Requests for Cassandra - Advanced Topics for NoSQL

Amine MILIANI Louis GAILLET Dimitrije DJOKIC
Nathan IMMACOLATO

Contents

Datamodel and Import

- We create a keyspace, we use it, then we create some types needed, then we create the table with the types. We chose a type instead of a map because we preferred to display only few attributes per query. Moreover we were focused on map in practical work 2 so when we had to choose between these 2 options we prefered to use type in order to improve our knowledge on Cassandra
- Cassandra doesn't allow for loading JSON files, only CSV ones. We decided instead on loading a .cql file, where each line would be an INSERT statement, for each JSON element. This was done with the `./cassanda/json-cassandra.lisp` script. We also needed to change `$date` to `date1` in order for the data to be read by CQL. This was done in vim, with a `:%s/$date/date1/g` command.
- Cassandra and CQLSH were running on docker containers, with a volume attached. It enabled us to all work on the same setup, and to load the same data file.

datamodel.cql

```
CREATE KEYSPACE IF NOT EXISTS restaurants_inspections
```

```
  WITH REPLICATION = {'class' : 'SimpleStrategy',
  'replication_factor' : 3};
USE restaurants_inspections;

CREATE TYPE IF NOT EXISTS coord (type VARCHAR,
  coordinates list<Double>);
CREATE TYPE IF NOT EXISTS address (building VARCHAR,
  coord frozen <coord>, street VARCHAR, zipcode VARCHAR);
CREATE TYPE IF NOT EXISTS dateType (date1 bigInt);
CREATE TYPE IF NOT EXISTS gradeType(date frozen<dateType>,
  grade VARCHAR, score INT);
CREATE TABLE IF NOT EXISTS restaurants(address frozen<address>,
  borough VARCHAR, cuisine VARCHAR, grades list<frozen<gradeType>>,
  name VARCHAR, restaurant_id VARCHAR, PRIMARY KEY (restaurant_id));
```

Requests

Easy requests

How many elements there are in the table, useful to know how many elements were able to be inserted

```
SELECT count(*) FROM restaurants;
```

Restaurant names for certain borough and cuisine

```
SELECT name FROM restaurants WHERE cuisine = 'Japanese'
  AND borough = 'Brooklyn' ALLOW FILTERING;
```

- But allowing filtering can be unstable, so we prefer to index the attributes.

```
CREATE INDEX IF NOT EXISTS cusineI ON restaurants(cusisine);
CREATE INDEX IF NOT EXISTS boroughI ON restaurants(borough);
SELECT name FROM restaurants WHERE cuisine = 'Japenese'
  AND borough = 'Brooklyn';
```

Restaurant name and cuisine for a certain restaurant name

```
CREATE INDEX IF NOT EXISTS nameI ON restaurants(name);
SELECT cuisine, borough FROM restaurants WHERE name = 'Kasumi';
```

Address except coordinates for a certain name restaurant

```
CREATE INDEX IF NOT EXISTS nameI ON restaurants(name);
SELECT name, address.building, address.street, address.zipcode,
  borough from restaurants WHERE name = 'Kasumi';
```

Historic of grades for a certain name restaurant

```
CREATE INDEX IF NOT EXISTS nameI ON restaurants(name);
SELECT grades from restaurants WHERE name='Kasumi';
```

Restaurant names and cuisine type from a certain borough

```
CREATE INDEX IF NOT EXISTS boroughI ON restaurants(borough);
SELECT name, cuisine FROM restaurants WHERE borough = 'Brooklyn';
```

Restaurant names and adresses from a certain cuisine type

```
CREATE INDEX IF NOT EXISTS cusineI ON restaurants(cuisine);
SELECT name, address.building, address.street, address.zipcode,
  borough FROM restaurants WHERE cuisine = 'Italian';
```

Restaurant names and historic of grades for a certain type of cuisine

```
CREATE INDEX IF NOT EXISTS cusineI ON restaurants(cuisine);
SELECT name, grades FROM restaurants WHERE cuisine='Italian';
```

Medium request

Gives a distribution of bakery restaurants.

- First we have a function that holds the number of bakeries and the number of total restaurants.

```
CREATE OR REPLACE FUNCTION distribution (state tuple<INT, INT>,val VARCHAR )
  CALLED ON NULL INPUT RETURNS tuple<INT,INT> LANGUAGE java
  AS '
    if (val !=null)
    {
      state.setInt(0, state.getInt(0)+1);
      if(val.equals("Bakery"))
      {
        state.setInt(1, state.getInt(1)+1);
      }
    }
    return state;';
```

- Then we divide the number of bakeries by the total.

```
CREATE OR REPLACE FUNCTION distribution_final ( state tuple<int,int> )
  CALLED ON NULL INPUT RETURNS double LANGUAGE java
  AS '
    double r = 0;
    if (state.getInt(0) == 0) return null;
    r = state.getInt(1);
    r/= state.getInt(0);
    return Double.valueOf(r);';
```

- Then we have an aggregate to be able to call it in a SELECT statement.

```
CREATE AGGREGATE IF NOT EXISTS distribution_bakery ( VARCHAR )
  SFUNC distribution STYPE tuple<int,int>
  FINALFUNC distribution_final INITCOND (0,0);
```

- Here we have a SELECT statement using it.

```
SELECT distribution_bakery(cuisine) FROM restaurants;
```

Hard request

Count how many restaurants there are for each type of something, like cuisine or borough

- First we need a function for taking a *type* value, and compare it with what it already knows. When it knows it, it increments the counter associated to it. When it doesn't know it, it remembers it with a counter set to 1.

```
CREATE OR REPLACE FUNCTION state_group_and_count( state map<text, int>, type text )
  CALLED ON NULL INPUT
  RETURNS map<text, int>
  LANGUAGE java AS '
    Integer count = (Integer) state.get(type);
    if (count == null)
    {
      count = 1;
    }
    else
    {
      count++;
    }
    state.put(type, count);
    return state; ' ;
```

- Then we use that function in an aggregate for it to be called in a SELECT statement.

```
CREATE OR REPLACE AGGREGATE group_and_count(text)
  SFUNC state_group_and_count
  STYPE map<text, int>
  INITCOND {};
```

- We then use the aggregate in a SELECT statement on something, here 'cuisine'.

```
SELECT group_and_count(cuisine) FROM restaurants;
```

- Or here, the count of restaurant cuisines for restaurants in Brooklyn.

```sql
SELECT group_and_count(cuisine) FROM restaurants WHERE borough = 'Brooklyn';
```