tagNum":"17657806299833"}
 ]
}
```

Timestamp functions

You can perform various arithmetic operations on Timestamp and Duration values.

If you want to follow along with the examples, download the script baggageschema_loaddata.sql and run it as shown below. This script creates the table used in the example and loads data into the table.

Start your KVSTORE or KVLite and open the SQL.shell.

```
java -jar lib/kvstore.jar kvlite -secure-config disable
java -jar lib/sql.jar -helper-hosts localhost:5000 -store kvstore
```

Using the `load` command, run the script.

```
load -file baggageschema_loaddata.sql
```


Extract Expressions

The EXTRACT expression extracts a component from a timestamp.

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

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 result 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, ISOWEEK.

Example 1: What is the full name and baggage arrival year for the customer with ticket number **1762383911861**.

```
SELECT fullName,
EXTRACT (YEAR FROM CAST (bag.bagInfo.bagArrivalDate AS Timestamp(0)))
```

```
AS YEAR FROM BaggageInfo bag
WHERE ticketNo=1762383911861
```

Explanation: You first use CAST to convert the `bagArrivalDate` to a TIMESTAMP and then fetch the YEAR component from the Timestamp.

Output:

```
{"fullName":"Joanne Diaz","YEAR":2019}
```

Example 2: Retrieve all bags that travelled through MIA between 10:00 am and 10:00 pm in February 2019.

```
SELECT bag.bagInfo[].tagNum,bag.bagInfo[].flightLegs[].fltRouteSrc,
$t1 AS HOUR FROM BaggageInfo bag,
EXTRACT(HOUR FROM CAST (bag.bagInfo[0].bagArrivalDate AS Timestamp(0))) $t1,
EXTRACT(YEAR FROM CAST (bag.bagInfo[0].bagArrivalDate AS Timestamp(0))) $t2,
EXTRACT(MONTH FROM CAST (bag.bagInfo[0].bagArrivalDate AS Timestamp(0))) $t3
WHERE bag.bagInfo[].flightLegs[].fltRouteSrc=any "MIA" AND
$t2=2019 AND $t3=02 AND ($t1>10 AND $t1<20)
```

Explanation: You want to know the details of flights that traveled through MIA between 10:00 am and 10:00 pm in February 2019. You use a number of filter conditions here. First, the flight should have originated or traversed through MIA. The year of arrival should be 2019 and the month of arrival should be 2 (February). Then you filter if the hour of arrival is between 10:00 am and 10:00 pm (20 hours).

Output:

```
{"tagNum":"17657806255240","fltRouteSrc":["MIA","LAX"],"HOUR":16}
{"tagNum":"17657806292518","fltRouteSrc":["MIA","LAX"],"HOUR":16}
```

Example 3: Which year and month did the passenger with the reservation code **PQ1M8N** receive the baggage?

```
SELECT fullName,
EXTRACT(YEAR FROM CAST (bag.bagInfo.bagArrivalDate AS Timestamp(0))) AS
YEAR,
EXTRACT(MONTH FROM CAST (bag.bagInfo.bagArrivalDate AS Timestamp(0))) AS
MONTH
FROM BaggageInfo bag WHERE bag.confNo="PQ1M8N"
```

Explanation: You first use CAST to convert the `bagArrivalDate` to a TIMESTAMP and then fetch the YEAR component and MONTH component from the Timestamp.

Output:

```
{"fullName":"Kendal Biddle","YEAR":2019,"MONTH":3}
```

Example 4: Group the baggage data based on the month of arrival and display the month and the number of baggage that arrived that month.

```
SELECT EXTRACT(MONTH FROM CAST ($bag_arr_date AS Timestamp(0))) AS
MONTH,
count(EXTRACT(MONTH FROM CAST ($bag_arr_date AS Timestamp(0)))) AS
COUNT
FROM BaggageInfo $bag, $bag.bagInfo[].bagArrivalDate $bag_arr_date
GROUP BY EXTRACT(MONTH FROM CAST ($bag_arr_date AS Timestamp(0)))
```

Explanation: You want to group the data based on the month of the arrival of baggage. You use an unnest array to flatten the data. The `bagInfo` array is flattened and the value of bag arrival date is fetched from the array. You then use CAST to convert the `bagArrivalDate` to a TIMESTAMP and then fetch the YEAR component and MONTH component from the Timestamp. You then use the `count` function to get the total baggage corresponding to every month.

> **Note:**
>
> One assumption in the data is that all the baggage has arrived in the same year. So you group the data only based on the month.

Output:

```
{"MONTH":2,"COUNT":11}
{"MONTH":3,"COUNT":10}
```

timestamp_add() function

Adds a duration to a timestamp value and returns the new timestamp. The duration can be positive or negative. The result type is `TIMESTAMP(9)`.

Syntax:

```
TIMESTAMP(9) timestamp_add(TIMESTAMP timestamp, STRING duration)
```

Semantics:

- **timestamp**: A TIMESTAMP value or a value that can be cast to TIMESTAMP.
- **duration**: A string with format [-](<n> <UNIT>)+, where 'n' is a number and the <UNIT> can be YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, MILLISECOND, NANOSECOND or the plural form of these keywords (e.g. YEARS).

  > **Note:**
  >
  > The UNIT keyword is case-insensitive.

- **returnvalue:** TIMESTAMP(9)

Example 1: In the airline application, a buffer of five minutes delay is considered "on time" . Print the estimated arrival time on the first leg with a buffer of five minutes for the passenger with ticket number **1762399766476**.

```
SELECT timestamp_add(bag.bagInfo.flightLegs[0].estimatedArrival, "5 minutes")
AS ARRIVAL_TIME FROM BaggageInfo bag
WHERE ticketNo=1762399766476
```

Explanation : In the airline application, a customer can have any number of flight legs depending on the source and destination. In the query above, you are fetching the estimated arrival in the "first leg" of the travel. So the first record of the `flightsLeg` array is fetched and the `estimatedArrival` time is fetched from the array and a buffer of "5 minutes" is added to that and displayed.

Output:

```
{"ARRIVAL_TIME":"2019-02-03T06:05:00.000000000Z"}
```

Note:

The column `estimatedArrival` is a STRING. If the column has STRING values in ISO-8601 format, then it will be automatically converted by the SQL runtime into TIMESTAMP data type.

ISO8601 describes an internationally accepted way to represent dates, times, and durations.

Syntax: Date with time: YYYY-MM-DDThh:mm:ss[.s[s[s[s[s[s]]]]]][Z|(+|-)hh:mm]

where

- YYYY specifies the year, as four decimal digits
- MM specifies the month, as two decimal digits, 00 to 12
- DD specifies the day, as two decimal digits, 00 to 31
- hh specifies the hour, as two decimal digits, 00 to 23
- mm specifies the minutes, as two decimal digits, 00 to 59
- ss[.s[s[s[s[s]]]]] specifies the seconds, as two decimal digits, 00 to 59, optionally followed by a decimal point and 1 to 6 decimal digits (representing the fractional part of a second).
- Z specifies UTC time (time zone 0). (It can also be specified by +00:00, but not by –00:00.)
- (+|-)hh:mm specifies the time-zone as difference from UTC. (One of + or – is required.)

Example 1a: Print the estimated arrival time in every leg with a buffer of five minutes for the passenger with ticket number **1762399766476**.

```
SELECT $s.ticketno, $value as estimate,
timestamp_add($value, '5 minute') AS add5min
```

```
FROM baggageinfo $s,
$s.bagInfo.flightLegs.estimatedArrival as $value
WHERE ticketNo=1762399766476
```

Explanation: You want to display the `estimatedArrival` time on every leg. The number of legs can be different for every customer. So variable reference is used in the query above and the `baggageInfo` array and the `flightLegs` array are unnested to execute the query.

Output:

```
{"ticketno":1762399766476,"estimate":"2019-02-03T06:00:00Z",
"add5min":"2019-02-03T06:05:00.000000000Z"}
{"ticketno":1762399766476,"estimate":"2019-02-03T08:22:00Z",
"add5min":"2019-02-03T08:27:00.000000000Z"}
```

Example 2 : How many bags arrived in the last week?

```
SELECT count(*) AS COUNT_LASTWEEK FROM baggageInfo bag
WHERE EXISTS bag.bagInfo[$element.bagArrivalDate < current_time()
AND $element.bagArrivalDate > timestamp_add(current_time(), "-7 days")]
```

Explanation: You get a count of the number of bags processed by the airline application in the last week. A customer can have more than one bag(that is `bagInfo` array can have more than one record). The`bagArrivalDate` should have a value between today and the last 7 days. For every record in the `bagInfo` array, you determine if the bag arrival time is between the time now and one week ago. The function `current_time` gives you the time now. An EXISTS condition is used as a filter for determining if the bag has an arrival date in the last week. The `count` function determines the total number of bags in this time period.

Output:

```
{"COUNT_LASTWEEK":0}
```

Example 3: Find the number of bags arriving in the next 6 hours.

```
SELECT count(*) AS COUNT_NEXT6HOURS FROM baggageInfo bag
WHERE EXISTS bag.bagInfo[$element.bagArrivalDate > current_time()
AND $element.bagArrivalDate < timestamp_add(current_time(), "6 hours")]
```

Explanation: You get a count of the number of bags that will be processed by the airline application in the next 6 hours. A customer can have more than one bag(that is`bagInfo` array can have more than one record). The `bagArrivalDate` should be between the time now and the next 6 hours. For every record in the `bagInfo` array, you determine if the bag arrival time is between the time now and six hours later. The function `current_time` gives you the time now. An EXISTS condition is used as a filter for determining if the bag has an arrival date in the next six hours. The `count` function determines the total number of bags in this time period.

Output:

```
{"COUNT_NEXT6HOURS":0}
```

timestamp_diff() and get_duration() functions

timestamp_diff()

Returns the number of milliseconds between two timestamp values. The result type is `LONG`.

Syntax:

```
LONG timestamp_diff(TIMESTAMP timestamp1, TIMESTAMP
      timestamp2)
```

Semantics:

- **timestamp1**: A TIMESTAMP value or a value that can be cast to TIMESTAMP
- **timestamp2**: A TIMESTAMP value or a value that can be cast to TIMESTAMP
- **returnvalue:** LONG

get_duration()

Converts the given number of milliseconds to a duration string. The result type is `STRING`.

Syntax:

```
STRING get_duration(LONG duration_millis)
```

Semantics:

- **duration_millis:** the duration in milliseconds
- **returnvalue:** STRING. The returned duration string format is [-](<n> <UNIT>)+, where the <UNIT> can be DAY, HOUR, MINUTE, SECOND and MILLISECOND, e.g. "1 day 2 hours" or "-10 minutes 0 second 500 milliseconds".

Examples:

Example 1: What is the duration between the time the baggage was boarded at one leg and reached the next leg for the passenger with ticket number **1762355527825**?

```
SELECT $s.ticketno, $bagInfo.bagArrivalDate, $flightLeg.flightDate,
get_duration(timestamp_diff($bagInfo.bagArrivalDate, $flightLeg.flightDate))
AS diff
FROM baggageinfo $s,
$s.bagInfo[] AS $bagInfo, $bagInfo.flightLegs[] AS $flightLeg
WHERE ticketNo=1762355527825
```

Explanation: In an airline application every customer can have a different number of hops/ legs between their source and destination. In this query, you determine the time taken between every flight leg. This is determined by the difference between `bagArrivalDate` and

`flightDate` for every flight leg. To determine the duration in days or hours or minutes, pass the result of the `timestamp_diff` function to the `get_duration` function.

Output:

```
{"bagArrivalDate":"2019-03-22T10:17:00Z","flightDate":"2019-03-22T07:00
:00Z",
"diff":"3 hours 17 minutes"}
{"bagArrivalDate":"2019-03-22T10:17:00Z","flightDate":"2019-03-22T07:23
:00Z",
"diff":"2 hours 54 minutes"}
{"bagArrivalDate":"2019-03-22T10:17:00Z","flightDate":"2019-03-22T08:23
:00Z",
"diff":"1 hour 54 minutes"}
```

To determine the duration in milliseconds, use only the `timestamp_diff` function.

```
SELECT $s.ticketno, $bagInfo.bagArrivalDate, $flightLeg.flightDate,
timestamp_diff($bagInfo.bagArrivalDate, $flightLeg.flightDate) AS diff
FROM baggageinfo $s,
$s.bagInfo[] AS $bagInfo,
$bagInfo.flightLegs[] AS $flightLeg
WHERE ticketNo=1762355527825
```

Example 2: How long does it take from the time of check-in to the time the bag is scanned at the point of boarding for the passenger with ticket number **176234463813**?

```
SELECT $flightLeg.flightNo,
$flightLeg.actions[contains($element.actionCode,
"Checkin")].actionTime AS checkinTime,
$flightLeg.actions[contains($element.actionCode, "BagTag
Scan")].actionTime AS bagScanTime,
get_duration(timestamp_diff(
    $flightLeg.actions[contains($element.actionCode,
"Checkin")].actionTime,
    $flightLeg.actions[contains($element.actionCode, "BagTag
Scan")].actionTime
)) AS diff
FROM baggageinfo $s,
$s.bagInfo[].flightLegs[] AS $flightLeg
WHERE ticketNo=176234463813 AND
starts_with($s.bagInfo[].routing, $flightLeg.fltRouteSrc)
```

Explanation: In the baggage data, every `flightLeg` has an actions array. There are three different actions in the action array. The action code for the first element in the array is Checkin/Offload. For the first leg, the action code is Checkin and for the other legs, the action code is Offload at the hop. The action code for the second element of the array is BagTag Scan. In the query above, you determine the difference in action time between the bag tag scan and check-in time. You use the `contains` function to filter the action time only if the action code is Checkin or BagScan. Since only the first flight leg has details of check-in and bag scan, you additionally filter the data using `starts_with` function to fetch only the source code `fltRouteSrc`. To determine the

duration in days or hours or minutes, pass the result of the `timestamp_diff` function to the `get_duration` function.

To determine the duration in milliseconds, only use the `timestamp_diff` function.

```
SELECT $flightLeg.flightNo,
$flightLeg.actions[contains($element.actionCode, "Checkin")].actionTime AS
checkinTime,
$flightLeg.actions[contains($element.actionCode, "BagTag Scan")].actionTime
AS bagScanTime,
timestamp_diff(
   $flightLeg.actions[contains($element.actionCode, "Checkin")].actionTime,
   $flightLeg.actions[contains($element.actionCode, "BagTag
Scan")].actionTime
) AS diff
FROM baggageinfo $s,
$s.bagInfo[].flightLegs[] AS $flightLeg
WHERE ticketNo=176234463813 AND
starts_with($s.bagInfo[].routing, $flightLeg.fltRouteSrc)
```

Output:

```
{"flightNo":"BM572","checkinTime":"2019-03-02T03:28:00Z",
"bagScanTime":"2019-03-02T04:52:00Z","diff":"- 1 hour 24 minutes"}
```

Example 3: How long does it take for the bags of a customer with ticket no **1762320369957** to reach the first transit point?

```
SELECT  $bagInfo.flightLegs[1].actions[2].actionTime,
$bagInfo.flightLegs[0].actions[0].actionTime,
get_duration(timestamp_diff($bagInfo.flightLegs[1].actions[2].actionTime,
                            $bagInfo.flightLegs[0].actions[0].actionTime))
AS diff
FROM baggageinfo $s, $s.bagInfo[] AS $bagInfo
WHERE ticketNo=1762320369957
```

Explanation: In an airline application every customer can have a different number of hops/ legs between their source and destination. In the example above, you determine the time taken for the bag to reach the first transit point. In the baggage data, the `flightLeg` is an array. The first record in the array refers to the first transit point details. The `flightDate` in the first record is the time when the bag leaves the source and the `estimatedArrival` in the first flight leg record indicates the time it reaches the first transit point. The difference between the two gives the time taken for the bag to reach the first transit point. To determine the duration in days or hours or minutes, pass the result of the `timestamp_diff` function to the `get_duration` function.

To determine the duration in milliseconds, use the `timestamp_diff` function.

```
SELECT  $bagInfo.flightLegs[0].flightDate,
$bagInfo.flightLegs[0].estimatedArrival,
timestamp_diff($bagInfo.flightLegs[0].estimatedArrival,
             $bagInfo.flightLegs[0].flightDate) AS diff
```

```
FROM baggageinfo $s, $s.bagInfo[] AS $bagInfo
WHERE ticketNo=1762320369957
```

Output:

```
{"flightDate":"2019-03-12T03:00:00Z","estimatedArrival":"2019-03-12T16:
00:00Z","diff":"13 hours"}
{"flightDate":"2019-03-12T03:00:00Z","estimatedArrival":"2019-03-12T16:
40:00Z","diff":"13 hours 40 minutes"}
```

Functions on Strings

There are various built-in functions on strings. In any string, position starts at 0 and ends at length - 1.

If you want to follow along with the examples, download the script baggageschema_loaddata.sql and execute it as shown below. This script creates the table used in the example and loads data into the table.

Start your KVSTORE or KVLite and open the SQL.shell.

```
java -jar lib/kvstore.jar kvlite -secure-config disable
java -jar lib/sql.jar -helper-hosts localhost:5000 -store kvstore
```

Using the `load` command, execute the script.

```
load -file baggageschema_loaddata.sql
```


substring function

The `substring` function extracts a string from a given string according to a given numeric starting position and a given numeric substring length.

```
returnvalue substring (source, position [, substring_length] )

source ::= any*
```

```
position ::= integer*
substring_length ::= integer*
returnvalue ::= string
```

Example: Fetch the first three characters from the routing details of a passenger with ticket number **1762376407826**.

```
SELECT substring(bag.baginfo.routing,0,3) AS Source
FROM baggageInfo bag
WHERE ticketNo=1762376407826
```

Output:

```
{"Source":"JFK"}
```

concat function

The `concat` function concatenates all its arguments and displays the concatenated string as output.

```
returnvalue concat (source,[source*])
source ::= any*
returnvalue ::= string
```

Example: Display the routing of a customer with a particular ticket number as "The route for passenger_name is ...".

```
SELECT concat("The route for passenger ",fullName , " is ",
bag.baginfo[0].routing)
FROM baggageInfo bag
WHERE ticketNo=1762376407826
```

Output:

```
{"Column_1":"The route for passenger Dierdre Amador is JFK/MAD"}
```

upper and lower functions

The `upper` and `lower` are simple functions to convert to fully upper case or lower case respectively. The `upper` function converts all the characters in a string to uppercase. The`lower` function converts all the characters in a string to lowercase.

```
returnvalue upper (source)
returnvalue lower (source)

source ::= any*
returnvalue ::= string
```

Example 1: Fetch the full name of the passenger in uppercase whose ticket number is **1762376407826**.

```
SELECT upper(fullname) AS FULLNAME_CAPITALS
FROM BaggageInfo
WHERE ticketNo=1762376407826
```

Output:

```
{"FULLNAME_CAPITALS":"DIERDRE AMADOR"}
```

Example 2: Fetch the full name of the passenger in lowercase whose ticket number is **1762376407826**.

```
SELECT lower(fullname) AS fullname_lowercase
FROM BaggageInfo WHERE ticketNo=1762376407826
```

Output:

```
{"fullname_lowercase":"dierdre amador"}
```

trim function

The `trim` function enables you to trim leading or trailing characters (or both) from a string. The `ltrim` function enables you to trim leading characters from a string. The `rtrim` function enables you to trim trailing characters from a string.

```
returnvalue trim(source [, position [, trim_character]])

source ::= any*
position ::= "leading"|"trailing"|"both"
trim_character ::= string*
returnvalue ::= string


returnvalue ltrim(source)

returnvalue rtrim(source)
source ::= any*
returnvalue ::= string
```

Example: Remove leading and trailing blank spaces from the route details of the passenger whose ticket number is **1762350390409**.

```
SELECT trim(bag.baginfo[0].routing,"trailing"," ")
FROM BaggageInfo bag
WHERE ticketNo=1762376407826
```

Output:

```
{"Column_1":"JFK/MAD"}
```

Using `ltrim` function to remove leading spaces:

```
SELECT ltrim(bag.baginfo[0].routing)
FROM BaggageInfo bag
WHERE ticketNo=1762376407826
```

Output:

```
{"Column_1":"JFK/MAD"}
```

Using `rtrim` function to remove trailing spaces:

```
SELECT rtrim(bag.baginfo[0].routing)
FROM BaggageInfo bag
WHERE ticketNo=1762376407826
```

Output:

```
{"Column_1":"JFK/MAD"}
```

length function

The `length` function returns the length of a character string. The length function calculates the length using the UTF character set.

```
returnvalue length(source)

source ::= any*
returnvalue ::= integer
```

Example: Find the length of the full name of the passenger whose ticket number is **1762350390409**.

```
SELECT fullname, length(fullname) AS fullname_length
FROM BaggageInfo
WHERE ticketNo=1762350390409
```

Output:

```
{"fullname":"Fallon Clements","fullname_length":15}
```

contains function

The `contains` function indicates whether or not a search string is present inside the source string.

```
returnvalue contains(source, search_string)

source ::= any*
search_string ::= any*
returnvalue ::= boolean
```

Example: Fetch the full names of passengers who have "SFO" in their route.

```
SELECT fullname FROM baggageInfo bag
WHERE EXISTS bag.bagInfo[contains($element.routing,"SFO")]
```

Output:

```
{"fullname":"Michelle Payne"}
{"fullname":"Lucinda Beckman"}
{"fullname":"Henry Jenkins"}
{"fullname":"Lorenzo Phil"}
{"fullname":"Gerard Greene"}
```

starts_with and ends_with functions

The `starts_with` function indicates whether or not the source string begins with the search string.

```
returnvalue starts_with(source, search_string)

source ::= any*
search_string ::= any*
returnvalue ::= boolean
```

The `ends_with`function indicates whether or not the source string ends with the search string.

```
returnvalue ends_with(source, search_string)

source ::= any*
search_string ::= any*
returnvalue ::= boolean
```

Example: How long does it take from the time of check-in to the time the bag is scanned at the point of boarding for the passenger with ticket number **176234463813**?

```
SELECT $flightLeg.flightNo,
$flightLeg.actions[contains($element.actionCode,
```

```
"Checkin")].actionTime AS checkinTime,
$flightLeg.actions[contains($element.actionCode, "BagTag Scan")].actionTime
AS bagScanTime,
timestamp_diff(
   $flightLeg.actions[contains($element.actionCode, "Checkin")].actionTime,
   $flightLeg.actions[contains($element.actionCode, "BagTag
Scan")].actionTime
) AS diff
FROM baggageinfo $s, $s.bagInfo[].flightLegs[] AS $flightLeg
WHERE ticketNo=176234463813
AND starts_with($s.bagInfo[].routing, $flightLeg.fltRouteSrc)
```

Explanation: In the baggage data, every `flightLeg` has an actions array. There are three different actions in the actions array. The action code for the first element in the array is Checkin/Offload. For the first leg, the action code is Checkin and for the other legs, the action code is Offload at the hop. The action code for the second element of the array is BagTag Scan. In the query above, you determine the difference in action time between the bag tag scan and check-in time. You use the `contains` function to filter the action time only if the action code is Checkin or BagScan. Since only the first flight leg has details of check-in and bag scan, you additionally filter the data using `starts_with` function to fetch only the source code `fltRouteSrc`.

Output:

```
{"flightNo":"BM572","checkinTime":"2019-03-02T03:28:00Z",
"bagScanTime":"2019-03-02T04:52:00Z","diff":-5040000}
```

Example 2 : Find list of passengers whose destination is **JTR**.

```
SELECT fullname FROM baggageInfo $bagInfo
WHERE ends_with($bagInfo.bagInfo[].routing, "JTR")
```

Output:

```
{"fullname":"Lucinda Beckman"}
{"fullname":"Gerard Greene"}
{"fullname":"Michelle Payne"}
```

index_of function

The `index_of` function determines the position of the first character of the search string at its first occurrence if any.

```
returnvalue index_of(source, search_string [, start_position])

source ::= any*
search_string ::= any*
start_position ::= integer*
returnvalue ::= integer
```

Various return values:

- Returns the position of the first character of the search string at its first occurrence. The position is relative to the start position of the string (which is zero).
- Returns -1 if `search_string` is not present in the source.
- Returns 0 for any value of source if the `search_string` is of length 0.
- Returns NULL if any argument is NULL.
- Returns NULL if any argument is an empty sequence or a sequence with more than one item.
- Returns error if `start_position` argument is not an integer.

Example 1: Determine at which position "-" is found in the estimated arrival time of the first leg for the passenger with ticket number **1762320569757**.

```
SELECT index_of(bag.baginfo.flightLegs[0].estimatedArrival,"-")
FROM BaggageInfo bag
WHERE ticketNo=1762320569757
```

Output:

```
{"Column_1":4}
```

Example 2: Determine at which position "/" is found in the routing of the first leg for passenger with ticket number **1762320569757**. This will help you determine how many characters are there for the source point for the passenger with ticket number **1762320569757**.

```
SELECT index_of(bag.baginfo.routing,"/")
FROM BaggageInfo bag
WHERE ticketNo=1762320569757
```

Output:

```
"Column_1":3}
```

replace function

The `replace` function returns the source with every occurrence of the search string replaced with the replacement string.

```
returnvalue replace(source, search_string [, replacement_string])

source ::= any*
search_string ::= any*
replacement_string ::= any*
returnvalue ::= string
```

Example: Replace the source location of the passenger with ticket number **1762320569757** from SFO to **SOF**.

```
SELECT replace(bag.bagInfo[0].routing,"SFO","SOF")
FROM baggageInfo bag
WHERE ticketNo=1762320569757
```

Output:

```
{"Column_1":"SOF/IST/ATH/JTR"}
```

Example 2: Replace the double quote in the passenger name with a single quote.

If your data might contain a double quote in the passenger's name, you can use replace function to change the double quote to a single quote.

```
SELECT fullname,
replace(fullname, "\"", "'") as new_fullname
FROM BaggageInfo bag
```

reverse function

The `reverse` function returns the characters of the source string in reverse order, where the string is written beginning with the last character first.

```
returnvalue reverse(source)

source ::= any*
returnvalue ::= string
```

Example: Display the full name and reverse the full name of the passenger with ticket number **1762330498104**.

```
SELECT fullname, reverse(fullname)
FROM baggageInfo
WHERE ticketNo=1762330498104
```

Output:

```
{"fullname":"Michelle Payne","Column_2":"enyaP ellehciM"}
```

Query execution plan

A query execution plan is the sequence of operations Oracle NoSQL Database performs to run a query.

Overview of query plan

A query execution plan is internally structured as a tree of plan iterators.

Each kind of iterator evaluates a different kind of expression that may appear in a query. In general, the choice of index and the kind of associated index predicates can have a drastic effect on query performance. As a result, you as a developer often want to see what index is used by a query and what predicates have been pushed down to it. Based on this information, you may want to force the use of a different index via index hints. This information is contained in the query execution plan. All Oracle NoSQL drivers provide APIs to display the execution plan of a query. All Oracle NoSQL graphical UIs including the IntelliJ, VSCode, and Eclipse plugins along with the Oracle Cloud Infrastructure Console include controls for displaying the query execution plan.

Some of the most common and important iterators used in queries are :

TABLE iterator

A table iterator is responsible for

- Scanning the index used by the query (which may be the primary index).
- Applying any filtering predicates pushed to the index
- Retrieve the rows pointed to by the qualifying index entries if necessary. If the index is covering, the result set of the TABLE iterator is a set of index entries, otherwise, it is a set of table rows.

> **Note:**
>
> An index is called a covering index with respect to a query if the query can be evaluated using only the entries of that index, that is, without the need to retrieve the associated rows.

A TABLE iterator will always have the following properties:

- **target table:** The name of the target table in the query.
- **index used:** The name of the index used by the query. If the primary index were used, “primary index” would appear as the value of this property.
- **covering index:** Whether the index is covering or not.

- **row variable:** The name of a variable ranging over the table rows produced by the TABLE iterator. If the index is covering, no table rows are produced and this variable is not used.
- **index scans:** Contains the start and stop conditions that define the index scans to be performed.

A TABLE iterator has 2 more optional properties:

- **index row variable:** The name of a variable ranging over the index entries produced by the TABLE iterator. Every time a new index entry is produced by the index scan, the index variable will be bound to that entry.
- **index filtering predicate:** A predicate evaluated on every index entry produced by the index scan. If the result of this evaluation is true, the index variable is bound to this entry and the entry or its associated table row is returned as the result of the next() call on the TABLE iterator. Otherwise, the entry is skipped, the next entry from the index scan is produced, the predicate is evaluated again on this entry and it continues until a qualifying entry is found.

SELECT iterator

It is responsible for executing the SELECT expression.

RECEIVE iterator

It is a special internal iterator that separates the query plan into 2 parts:

1. The RECEIVE iterator itself and all iterators that are above it in the iterator tree are executed at the driver.
2. All iterators below the RECEIVE iterator are executed at the replication nodes (RNs); these iterators form a subtree rooted at the unique child of the RECEIVE iterator.

In general, the RECEIVE iterator acts as a **query coordinator**. It sends its subplan to appropriate RNs for execution and collects the results. It may perform additional operations such as sorting and duplicate elimination and propagates the results to its ancestor iterators (if any) for further processing.

Distribution kinds

A distribution kind specifies how the query will be distributed for execution across the RNs participating in an Oracle NoSQL database (a store). The distribution kind is a property of the RECEIVE iterator.

Different choices of Distribution kinds are:

- **SINGLE_PARTITION:** A SINGLE_PARTITION query specifies a complete shard key in its WHERE clause. As a result, its full result set is contained in a single partition, and the RECEIVE iterator will send its subplan to a single RN that stores that partition. A SINGLE_PARTITION query may use either the primary-key index or a secondary index.
- **ALL_PARTITIONS:** Queries use the primary-key index here and they don't specify a complete shard key. As a result, if the store has M partitions, the RECEIVE iterator will send M copies of its subplan to be executed over one of the M partitions each.
- **ALL_SHARDS**: Queries use a secondary index here and they don't specify a complete shard key. As a result, if the store has N shards, the RECEIVE iterator will send N copies of its subplan to be executed over one of the N shards each.

Populating the tables to view the query execution plan :

If you want to follow along with the examples, download the script baggageschema_loaddata.sql and run it as shown below. This script creates the table used in the example and loads data into the table.

Start your KVSTORE or KVLite and open the SQL.shell.

```
java -jar lib/kvstore.jar kvlite -secure-config disable
java -jar lib/sql.jar -helper-hosts localhost:5000 -store kvstore
```

Using the `load` command, run the script.

```
load -file baggageschema_loaddata.sql
```

Creating indexes:

Create the following indexes in the `baggageInfo` table as shown below.

1. Create an index on passengers reservation code.

   ```
   CREATE INDEX fixedschema_conf ON baggageInfo confNo)
   ```

2. Create an index on the full name and phone number of passengers

   ```
   CREATE INDEX compindex_namephone ON baggageInfo
   (fullName,contactPhone)
   ```

3. Create an index on three fields, when the bag was last seen, the last seen station, and the arrival date and time.

   ```
   CREATE INDEX simpleindex_arrival ON
   baggageInfo(bagInfo[].lastSeenTimeGmt as ANYATOMIC,
   bagInfo[].bagArrivalDate as ANYATOMIC,
   bagInfo[].lastSeenTimeStation as ANYATOMIC)
   ```

Query 1: Using primary key index with an index range scan

Fetch the bag details of passengers for ticket numbers in a range.

```
SELECT fullname, ticketNo,
bag.bagInfo[].tagNum,bag.bagInfo[].routing
FROM BaggageInfo bag WHERE
1762340000000 < ticketNo AND ticketNo < 1762352000000
```

Plan:

```
{
  "iterator kind" : "RECEIVE",
  "distribution kind" : "ALL_PARTITIONS",
  "input iterator" :
  {
```

```
    "iterator kind" : "SELECT",
    "FROM" :
    {
      "iterator kind" : "TABLE",
      "target table" : "BaggageInfo",
      "row variable" : "$$bag",
      "index used" : "primary index",
      "covering index" : false,
      "index scans" : [
        {
          "equality conditions" : {},
          "range conditions" : { "ticketNo" : { "start value" :
1762340000000, "start inclusive" : false, "end value" : 1762352000000, "end
inclusive" : false } }
        }
      ]
    },
    "FROM variable" : "$$bag",
    "SELECT expressions" : [
      {
        "field name" : "fullname",
        "field expression" :
        {
          "iterator kind" : "FIELD_STEP",
          "field name" : "fullname",
          "input iterator" :
          {
            "iterator kind" : "VAR_REF",
            "variable" : "$$bag"
          }
        }
      },
      {
        "field name" : "ticketNo",
        "field expression" :
        {
          "iterator kind" : "FIELD_STEP",
          "field name" : "ticketNo",
          "input iterator" :
          {
            "iterator kind" : "VAR_REF",
            "variable" : "$$bag"
          }
        }
      },
      {
        "field name" : "tagNum",
        "field expression" :
        {
          "iterator kind" : "ARRAY_CONSTRUCTOR",
          "conditional" : true,
          "input iterators" : [
            {
              "iterator kind" : "FIELD_STEP",
              "field name" : "tagNum",
```

```
            "input iterator" :
            {
              "iterator kind" : "ARRAY_FILTER",
              "input iterator" :
              {
                "iterator kind" : "FIELD_STEP",
                "field name" : "bagInfo",
                "input iterator" :
                {
                  "iterator kind" : "VAR_REF",
                  "variable" : "$$bag"
                }
              }
            }
          }
        ]
      }
    },
    {
      "field name" : "routing",
      "field expression" :
      {
        "iterator kind" : "ARRAY_CONSTRUCTOR",
        "conditional" : true,
        "input iterators" : [
          {
            "iterator kind" : "FIELD_STEP",
            "field name" : "routing",
            "input iterator" :
            {
              "iterator kind" : "ARRAY_FILTER",
              "input iterator" :
              {
                "iterator kind" : "FIELD_STEP",
                "field name" : "bagInfo",
                "input iterator" :
                {
                  "iterator kind" : "VAR_REF",
                  "variable" : "$$bag"
                }
              }
            }
          }
        ]
      }
    }
  ]
 }
}
```

Explanation:

- The root iterator of this query plan is a **RECEIVE** iterator with a single child (input iterator) that is a **SELECT** iterator.

- The value of the **FROM** field is an iterator; in this case, it is a **TABLE** iterator.
- The **primary key index** is used and the index is not covering (as you need to scan the table to fetch columns other than the index entries).
- The index scan property contains the start and stop conditions that define the index scans to be performed.
- The FROM variable is the name of a variable ranging over the records produced by the FROM iterator. Here the FROM iterator is a TABLE iterator, and the FROM variable (**$$bag**)is the same as the **row variable** of the TABLE iterator, as the index used is not covering.
- In the SELECT expression four fields (**fullname,ticketNo,bag.bagInfo[].tagNum,bag.bagInfo[].routing**) are fetched. These correspond to four field names and field expressions in the SELECT expression clause. For the first two fields, the field expression is computed using **FIELD_STEP** iterator. For the last 2 fields, an **ARRAY_CONSTRUCTOR** iterator is used which iterates over the corresponding arrays to fetch the field value.

Query 2: Using primary key index with an index predicate

Fetch the bag details of passengers for ticket numbers satisfying one of the two ranges of values.

```
SELECT fullname, ticketNo, bag.bagInfo[].tagNum,bag.bagInfo[].routing
FROM BaggageInfo bag WHERE ticketNo > 1762340000000 OR ticketNo <
1762352000000;
```

Plan:

```
{
  "iterator kind" : "RECEIVE",
  "distribution kind" : "ALL_PARTITIONS",
  "input iterator" :
  {
    "iterator kind" : "SELECT",
    "FROM" :
    {
      "iterator kind" : "TABLE",
      "target table" : "BaggageInfo",
      "row variable" : "$$bag",
      "index used" : "primary index",
      "covering index" : false,
      "index scans" : [
        {
          "equality conditions" : {},
          "range conditions" : {}
        }
      ],
      "index filtering predicate" :
      {
        "iterator kind" : "OR",
        "input iterators" : [
          {
```

```
          "iterator kind" : "GREATER_THAN",
          "left operand" :
          {
            "iterator kind" : "FIELD_STEP",
            "field name" : "ticketNo",
            "input iterator" :
            {
              "iterator kind" : "VAR_REF",
              "variable" : "$$bag"
            }
          },
          "right operand" :
          {
            "iterator kind" : "CONST",
            "value" : 1762340000000
          }
        },
        {
          "iterator kind" : "LESS_THAN",
          "left operand" :
          {
            "iterator kind" : "FIELD_STEP",
            "field name" : "ticketNo",
            "input iterator" :
            {
              "iterator kind" : "VAR_REF",
              "variable" : "$$bag"
            }
          },
          "right operand" :
          {
            "iterator kind" : "CONST",
            "value" : 1762352000000
          }
        }
      ]
    }
  },
  "FROM variable" : "$$bag",
  "SELECT expressions" : [
    {
      "field name" : "fullname",
      "field expression" :
      {
        "iterator kind" : "FIELD_STEP",
        "field name" : "fullname",
        "input iterator" :
        {
          "iterator kind" : "VAR_REF",
          "variable" : "$$bag"
        }
      }
    },
    {
      "field name" : "ticketNo",
```

```
      "field expression" :
      {
        "iterator kind" : "FIELD_STEP",
        "field name" : "ticketNo",
        "input iterator" :
        {
          "iterator kind" : "VAR_REF",
          "variable" : "$$bag"
        }
      }
    },
    {
      "field name" : "tagNum",
      "field expression" :
      {
        "iterator kind" : "ARRAY_CONSTRUCTOR",
        "conditional" : true,
        "input iterators" : [
          {
            "iterator kind" : "FIELD_STEP",
            "field name" : "tagNum",
            "input iterator" :
            {
              "iterator kind" : "ARRAY_FILTER",
              "input iterator" :
              {
                "iterator kind" : "FIELD_STEP",
                "field name" : "bagInfo",
                "input iterator" :
                {
                  "iterator kind" : "VAR_REF",
                  "variable" : "$$bag"
                }
              }
            }
          }
        ]
      }
    },
    {
      "field name" : "routing",
      "field expression" :
      {
        "iterator kind" : "ARRAY_CONSTRUCTOR",
        "conditional" : true,
        "input iterators" : [
          {
            "iterator kind" : "FIELD_STEP",
            "field name" : "routing",
            "input iterator" :
            {
              "iterator kind" : "ARRAY_FILTER",
              "input iterator" :
              {
                "iterator kind" : "FIELD_STEP",
```

```
                "field name" : "bagInfo",
                "input iterator" :
                {
                  "iterator kind" : "VAR_REF",
                  "variable" : "$$bag"
                }
              }
            }
          }
        ]
      }
    }
  ]
 }
}
```

Explanation:

- The root iterator of this query plan is a **RECEIVE** iterator with a single child (input iterator) that is a **SELECT** iterator.
- The value of the **`FROM`** field is an iterator; in this case, it is a **TABLE** iterator.
- The **primary key index** is used and the index is not covering (as you need to scan the table to fetch columns other than the index entries).
- The **`index filtering predicate`** evaluates the filter criteria on the **`ticketNo`** field. Using the greater than and less than operators the filter condition is evaluated.
- The FROM variable is the name of a variable ranging over the records produced by the FROM iterator. Here the FROM iterator is a TABLE iterator, and the FROM variable (**`$$bag`**) is the same as the **row variable** of the TABLE iterator, as the index used is not covering.
- In the SELECT expression four fields (**`fullname`**,**`ticketNo`**,**`bag.bagInfo[].tagNum`**, **`bag.bagInfo[].routing`**) are fetched. These correspond to four field names and field expressions in the SELECT expression clause. For the first two fields, the field expression is computed using **`FIELD_STEP`** iterator. For the last 2 fields, an **`ARRAY_CONSTRUCTOR`** iterator is used which iterates over the corresponding arrays to fetch the field value.

Query 3: Using a secondary index with an index range scan

Fetch the bag details for a particular reservation code.

```
SELECT fullName,bag.ticketNo, bag.confNo, bag.bagInfo[].tagNum,
bag.bagInfo[].routing FROM BaggageInfo bag WHERE bag.confNo="FH7G1W"
```

Plan:

```
{
  "iterator kind" : "RECEIVE",
  "distribution kind" : "ALL_SHARDS",
  "input iterator" :
  {
```

```
"iterator kind" : "SELECT",
"FROM" :
{
  "iterator kind" : "TABLE",
  "target table" : "BaggageInfo",
  "row variable" : "$$bag",
  "index used" : "fixedschema_conf",
  "covering index" : false,
  "index scans" : [
    {
      "equality conditions" : {"confNo":"FH7G1W"},
      "range conditions" : {}
    }
  ]
},
"FROM variable" : "$$bag",
"SELECT expressions" : [
  {
    "field name" : "fullName",
    "field expression" :
    {
      "iterator kind" : "FIELD_STEP",
      "field name" : "fullName",
      "input iterator" :
      {
        "iterator kind" : "VAR_REF",
        "variable" : "$$bag"
      }
    }
  },
  {
    "field name" : "ticketNo",
    "field expression" :
    {
      "iterator kind" : "FIELD_STEP",
      "field name" : "ticketNo",
      "input iterator" :
      {
        "iterator kind" : "VAR_REF",
        "variable" : "$$bag"
      }
    }
  },
  {
    "field name" : "confNo",
    "field expression" :
    {
      "iterator kind" : "FIELD_STEP",
      "field name" : "confNo",
      "input iterator" :
      {
        "iterator kind" : "VAR_REF",
        "variable" : "$$bag"
      }
    }
```

```
},
{
  "field name" : "tagNum",
  "field expression" :
  {
    "iterator kind" : "ARRAY_CONSTRUCTOR",
    "conditional" : true,
    "input iterators" : [
      {
        "iterator kind" : "FIELD_STEP",
        "field name" : "tagNum",
        "input iterator" :
        {
          "iterator kind" : "ARRAY_FILTER",
          "input iterator" :
          {
            "iterator kind" : "FIELD_STEP",
            "field name" : "bagInfo",
            "input iterator" :
            {
              "iterator kind" : "VAR_REF",
              "variable" : "$$bag"
            }
          }
        }
      }
    ]
  }
},
{
  "field name" : "routing",
  "field expression" :
  {
    "iterator kind" : "ARRAY_CONSTRUCTOR",
    "conditional" : true,
    "input iterators" : [
      {
        "iterator kind" : "FIELD_STEP",
        "field name" : "routing",
        "input iterator" :
        {
          "iterator kind" : "ARRAY_FILTER",
          "input iterator" :
          {
            "iterator kind" : "FIELD_STEP",
            "field name" : "bagInfo",
            "input iterator" :
            {
              "iterator kind" : "VAR_REF",
              "variable" : "$$bag"
            }
          }
        }
      }
    ]
```

```
        }
      }
    ]
  }
}
```

Explanation:

- The root iterator of this query plan is a **RECEIVE** iterator with a single child (input iterator) that is a **SELECT** iterator.
- The value of the **FROM** field is an iterator; in this case, it is a **TABLE** iterator.
- The index **fixedschema_conf** is used and the index is not covering (as you need to scan the table to fetch columns other than the index entries).
- The index scan property contains the start and stop conditions that define the index scans to be performed.
- The FROM variable is the name of a variable ranging over the records produced by the FROM iterator. Here the FROM iterator is a TABLE iterator, and the FROM variable (**$$bag**) is the same as the **row variable** of the TABLE iterator, as the index used is not covering.
- In the SELECT expression four fields (**fullname,ticketNo,confNo,bag.bagInfo[].tagNum,bag.bagInfo[].routing**) are fetched. These correspond to five field names and field expressions in the SELECT expression clause. For the first three fields, the field expression is computed using **FIELD_STEP** iterator. For the last two fields, an **ARRAY_CONSTRUCTOR** iterator is used which iterates over the corresponding arrays to fetch the field value.

Query 4: Using the primary index

Fetch the name and routing details of all male passengers.

```
SELECT fullname,bag.bagInfo[].routing FROM BaggageInfo bag
WHERE gender!="F"
```

Plan:

```
{
  "iterator kind" : "RECEIVE",
  "distribution kind" : "ALL_PARTITIONS",
  "input iterator" :
  {
    "iterator kind" : "SELECT",
    "FROM" :
    {
      "iterator kind" : "TABLE",
      "target table" : "BaggageInfo",
      "row variable" : "$$bag",
      "index used" : "primary index",
      "covering index" : false,
      "index scans" : [
        {
          "equality conditions" : {},
```

```
      "range conditions" : {}
    }
  ]
},
"FROM variable" : "$$bag",
"WHERE" :
{
  "iterator kind" : "NOT_EQUAL",
  "left operand" :
  {
    "iterator kind" : "FIELD_STEP",
    "field name" : "gender",
    "input iterator" :
    {
      "iterator kind" : "VAR_REF",
      "variable" : "$$bag"
    }
  },
  "right operand" :
  {
    "iterator kind" : "CONST",
    "value" : "F"
  }
},
"SELECT expressions" : [
  {
    "field name" : "fullname",
    "field expression" :
    {
      "iterator kind" : "FIELD_STEP",
      "field name" : "fullname",
      "input iterator" :
      {
        "iterator kind" : "VAR_REF",
        "variable" : "$$bag"
      }
    }
  },
  {
    "field name" : "routing",
    "field expression" :
    {
      "iterator kind" : "ARRAY_CONSTRUCTOR",
      "conditional" : true,
      "input iterators" : [
        {
          "iterator kind" : "FIELD_STEP",
          "field name" : "routing",
          "input iterator" :
          {
            "iterator kind" : "ARRAY_FILTER",
            "input iterator" :
            {
              "iterator kind" : "FIELD_STEP",
              "field name" : "bagInfo",
```

```
                "input iterator" :
                {
                  "iterator kind" : "VAR_REF",
                  "variable" : "$$bag"
                }
              }
            }
          }
        ]
      }
    }
  ]
 }
}
```

Explanation:

- The root iterator of this query plan is a **RECEIVE** iterator with a single child (input iterator) that is a **SELECT** iterator.
- The value of the **`FROM`** field is an iterator; in this case, it is a **TABLE** iterator.
- The **primary key index** is used and the index is not covering (as you need to scan the table to fetch columns other than the index entries).
- The FROM variable is the name of a variable ranging over the records produced by the FROM iterator. Here the FROM iterator is a TABLE iterator, and the FROM variable (**`$$bag`**) is the same as the **row variable** of the TABLE iterator, as the index used is not covering.
- In the SELECT expression two fields (**`fullname,bag.bagInfo[].routing`**) are fetched. These correspond to two field names and field expressions in the SELECT expression clause. For the first field, the field expression is computed using **`FIELD_STEP`** iterator. For the second field, an **`ARRAY_CONSTRUCTOR`** iterator is used which iterates over the corresponding array to fetch the field value.

Query 5: Sort the data using a Covering index

Fetch the name and phone number of all passengers.

```
SELECT bag.contactPhone, bag.fullName FROM BaggageInfo bag
ORDER BY bag.fullName
```

Plan:

```
{
  "iterator kind" : "RECEIVE",
  "distribution kind" : "ALL_SHARDS",
  "order by fields at positions" : [ 1 ],
  "input iterator" :
  {
    "iterator kind" : "SELECT",
    "FROM" :
    {
      "iterator kind" : "TABLE",
```

```
      "target table" : "BaggageInfo",
      "row variable" : "$$bag",
      "index used" : "compindex_namephone",
      "covering index" : true,
      "index row variable" : "$$bag_idx",
      "index scans" : [
        {
          "equality conditions" : {},
          "range conditions" : {}
        }
      ]
    },
    "FROM variable" : "$$bag_idx",
    "SELECT expressions" : [
      {
        "field name" : "contactPhone",
        "field expression" :
        {
          "iterator kind" : "FIELD_STEP",
          "field name" : "contactPhone",
          "input iterator" :
          {
            "iterator kind" : "VAR_REF",
            "variable" : "$$bag_idx"
          }
        }
      },
      {
        "field name" : "fullName",
        "field expression" :
        {
          "iterator kind" : "FIELD_STEP",
          "field name" : "fullName",
          "input iterator" :
          {
            "iterator kind" : "VAR_REF",
            "variable" : "$$bag_idx"
          }
        }
      }
    ]
  }
}
```

Explanation:

- The root iterator of this query plan is a RECEIVE iterator with a single child (input iterator) that is a SELECT iterator. The only property of the RECEIVE iterator in this example is the distribution kind whose value is **ALL_SHARDS**.
- The results need to be sorted by **`fullName`**. The `fullName` is part of the **`compindex_namephone`** index. So in this example, you don't need a separate SORT operator. The sorting is done by the RECEIVE operator using its property `order`

`by fields at positions`, which is an array. The value of this array depends on the position of the field which is sorted in the SELECT expression.

```
"order by fields at positions" : [ 1 ]
```

- In this example, the order by is done using the **`fullName`** which is the second field in the SELECT expression. That is why you see **`1`** in the `order by fields at position` property of the iterator.
- The index **`compindex_namephone`** is used here and in this example, it is a covering index as the query can be evaluated using only the entries of the index.
- The index row variable is **`$$bag_idx`** which is the name of a variable ranging over the index entries produced by the TABLE iterator. Every time a new index entry is produced by the index scan, the `$$bag_idx` variable will be bound to that entry.
- When the FROM iterator is a TABLE iterator, the FROM variable is the same as either the index row variable or the row variable of the TABLE iterator, depending on whether the used index is covering or not. In this example, the FROM variable is the same as the index row variable (**`$$bag_idx`**) as the index is covering.
- This index row variable (`$$bag_idx`) will be referenced by iterators implementing the other clauses of the SELECT expression.
- In the SELECT expression two fields (`contactPhone`, `fullName`) are fetched. These correspond to two field names and field expressions in the SELECT expression clause. For both fields, the field expression is computed using **`FIELD_STEP`** iterator.

Query 6: Using a secondary index with an index predicate

Fetch the name, ticket number, and arrival date of passengers whose arrival date is greater than a given value.

```
SELECT fullName, bag.ticketNo, bag.bagInfo[].bagArrivalDate
FROM BaggageInfo bag WHERE EXISTS
bag.bagInfo[$element.bagArrivalDate >="2019-01-01T00:00:00"]
```

Plan:

```
{
  "iterator kind" : "RECEIVE",
  "distribution kind" : "ALL_SHARDS",
  "distinct by fields at positions" : [ 1 ],
  "input iterator" :
  {
    "iterator kind" : "SELECT",
    "FROM" :
    {
      "iterator kind" : "TABLE",
      "target table" : "BaggageInfo",
      "row variable" : "$$bag",
      "index used" : "simpleindex_arrival",
      "covering index" : false,
      "index row variable" : "$$bag_idx",
      "index scans" : [
```

```
      {
        "equality conditions" : {},
        "range conditions" : {}
      }
    ],
    "index filtering predicate" :
    {
      "iterator kind" : "GREATER_OR_EQUAL",
      "left operand" :
      {
        "iterator kind" : "FIELD_STEP",
        "field name" : "bagInfo[].bagArrivalDate",
        "input iterator" :
        {
          "iterator kind" : "VAR_REF",
          "variable" : "$$bag_idx"
        }
      },
      "right operand" :
      {
        "iterator kind" : "CONST",
        "value" : "2019-01-01T00:00:00"
      }
    }
  },
  "FROM variable" : "$$bag",
  "SELECT expressions" : [
    {
      "field name" : "fullName",
      "field expression" :
      {
        "iterator kind" : "FIELD_STEP",
        "field name" : "fullName",
        "input iterator" :
        {
          "iterator kind" : "VAR_REF",
          "variable" : "$$bag"
        }
      }
    },
    {
      "field name" : "ticketNo",
      "field expression" :
      {
        "iterator kind" : "FIELD_STEP",
        "field name" : "ticketNo",
        "input iterator" :
        {
          "iterator kind" : "VAR_REF",
          "variable" : "$$bag"
        }
      }
    },
    {
      "field name" : "bagArrivalDate",
```

```
        "field expression" :
        {
          "iterator kind" : "ARRAY_CONSTRUCTOR",
          "conditional" : true,
          "input iterators" : [
            {
              "iterator kind" : "FIELD_STEP",
              "field name" : "bagArrivalDate",
              "input iterator" :
              {
                "iterator kind" : "ARRAY_FILTER",
                "input iterator" :
                {
                  "iterator kind" : "FIELD_STEP",
                  "field name" : "bagInfo",
                  "input iterator" :
                  {
                    "iterator kind" : "VAR_REF",
                    "variable" : "$$bag"
                  }
                }
              }
            }
          ]
        }
      }
    ]
  }
}
```

Explanation:

- The root iterator of this query plan is a **RECEIVE** iterator with a single child (input iterator) that is a **SELECT** iterator.
- The value of the **`FROM`** field is an iterator; in this case, it is a **TABLE** iterator.
- The EXISTS condition is actually converted to a filtering predicate. There is one filtering predicate which is the whole WHERE expression. The index **`simpleindex_arrival`** is the only one applicable here and is used.
- The **`index filtering predicate`** evaluates the filter criteria on the **`bagArrivalDate`** field. Using the greater than and less than operators the filter condition is evaluated.
- The FROM variable is the name of a variable ranging over the records produced by the FROM iterator. Here the FROM iterator is a TABLE iterator, and the FROM variable (**`$$bag`**)is the same as the **row variable** of the TABLE iterator, as the index used is not covering.
- In the SELECT expression three fields (**`fullname`**, **`ticketNo`**,**`bag.bagInfo[].bagArrivalDat`**) are fetched. These correspond to three field names and field expressions in the SELECT expression clause. For the first two fields, the field expression is computed using **`FIELD_STEP`** iterator. For the last field, an **`ARRAY_CONSTRUCTOR`** iterator is used which iterates over the corresponding arrays to fetch the field value.

Query 7: Group data with fields as part of the index

Fetch the reservation code and count of bags for all passengers.

```
SELECT bag.confNo, count(bag.bagInfo) AS TOTAL_BAGS
FROM BaggageInfo bag GROUP BY bag.confNo;
```

Plan:

```
{
  "iterator kind" : "SELECT",
  "FROM" :
  {
    "iterator kind" : "RECEIVE",
    "distribution kind" : "ALL_SHARDS",
    "order by fields at positions" : [ 0 ],
    "input iterator" :
    {
      "iterator kind" : "SELECT",
      "FROM" :
      {
        "iterator kind" : "TABLE",
        "target table" : "BaggageInfo",
        "row variable" : "$$bag",
        "index used" : "fixedschema_conf",
        "covering index" : false,
        "index scans" : [
          {
            "equality conditions" : {},
            "range conditions" : {}
          }
        ]
      },
      "FROM variable" : "$$bag",
      "GROUP BY" : "Grouping by the first expression in the SELECT
list",
      "SELECT expressions" : [
        {
          "field name" : "confNo",
          "field expression" :
          {
            "iterator kind" : "FIELD_STEP",
            "field name" : "confNo",
            "input iterator" :
            {
              "iterator kind" : "VAR_REF",
              "variable" : "$$bag"
            }
          }
        },
        {
          "field name" : "TOTAL_BAGS",
          "field expression" :
```

```
          {
            "iterator kind" : "FN_COUNT",
            "input iterator" :
            {
              "iterator kind" : "FIELD_STEP",
              "field name" : "bagInfo",
              "input iterator" :
              {
                "iterator kind" : "VAR_REF",
                "variable" : "$$bag"
              }
            }
          }
        }
      ]
    }
  },
  "FROM variable" : "$from-1",
  "GROUP BY" : "Grouping by the first expression in the SELECT list",
  "SELECT expressions" : [
    {
      "field name" : "confNo",
      "field expression" :
      {
        "iterator kind" : "FIELD_STEP",
        "field name" : "confNo",
        "input iterator" :
        {
          "iterator kind" : "VAR_REF",
          "variable" : "$from-1"
        }
      }
    },
    {
      "field name" : "TOTAL_BAGS",
      "field expression" :
      {
        "iterator kind" : "FUNC_SUM",
        "input iterator" :
        {
          "iterator kind" : "FIELD_STEP",
          "field name" : "TOTAL_BAGS",
          "input iterator" :
          {
            "iterator kind" : "VAR_REF",
            "variable" : "$from-1"
          }
        }
      }
    }
  ]
}
```

Explanation:

- In this query, you group all bags based on the **confNo** of the users and determine the total count of bags belonging to each **confNo**.
- The group-by is index-based, that is the group-by field (confNo) is also part of the index used. This is indicated by the lack of any GROUP iterators. Instead, the grouping is done by the SELECT iterators.
- There are two **SELECT** iterators, the inner one has a **GROUP BY** property that specifies which of the SELECT-clause expressions are also grouping expressions. Here the group by fields is the first expression in the SELECT list (**bag.confNo**).

  ```
  "GROUP BY" : "Grouping by the first expression in the SELECT list"
  ```

- The index **fixedschema_conf** is used here and in this example, it is a non-covering index as the query also needs to fetch **count(bag.bagInfo)** which is outside of the entries of the index.
- When the FROM iterator is a TABLE iterator, the FROM variable is the same as either the **index row variable** or the **row variable** of the TABLE iterator, depending on whether the used index is covering or not. In this example, the inner FROM variable is the same as the row variable (**$$bag**) as the index is not covering.
- In the SELECT expression two fields are fetched: **bag.confNo,count(bag.bagInfo)**. These correspond to two field names and field expressions in the SELECT expression clause.
- The results returned by the inner SELECT iterators from the various RNs are partial groups, because rows with the same bag.confNo may exist at multiple RNs. So, regrouping and re-aggregation have to be performed by the driver. This is done by the outer SELECT iterator (above the RECEIVE iterator).
- The result is also sorted by confNo. The **order by fields at positions** property specifies the field used for sorting. The value of this array depends on the position of the field which is sorted in the SELECT expression. In this example **bag.confNo** is the first field in the SELECT expression. So order by fields at positions takes an array index of 0.

  ```
  "order by fields at positions" : [ 0 ]
  ```

- In the outer SELECT expression, two fields are fetched: bag.confNo,count(bag.bagInfo). The $from-1 FROM variable will be referenced by iterators implementing the other clauses of the outer SELECT expression. These correspond to two field names and field expressions in the outer SELECT expression clause. For the first field, the field expression uses **FIELD_STEP** iterator. The second field is the aggregate function **count**. The iterator **FUNC_SUM**is used to iterate the result produced by its parent iterator and determine the total number of bags.

Query 8: Using the secondary index with multiple index scans

Fetch the full name and tag number of passengers who are in the given list of names.

```
SELECT bagdet.fullName, bagdet.bagInfo[].tagNum
FROM BaggageInfo bagdet WHERE bagdet.fullName IN
```

```
("Lucinda Beckman", "Adam Phillips",
"Zina Christenson","Fallon Clements");
```

Plan:

```
{
  "iterator kind" : "SELECT",
  "FROM" :
  {
     "iterator kind" : "RECEIVE",
     "distribution kind" : "ALL_SHARDS",
     "order by fields at positions" : [ 0 ],
     "input iterator" :
     {
       "iterator kind" : "SELECT",
       "FROM" :
       {
         "iterator kind" : "TABLE",
         "target table" : "BaggageInfo",
         "row variable" : "$$bag",
         "index used" : "fixedschema_conf",
         "covering index" : false,
         "index scans" : [
           {
             "equality conditions" : {},
             "range conditions" : {}
           }
         ]
       },
       "FROM variable" : "$$bag",
       "GROUP BY" : "Grouping by the first expression in the SELECT list",
       "SELECT expressions" : [
          {
            "field name" : "confNo",
            "field expression" :
            {
              "iterator kind" : "FIELD_STEP",
              "field name" : "confNo",
              "input iterator" :
              {
                "iterator kind" : "VAR_REF",
                "variable" : "$$bag"
              }
            }
          },
          {
            "field name" : "TOTAL_BAGS",
            "field expression" :
            {
              "iterator kind" : "FN_COUNT",
              "input iterator" :
              {
                "iterator kind" : "FIELD_STEP",
                "field name" : "bagInfo",
```

```
              "input iterator" :
              {
                "iterator kind" : "VAR_REF",
                "variable" : "$$bag"
              }
            }
          }
        }
      ]
    }
  },
  "FROM variable" : "$from-1",
  "GROUP BY" : "Grouping by the first expression in the SELECT list",
  "SELECT expressions" : [
    {
      "field name" : "confNo",
      "field expression" :
      {
        "iterator kind" : "FIELD_STEP",
        "field name" : "confNo",
        "input iterator" :
        {
          "iterator kind" : "VAR_REF",
          "variable" : "$from-1"
        }
      }
    },
    {
      "field name" : "TOTAL_BAGS",
      "field expression" :
      {
        "iterator kind" : "FUNC_SUM",
        "input iterator" :
        {
          "iterator kind" : "FIELD_STEP",
          "field name" : "TOTAL_BAGS",
          "input iterator" :
          {
            "iterator kind" : "VAR_REF",
            "variable" : "$from-1"
          }
        }
      }
    }
  ]
}
```

Explanation:

- The root iterator of this query plan is a **RECEIVE** iterator with a single child (input iterator) that is a **SELECT** iterator.
- The value of the **`FROM`** field is an iterator; in this case, it is a **TABLE** iterator.
- The index **`compindex_namephone`** is used and the index is not covering (as you need to scan the table to fetch columns other than the index entries).

- Every value in the IN clause is evaluated using an index scan with an equality condition. There are four index scans that are performed each evaluating one equality condition.
- The FROM variable is the name of a variable ranging over the records produced by the FROM iterator. Here the FROM iterator is a TABLE iterator, and the FROM variable (**`$$bagdet`**) is the same as the **row variable** of the TABLE iterator, as the index used is not covering.
- In the SELECT expression two fields (**`fullname, bag.bagInfo[].tagNum`**) are fetched. These correspond to two field names and field expressions in the SELECT expression clause. For the first field, the field expression is computed using **`FIELD_STEP`** iterator. For the second field, an **`ARRAY_CONSTRUCTOR`** iterator is used which iterates over the corresponding arrays to fetch the field value.

Query 9: A SINGLE PARTITION query using a primary index

Select the ticket details (ticket number, reservation code, tag number, and routing) for a passenger with a specific ticket number and reservation code.

```
SELECT fullName,bag.ticketNo, bag.confNo, bag.bagInfo[].tagNum,
bag.bagInfo[].routing FROM BaggageInfo bag WHERE
bag.ticketNo=1762311547917 AND bag.confNo="FH7G1W"
```

Plan:

```
{
  "iterator kind" : "RECEIVE",
  "distribution kind" : "SINGLE_PARTITION",
  "input iterator" :
  {
    "iterator kind" : "SELECT",
    "FROM" :
    {
      "iterator kind" : "TABLE",
      "target table" : "BaggageInfo",
      "row variable" : "$$bag",
      "index used" : "primary index",
      "covering index" : false,
      "index scans" : [
        {
          "equality conditions" : {"ticketNo":1762311547917},
          "range conditions" : {}
        }
      ]
    },
    "FROM variable" : "$$bag",
    "WHERE" :
    {
      "iterator kind" : "EQUAL",
      "left operand" :
      {
        "iterator kind" : "FIELD_STEP",
        "field name" : "confNo",
        "input iterator" :
```

```
      {
        "iterator kind" : "VAR_REF",
        "variable" : "$$bag"
      }
    },
    "right operand" :
    {
      "iterator kind" : "CONST",
      "value" : "FH7G1W"
    }
  },
  "SELECT expressions" : [
    {
      "field name" : "fullName",
      "field expression" :
      {
        "iterator kind" : "FIELD_STEP",
        "field name" : "fullName",
        "input iterator" :
        {
          "iterator kind" : "VAR_REF",
          "variable" : "$$bag"
        }
      }
    },
    {
      "field name" : "ticketNo",
      "field expression" :
      {
        "iterator kind" : "FIELD_STEP",
        "field name" : "ticketNo",
        "input iterator" :
        {
          "iterator kind" : "VAR_REF",
          "variable" : "$$bag"
        }
      }
    },
    {
      "field name" : "confNo",
      "field expression" :
      {
        "iterator kind" : "FIELD_STEP",
        "field name" : "confNo",
        "input iterator" :
        {
          "iterator kind" : "VAR_REF",
          "variable" : "$$bag"
        }
      }
    },
    {
      "field name" : "tagNum",
      "field expression" :
      {
```

```
            "iterator kind" : "ARRAY_CONSTRUCTOR",
            "conditional" : true,
            "input iterators" : [
              {
                "iterator kind" : "FIELD_STEP",
                "field name" : "tagNum",
                "input iterator" :
                {
                  "iterator kind" : "ARRAY_FILTER",
                  "input iterator" :
                  {
                    "iterator kind" : "FIELD_STEP",
                    "field name" : "bagInfo",
                    "input iterator" :
                    {
                      "iterator kind" : "VAR_REF",
                      "variable" : "$$bag"
                    }
                  }
                }
              }
            ]
          }
        },
        {
          "field name" : "routing",
          "field expression" :
          {
            "iterator kind" : "ARRAY_CONSTRUCTOR",
            "conditional" : true,
            "input iterators" : [
              {
                "iterator kind" : "FIELD_STEP",
                "field name" : "routing",
                "input iterator" :
                {
                  "iterator kind" : "ARRAY_FILTER",
                  "input iterator" :
                  {
                    "iterator kind" : "FIELD_STEP",
                    "field name" : "bagInfo",
                    "input iterator" :
                    {
                      "iterator kind" : "VAR_REF",
                      "variable" : "$$bag"
                    }
                  }
                }
              }
            ]
          }
        }
      ]
    }
  }
```

Explanation:

- The root iterator of this query plan is a RECEIVE iterator with a single child (input iterator) that is a SELECT iterator.
- This query specifies a complete shard key in its WHERE clause. As a result, its full result set is contained in a single partition, and the RECEIVE iterator will send its subplan to a single RN that stores that partition.
- The value of the **`FROM`** field is an iterator; in this case, it is a **TABLE** iterator.
- A **SINGLE_PARTITION** query can reference a primary index or a secondary index. The **primary key index** is used in this example. The index is not covering (as you need to scan the table to fetch columns other than the index entries).
- The index scan property contains the start and stop conditions that define the index scans to be performed.
- The FROM variable is the name of a variable ranging over the records produced by the FROM iterator. Here the FROM iterator is a TABLE iterator, and the FROM variable (**`$$bag`**) is the same as the **row variable** of the TABLE iterator, as the index used is not covering.
- In the SELECT expression five fields (**`fullname, ticketNo,confNo, bag.bagInfo[].tagNum,bag.bagInfo[].routing`**) are fetched. These correspond to five field names and field expressions in the SELECT expression clause. For the first three fields, the field expression is computed using **`FIELD_STEP`** iterator. For the last 2 fields, an **`ARRAY_CONSTRUCTOR`** iterator is used which iterates over the corresponding arrays to fetch the field value.

Query 10: Group data with fields not part of any index

Fetch the source of passenger bags and the count of bags for all passengers and group the data by the source.

```
SELECT $flt_src as SOURCE, count(*) as COUNT FROM BaggageInfo $bag,
$bag.bagInfo.flightLegs[0].fltRouteSrc $flt_src GROUP BY $flt_src
```

Plan:

```
{
  "iterator kind" : "GROUP",
  "input variable" : "$gb-2",
  "input iterator" :
  {
    "iterator kind" : "RECEIVE",
    "distribution kind" : "ALL_PARTITIONS",
    "input iterator" :
    {
      "iterator kind" : "GROUP",
      "input variable" : "$gb-1",
      "input iterator" :
      {
        "iterator kind" : "SELECT",
        "FROM" :
        {
          "iterator kind" : "TABLE",
```

```
    "target table" : "BaggageInfo",
    "row variable" : "$bag",
    "index used" : "primary index",
    "covering index" : false,
    "index scans" : [
      {
        "equality conditions" : {},
        "range conditions" : {}
      }
    ]
  },
  "FROM variable" : "$bag",
  "FROM" :
  {
    "iterator kind" : "FIELD_STEP",
    "field name" : "fltRouteSrc",
    "input iterator" :
    {
      "iterator kind" : "ARRAY_SLICE",
      "low bound" : 0,
      "high bound" : 0,
      "input iterator" :
      {
        "iterator kind" : "FIELD_STEP",
        "field name" : "flightLegs",
        "input iterator" :
        {
          "iterator kind" : "FIELD_STEP",
          "field name" : "bagInfo",
          "input iterator" :
          {
            "iterator kind" : "VAR_REF",
            "variable" : "$bag"
          }
        }
      }
    }
  },
  "FROM variable" : "$flt_src",
  "SELECT expressions" : [
    {
      "field name" : "SOURCE",
      "field expression" :
      {
        "iterator kind" : "VAR_REF",
        "variable" : "$flt_src"
      }
    },
    {
      "field name" : "COUNT",
      "field expression" :
      {
        "iterator kind" : "CONST",
        "value" : 1
      }
```

```
            }
          ]
        },
        "grouping expressions" : [
          {
            "iterator kind" : "FIELD_STEP",
            "field name" : "SOURCE",
            "input iterator" :
            {
              "iterator kind" : "VAR_REF",
              "variable" : "$gb-1"
            }
          }
        ],
        "aggregate functions" : [
          {
            "iterator kind" : "FUNC_COUNT_STAR"
          }
        ]
      }
    },
    "grouping expressions" : [
      {
        "iterator kind" : "FIELD_STEP",
        "field name" : "SOURCE",
        "input iterator" :
        {
          "iterator kind" : "VAR_REF",
          "variable" : "$gb-2"
        }
      }
    ],
    "aggregate functions" : [
      {
        "iterator kind" : "FUNC_SUM",
        "input iterator" :
        {
          "iterator kind" : "FIELD_STEP",
          "field name" : "COUNT",
          "input iterator" :
          {
            "iterator kind" : "VAR_REF",
            "variable" : "$gb-2"
          }
        }
      }
    ]
  }
```

Explanation:

- In this query, you group passenger bags based on the flight source and determine the total number of bags belonging to one flight source.

- As the GROUP BY field (**`bagInfo.flightLegs[0].fltRouteSrc`** in this example) is not part of any index, you need a separate GROUP operator to do the grouping. This is indicated by the existence of the **GROUP** iterators in the execution plan. There are two **GROUP** iterators: one that operates at the driver (above the **RECEIVE** iterator) and another that operates at the RNs (below the **RECEIVE** iterator).
- The lower GROUP iterator has a SELECT iterator as input. The SELECT returns the **`fltRouteSrc`** and **`count`** of bags. The GROUP iterator will operate until the batch limit is reached. If the batch limit is defined as the max number N of results produced, the GROUP iterator will stop when up to N flight source groups have been created. If the batch limit is defined as the max number of bytes read, it will stop when this max is reached. The GROUP operator has an input variable. For the inner GROUP operator, the input variable is **`$gb-1`** and for the outer GROUP operator it is **`$gb-2.`**

  ```
  "iterator kind" : "GROUP","input variable" : "$gb-1",
  ```

- The **primary key index** is used here and in this example, it is not a covering index as the query has fields that are not part of the entries of the primary index.
- When the FROM iterator is a TABLE iterator, the FROM variable is the same as either the **index row variable** or the **row variable** of the TABLE iterator, depending on whether the used index is covering or not. Every time a next() call on the FROM iterator returns true, the variable will be bound to the result produced by that iterator. In this example, the FROM variable is the row variable as the index is not covering.
- This row variable (**`$bag`**) will be referenced by iterators implementing the other clauses of the inner SELECT expression.
- The GROUP iterator creates an internal variable (`$gb-1`) that iterates over the records produced by the SELECT expression.
- The result set produced by the lower GROUP iterator is partial: it may not contain all the `fltRouteSrc` groups and for the `fltRouteSrc` groups that it does contain, the count may be a partial sum (because all rows for a given `fltRouteSrc` may not have been retrieved when query execution stops). The upper GROUP iterator receives the partial results from each RN and performs the final grouping and aggregation. It operates the same way as the lower GROUP iterators and will keep operating until the are no more partial results from the RNs. At that point, the full and final result set is cached at the upper GROUP iterator and is returned to the application.
- The upper GROUP iterator creates an internal variable (**`$gb-2`**) that iterates over the records produced by the outer SELECT. The **`$gb-2`** variable has the `fltRouteSrc` and count of all bags grouped by `fltRouteSrc`.
- In the SELECT expression, two fields are fetched: `fltRouteSrc,count(*)`. These correspond to two field names and field expressions in the SELECT expression clause. For the first field, the field expression uses **`FIELD_STEP`** iterator. The second field is the aggregate function **`count`**. The iterator **`FUNC_SUM`** is used to iterate the result produced by its parent iterator and determine the total number of bags.

Table Modelling and Design

A critical part of the application development process is the task of modeling your data.

Proper modeling of your data is crucial to application performance, extensibility, application correctness, and finally, the ability for your application to support rich user experiences. In this

article, you will learn some crucial aspects of data modeling and understand guidelines on how to model your persistent data for an Oracle NoSQL Database application.

The Oracle NoSQL Database gives the data modeler a large range of flexibility with respect to modeling application data. Understanding the tradeoffs associated with each level of flexibility is extremely useful in making wise data modeling decisions.

Schema Flexibility in Oracle NoSQL Database

Unlike the relational database world with purely fixed schemas, NoSQL Database is largely about schema flexibility – that is the ability to easily change how data is organized and stored.

Schema flexibility in Oracle NoSQL Database mostly takes the form of non-scalar data types. These non-scalar data types can be used to embed flexible structures inside your tables.

Non-scalar data types:

Oracle NoSQL database supports the following non-scalar data types:

- **JSON** – JSON is a map of key/value pairs that can be used as a datatype of a column in the Oracle NoSQL Database. The JSON datatype gives you the ability to dynamically read and write attributes having no prior knowledge of what is and what is not stored in the JSON document. You can introspect into the document by reading from Oracle NoSQL Database as a JSON string, or you can specify path expressions as deep as you like into a hierarchy of JSON. As an example, you can create a JSON document that represents the variable terms and conditions of a contract. The document attribute names (or keys) can represent the tag or **name** of the contractual term or condition and the value of the attribute can represent the text of that term. Using a JSON brings you ultimate flexibility in your data model.
- **Records** – Records containing scalar or non-scalar values can be used as a datatype for a column in an Oracle NoSQL Database table. You can think of a record as a document with a fixed set of attributes, however noting that one or more attributes of the record can be a non-fixed array or JSON document, giving you the flexibility to extend a fixed document without modifying the schema. Records present an interesting intermediate step between the benefits of the fixed schema world (single copy of a schema) and the ultimate flexibility of the JSON world.
- **Arrays** – Arrays of scalar or non-scalar values can be used as a datatype for a column in an Oracle NoSQL Database table. Arrays can be convenient for storing a collection of event values. For example, you may wish to collect a list of behavioral segments for users as they browse web pages.

Trade-offs while using Flexible Schema:

Some guidelines that you can follow while considering flexible schema are listed below.

The Flexibility/Cost of scale Tradeoff :

When thinking about how flexible you want your schema to be, it's important to understand that the more flexible you make your schema, the bigger the challenge is for scaling your solution. For example, let's say that you are storing information on user behavior. And you want to store this information as the users access your website. You can implement one of the two options here. You can choose to model the solution using fixed columns for the required user attributes that you will need to track. Alternatively, you can choose to model this using a JSON document, giving you the flexibility to add and remove attributes for users without having to evolve your schema.

The second option may work quite well for small numbers of users; however, if you will need to scale this solution to large numbers of users, then you need extra storage. You also need additional compute overhead for processing the key/value pairs (attribute names and their values) in the JSON document. This could make the cost of scaling your solution prohibitive. Extra storage is needed to store the metadata along with the data (e.g. the attribute names) and extra compute is needed to serialize and de-serialize these documents. If you are using a replication factor of more than one, then that adds additional overhead for each tracked user. If a large scale is a major requirement for you then you'll most likely want to trade off flexibility for storage efficiency and consider using a more fixed schema.

The Flexibility/Latency Tradeoff :

In many NoSQL applications, low latency data access is a key requirement. In these situations, it's important to understand the potential tradeoff with respect to the I/O latency of using one data modeling method over another. In this respect, using a non-scalar data type such as a record, array, or JSON document will entail a read followed by an update.

For example, when you add a new value to an array your application must read the record from NoSQL Database first, add the value to the array, and then write it back to NoSQL Database. Even if performing this operation using the SQL UPDATE operator of Oracle NoSQL Database (which executes in the replication node), the record must still be read from persistent storage, de-serialized, modified, serialized, and written back. On a system with a spinning disk, this could cost anywhere from fifteen to thirty milliseconds (or more). For certain applications like online advertising, this may be beyond the latency SLA that can be tolerated. If you are faced with similar stringent latency SLAs then you should consider favoring a child table approach which eliminates the read and will allow you to simply perform a write of the new value. Of course, the tradeoff here is one of flexibility for low latency.

For more information on when to use parent-child tables, see Using Parent-Child tables in Oracle NoSQL Database.

Updates to Non-Scalars versus Inserts :

The Oracle NoSQL Database storage engine is based on an append-only architecture, also known as log-structured storage. Log structured storage systems perform extremely well for insert operations where the new records to be inserted are simply appended to the end of the log. Update operations involve appending the updated record to the log and then marking the old record for deletion. Records marked for deletion are regularly cleaned from NoSQL Database's logs to free up disk space by a background process called the cleaner. Although the cleaner is highly optimized, it will add some CPU and I/O overhead to the replication node. The more updates performed by your application, the more log cleaning activity there will be.

As a guideline, if you have extreme performance goals for your application (or for a specific table), you should strongly consider trying to craft your data model by using parent/child tables versus non-scalar columns, giving you the potential for replacing updates with inserts. For more information on when to use parent-child tables, see Using Parent-Child tables in Oracle NoSQL Database.

Static Vs Dynamic Data :

In many applications, it's possible to identify portions of the data that are somewhat static and change relatively slowly, and other portions of the data which are highly dynamic and change frequently, even at millisecond granularity. For example, in online advertising, campaigns are a relatively slow-moving piece of data while the budget spent (impressions or clicks delivered) can change every few milliseconds as millions of users load web pages that have ads associated with the campaign. The data pertaining to budgets is a case of highly dynamic data. This is an example of a scenario that has a high velocity of write operations. Oracle NoSQL database is a log-structured, append-only storage architecture, where inserts are more optimal than an update operation.

For the more static portions of your data, the flexibility of the non-scalar datatypes may be an attractive option for your application. Using a JSON document could provide an extensible way for your application to interact with this data without undue sacrifice to performance. On the flip side, for data that is changing rapidly or being inserted rapidly, you'll want to consider trading off flexibility for this data and use a parent table with a fixed schema and a child table with a fixed schema. Whether or not you choose to model the rapidly changing data as a parent or child table will depend largely on how you wish to access it. For more information on when to use parent-child tables, see Using Parent-Child tables in Oracle NoSQL Database.

Choice of Keys in NoSQL Database

Primary keys and shard keys are important elements in your schema and help you access and distribute data efficiently.

Primary keys and shard keys are indispensable for data distribution and easy accessibility. You specify primary keys and shard keys only when you create a table. They remain in place for the life of the table, and cannot be changed or dropped.

Using Primary Keys and Shard Keys in Oracle NoSQL tables

Primary Keys

You must designate one or more primary key columns when you create your table. The primary key cannot be changed and exists for the life of the table. A primary key uniquely identifies every row in the table. For simple CRUD operations, Oracle NoSQL Database uses the primary key to retrieve a specific row to read or modify. Since the underlying storage in NoSQL Database is based on a key/value model, the choice of the primary key can greatly enhance the performance of certain lookup operations.

Shard Keys

The main purpose of shard keys is to distribute data across the Oracle NoSQL Database cluster for scalability and to co-locate the records that share the same shard key on the same physical node for easy reference. These records can be accessed atomically and efficiently.

Impact of keys while developing an application:

In an Oracle NoSQL Database, replication nodes are grouped together to form the shards of the NoSQL Database cluster. When an application asks to retrieve the record for a given key, the NoSQL Database driver will hash a portion of the key (denoted as the **shard key**) to identify the shard that houses the data. Once the shard is identified, the NoSQL Database driver can choose to read the data from the most optimal replica in the shard, depending on the requested consistency level. With

respect to the write operations, the NoSQL Database driver will always route the write requests to the dynamically elected leader node of the shard. Hence, from the perspective of workload scaling, you can generally think of this architecture as being scaled by adding shards. Oracle NoSQL Database supports the online elastic expansion of the cluster by adding shards, however, without the proper selection of a shard key, expanding the cluster will be useless in scaling your solution.

How you design primary keys and shard keys has huge implications on scaling and realizing the system throughput. For instance, when records share shard keys, you can delete multiple table rows in an atomic operation, or retrieve a subset of rows in your table in a single atomic operation. In addition to enabling scalability, well-designed shard keys can improve performance by requiring fewer cycles to put data on, or get data from, a single shard. Shard keys designate storage on the same shard to facilitate efficient queries for key values. However, because you want your data to be distributed across the shards for best performance and scalability, you will want to avoid shard keys that have a small number of unique values.

Important factors to consider when choosing a shard key:

- **Cardinality:** Low cardinality field groups are stored together on a small number of shards. In turn, those shards require frequent data rebalancing, increasing the likelihood of hot shard issues. Instead, each shard key should have high cardinality, where it can express several million values. For best performance and value, choose fields with high cardinalities, such as identity numbers, where millions of records are possible.
- **Atomicity:**Only objects that share the same shard key can participate in a transaction. If you have a requirement for ACID transactions that span multiple records, choose only a shard key that lets you meet that requirement.

Best practices to follow:

- **Uniform distribution of shard keys:**Operations may be limited by the capacity of a single shard. When shard keys are uniformly distributed, no single shard limits the capacity of the system. Choosing one or more columns whose values are known to be uniformly distributed is ideal.
- **Query Isolation:** Queries should be targeted to a specific shard to maximize scalability. If queries are not isolated to a single shard, the query will be applied to all shards. This is less efficient and increases query latency. Make sure your queries fetch data stored in a single shard. Well-designed shard keys can improve performance by getting data from a single shard. Shard keys designate storage on the same shard to facilitate efficient queries for key values. Specify the fields (which are frequently used in your application queries) as shard keys.

Key Sizes and Key Only Modeling Methods

Oracle NoSQL Database caches the keys for each table. So the key size is a critical component to the effective use of memory and ultimately may be a determining factor in the ability of Oracle NoSQL Database to service your performance SLAs. Hence, it is important for you to create primary keys that are as efficient as possible with respect to size. For workloads that require very low latencies for the read and writes (single to low double-digit milliseconds) across millions of operations per second, exploiting cached keys in NoSQL's B-trees can be the make or break of building an application capable of achieving these stringent requirements. Furthermore, if you can encode what would otherwise be non-key values as part of the primary key and also size your keys and the NoSQL Database cluster carefully, then you can realize the enormous benefits of memory cached B-tree access methods that are maintained with ACID semantics. For highly optimal, ultra-low latency applications, Oracle NoSQL provides key-only accessors for those workloads that can model everything as

key-only data. Oracle NoSQL Database offers convenient key-only access APIs such as `multiKetGeys` and `tableKeysIterator` for doing key-only scans.

When considering whether or not key-only modeling of your data is right for your application, you should consider the following:

- **Latency and throughput SLAs** – Do you have very stringent latency and throughput SLAs that would require a key-only model? Can you afford to perform an I/O when retrieving a value, noting that for spinning disks, the average latency of retrieving your value could be anywhere from fifteen to thirty milliseconds and for Single Shared Disk (SSD) this could be anywhere from one to 5 milliseconds.
- **Spinning disk versus SSDs** – If you are considering using SSDs and your latency SLAs are for reads that can comfortably fit within the 5-millisecond range then it's probably not worth the effort to try and craft a key-only model for your application.
- **Code maintainability and extensibility** – Key-only modeling brings large performance benefits to your application at the potential cost of code maintainability and extensibility. You may find that encoding your value into the key can ultimately be a complex and esoteric strategy. Ultimately, you will have to make a judgment call on whether or not the code you develop and maintain is too complex and esoteric to be worth the benefit of the key-only solution.
- **Accurate sizing data** – Is it possible for you to derive a somewhat accurate sizing of your keys such that you can adequately size the Oracle NoSQL Database cluster? Sizing the cluster and the cache of each replication node will be crucial to exploiting the benefits of a key-only data model.

Key Column Ordering and Query-ability

In Oracle NoSQL Database, the order of declaration for key columns is crucial to satisfying partial key lookup queries. This is because of the way that the storage engine manages the underlying B-trees. You can think of composite keys as an ordered concatenation of the columns specified in the DDL for the key declaration (primary key or index key). You should think of the order from the most significant column to the least significant column based on the appearance of the columns in the DDL for the key. If your table has a composite primary key (a primary key with more than one column), then the primary key becomes a concatenation of the string representation of each column. Here for better performance of queries, it is important to specify the most commonly used query column as the most significant column in the primary key.

As you start to think about how you will size your cluster and your Oracle NoSQL Database caches, a critical consideration is to get an estimate of your key sizes. Sizing your caches so that Oracle NoSQL can keep most or all of the index nodes in memory can help your application realize enormous performance benefits. Understanding how keys are serialized and stored persistently can help you in getting a more accurate sizing estimate. In Oracle NoSQL Database, numeric keys are stored as compressed String values but must remain sortable when in string format. This means that a numeric key must be a fixed size when represented as a key string. See Initial Capacity Planning for more details on shard capacity, shard storage, and throughput capacities and how to estimate total shards and machines.

Using Indexes in NoSQL Database

n Oracle NoSQL Database, the query processor can identify which of the available indexes are beneficial for a query and rewrite the query to make use of such an index.

Using an index means scanning a contiguous subrange of its entries, potentially applying further filtering conditions on the entries within this subrange, and using the primary keys stored in the 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.

In an Oracle NoSQL Database, a primary-key index is always created by default. This index maps the primary key columns of a table to the physical location of the table rows. Furthermore, if no other index is available, the primary index will be used. In other words, there is no pure **table scan** mechanism; a table scan is equivalent to a scan via the primary-key index. When it comes to indexes and queries, the query processor must answer two questions:

1. Is an index applicable to a query? That is, will accessing the table via this index be more efficient than doing a full table scan (via the primary index)?
2. Among the applicable indexes, which index or combination of indexes is the best to use?

There are no statistics on the number and distribution of values in a table column. As a result, the query processor has to rely on some simple heuristics in choosing among the applicable indexes. In addition, SQL for Oracle NoSQL Database allows for the inclusion of index hints in the queries. You can use index hints to force the use of a particular index in queries. You can use a query execution plan to understand what indexes are being used in the query. For more information on how a query is executed, see Query execution plan.

Secondary Index

There will be cases where you will want to use a secondary index to support some of your read requirements. Each secondary index that you add to a table will incur some overhead for writes as each index will need to be maintained. The good news with Oracle NoSQL is that secondary index partitions live on the same shard as the primary data, so the updates to the secondary index are limited on a per-shard basis. Index updates in Oracle NoSQL are also atomic, so your application can be guaranteed that updates to records in the shard are consistent with updates to the secondary index and these structures will never be out of sync. Another factor for consideration is that Oracle NoSQL Database nodes will keep the non-leaf index nodes in the cache, and will never cache the leaf portion (i.e. the data record). This gives the indexed scan an enormous performance benefit (for systems using spinning disk) over the non-indexed scan.

There are several things that you should think about when deciding on using a secondary index in Oracle NoSQL Database:

- Filtering data close to the source – In Oracle NoSQL Database, secondary indexes are the primary mechanism for you to utilize when your query needs a filter and that filter needs to be executed as close as possible to the data. To fully understand why you may need a secondary index to filter your data for querying, let's consider your options for scanning the data in a table:
  - Unordered parallel table scan with no full shard key – The shard key is a table column or multiple columns used to control how the rows of that table are distributed. The main purpose of shard keys is to distribute data across the Oracle NoSQL Database Cloud cluster for scalability, and to position records that share the same shard key locally for easy reference and access. When you write a query using filters as columns that are part of the shard key but also include other columns, then you end up doing a parallel table scan. Each shard is scanned in parallel and the data is

returned to your application. This will return every record in the table across all shards in the NoSQL Database.

  - Ordered or unordered parallel index scan – The B-tree index at each shard is scanned in parallel. If an ordered scan is requested, the results are merged and presented.

- Each option for scanning a table has its own costs and benefits and you should carefully weigh these tradeoffs and use what you know about the application requirements and expected workload to help guide your modeling decision.
  - Efficient range scans – Will it be common for your queries to restrict the value ranges? For example, if your application needs to answer queries like “find all records between a range of dates” then using secondary indexes in Oracle NoSQL Database will be the easiest and most efficient way for your application to answer these types of queries.
  - Workload and index maintenance update – Is it acceptable for writes to incur some extra overhead for index maintenance? Does your workload exhibit heavy read activity where latency for reads is more important than incurring extra write overhead?

See Tuning and Optimizing SQL queries for more guidelines on using indexes in queries.

Transactions in NoSQL database

In Oracle NoSQL Database, a transaction is treated as a logical, atomic unit of work that entails a single database operation.

Every data modification in the database takes place in a single transaction, managed by the system. Database developers do not have the ability to group multiple operations into a single transaction because there isn't the notion of begin/end transactions. In a database, transactional semantics are often described in terms of ACID properties.

ACID properties

In Oracle NoSQL Database, transactions maintain all the following properties and developers can control some of them.

- **Atomicity:** Transaction either completes or fails in its entirety. There is no in-between state or no partial transactions.
- **Consistency:** Transaction leaves the database in a valid state.
- **Isolation:** No two transactions mingle or interfere with each other. Developers get the same result when the two transactions are executed in sequence or executed in parallel.
- **Durability:** Changes in a transaction are saved and the changes survive any type of failure (network, disk, CPU, or a power failure).

Developers can define a wide range of consistency levels depending on the application's needs with the Oracle NoSQL Database Direct Driver. In addition, the Oracle NoSQL Database Drivers (commonly called the SDKs) support eventual and absolute consistency.

Developers can also configure durability such that updated rows in the database survive any failure with the Oracle NoSQL Database Direct Driver. Durability is not configurable in the SDKs.

Atomicity and Isolation are not configurable but Oracle NoSQL Database allows you to control consistency and durability policies in order to trade-off the performance for application needs. Some NoSQL databases only support eventual consistency but have no mechanism for absolute consistency.

Shard keys play an important role in achieving the ACID properties in the Oracle NoSQL database. For instance, when records share shard keys, you can delete multiple table rows in an atomic operation, or retrieve a subset of rows in your table in a single atomic operation. In addition to enabling scalability, well-designed shard keys can improve performance by requiring fewer cycles to put data on, or get data from, a single shard.

The NoSQL table hierarchy is an ideal data model for applications that need some data normalization, but also require predictable, low latency at scale. The hierarchy links different tables to enable left outer joins, combining rows from two or more tables based on related columns between them. Such joins execute efficiently since rows from the parent-child tables are co-located in the same shards. Also, writes to multiple tables in a table hierarchy obey transactional ACID properties since the records residing in each table of the hierarchy share the same shard key. All write operations perform as a single atomic unit. So all of the write operations will execute successfully, or none of them will.

Using Parent-Child tables in the Oracle NoSQL database

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

Many NoSQL databases support data types like arrays and maps. When modeling a data relationship, application developers may find it easier to have each parent row store its child rows inside an array or a map in a nested structure. By doing so, not only is the data relationship denormalized but it has the potential for creating large parent rows, especially when the hierarchy is heavily nested, resulting in inefficient storage and poor performance. Oracle NoSQL Database's table hierarchy is the ideal data model to avoid issues associated with arrays and maps. One of the biggest benefits of using child tables over embedded arrays is for those workloads that have a high velocity of write operations. When using embedded arrays, the write operations become updates, but when they are modeled as child tables, those operations become inserts. Inserts in a log-structured, append-only storage architecture are much more optimal than updates. Utilizing a table hierarchy should be considered when building data relationships in Oracle NoSQL Database.

The NoSQL table hierarchy is an ideal data model for applications that need some data normalization, but also require predictable, low latency at scale. The hierarchy links different tables to enable left outer joins, combining rows from two or more tables based on related columns between them. Such joins execute efficiently since rows from the parent-child tables are co-located in the same shards. Also, writes to multiple tables in a table hierarchy obey transactional ACID properties since the records residing in each table of the hierarchy share the same shard key. All write operations perform as a single atomic unit. So all of the write operations will execute successfully, or none of them will.

The Benefits of a Table Hierarchy

Oracle NoSQL Database table hierarchy comes with the following benefits:

- **Highly efficient for storing data in a parent-child hierarchy** - Parent and child rows are stored in separate NoSQL tables, reducing the size of parent rows compared with the single parent with child rows in nested arrays or maps. Write operations on parent or

child tables create new versions of smaller rows and store these changes efficiently, given the append-only architecture of Oracle NoSQL Database.

- **Highly performant for read and write workloads** - Parent and child rows reside in the same local shard, enabling write and read operations to achieve high performance since all records in the hierarchy can be read or written in a single network call.
- **Highly flexible for fine-grained authorization** - Access rights to a parent or child table can be configured individually based on conditions at run-time, offering granular and flexible authorization.
- **Scalable ACID transactions** - Uniquely balance the goals of scalability, low latency, and ACID by co-locating parent and child data on the same shard.
- **Table joins** - Data can be queried using the nested table clause or left outer joins.

Characteristics of parent-child tables:

- A child table inherits the primary key columns of its parent table.
- All tables in the hierarchy have the same shard key columns, which are specified in the create table statement of the root table.
- A parent table cannot be dropped before its children are dropped.
- A referential integrity constraint is not enforced in a parent-child table.

A NoSQL table hierarchy not only captures the relationship between data entities but also takes advantage of the co-location of the parent-child rows to offer highly performant retrievals and superior scalability. The table hierarchy enables applications to implement ACID transactions. All data in the same parent-child rows are stored in the same shard and can be committed as a single database operation to ensure atomicity, consistency, isolation, durability.

Glossary

Index