Complex Systems Theory and Practice

Lecture 12

Piotr Syga

2022-01-05

Select – a recap

- Selects follow the same syntax as in SQL however there are significant constraints.
- The select query, by default, may not condition the selection (no `WHERE` clause) or has to use the whole partition key in the condition.
- In order to allow conditions on a different column, we have to use `ALLOW FILTERING`, use a different table, a view or create a secondary index.
- We can use writetime(column) to get the time of the last update of each row in the column.
- The results may be `ORDER`ed BY one or many clustering columns.
- We can `GROUP BY` by one or more clustering column.
- We can use standard aggregate functions – `COUNT`, `SUM`, `MIN`, `MAX`, `AVG` or create our own.

Select – a recap

Examples

```
SELECT * FROM StudendsByName;
SELECT lastName, writetime(lastName) FROM StudendsByName;
SELECT * FROM StudendsByName WHERE lastName='Doe'; — will result in an error
SELECT * FROM StudendsByName WHERE lastName='Doe' AND firstName='Jane';
SELECT * FROM StudendsByName ORDER BY startingYear;
SELECT MIN(startingYear) FROM StudendsByName;
```

Creating aggregate functions

```
CREATE OR REPLACE FUNCTION avgState ( state tuple<int,bigint>, val int )
CALLED ON NULL INPUT
RETURNS tuple<int,bigint>
LANGUAGE java AS
$$ if (val !=null) {
      state.setInt(0, state.getInt(0)+1);
      state.setLong(1, state.getLong(1)+val.intValue());
      }
   return state; $$
;

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

CREATE AGGREGATE mean(int)
SFUNC avgState
STYPE tuple<int,bigint>
FINALFUNC avgFinal
INITCOND (0,0);
```

Inserts

- Inserts follow a familiar syntax, yet we always should enumerate the columns we are inserting data into and we do not have to insert anything into non-key columns.
- If we insert data with the same primary key, the data is overwritten.
- `INSERT INTO` <table> (columns) `VALUES`
- `INSERT INTO` addressBook(name, email) `VALUES` ('John Doe', ['JD@gmail.com','john@doe.com']);
- `INSERT INTO` phoneBook(name, phone) `VALUES` ('John Doe', {555555555, 555123456});
- `INSERT INTO` location(name, address) `VALUES` ('John Doe', {'home': '81st Street, NY, NY', 'office' : 'Grunwaldzka, Wrocław' });

Updates

- Similarly to selects, updates require the whole primary key as a condition in `WHERE` clause.
- Cassandra supports temporary updates: `UPDATE ... USING TTL` <seconds> changes the value only for a given time, after the expiration it is removed (so we have no value in the column).
- We can not update simple-data-columns as in SQL and we can modify the contents of a collection – adding to set column = column + {new values}, removing by column = column - {old values}, column={} removes all values
- `UPDATE` phoneBook `SET` phone = phone + {555000001} `WHERE` name = 'John Doe';
- `UPDATE` location `SET` address = address-{'home': '81st Street, NY, NY'} `WHERE` name = 'John Doe';

Deletes

- Deletes follow the same rules that updates.
- We can delete the whole row, just like in SQL `DELETE FROM <table> WHERE <condition>;`
- We can also delete a specific column value in a single row (similarly to not inserting the value it is displayed as null, yet nothing is stored) `DELETE <column> FROM <table> WHERE <condition>;` Naturally, all the restrictions on the deleted columns (e.g., cannot be a part of the key) and conditions apply.
- TTL removal after the time passes is not a delete and is bascially free operation in terms of cost.

Batches

- As a distributed database Cassandra does not support ACID transactions.
- We have a counterpart – `BATCH` operations.
- `BATCH`es allow us to use multiple modification queries (inserts, updates, deletes), possibly on a different tables, as an (almost) atomic operation.
- Syntax:

```
BEGIN BATCH
<inserts>
<updates>
<deletes>
APPLY BATCH;
```

Materialized Views and Indexes

- In order to use `WHERE` clause on a non-partition key column we have to use `ALLOW FILTERING`, e.g., `SELECT * FROM` StudendsByName `WHERE` indexNo=200833;
- Such a query has to scan all the records and is highly inefficient, to speed it up, yet still performing a costly operation we may create a secondary index on a table, so that queries are possible – `CREATE INDEX ON` <StudendsByName> (indexNo);
- A more efficient way is to create a Materialized View – `CREATE MATERIALIZED VIEW` <table> `AS SELECT` .... <filters>
- Primary key of the table has to be NOT NULL AND new partition key has to be NOT NULL, all table's primary key kolumn have to be a part of MV primary key
- Materialized Views are kept up to date when the data from original table is changed

```
CREATE MATERIALIZED VIEW view1 AS
SELECT indexNo, lastName, firstName, email
FROM StudendsByName
WHERE indexNo IS NOT NULL AND lastName IS NOT NULL AND firstName IS NOT NULL

PRIMARY KEY (indexNo, lastName, firstName);
```

Imports and exports

- To import data from csv file to Cassandra the table has to be already present in the table, the whole primary key must be consistent.
- Importing: `COPY` <table> (columns) `FROM` '<path>/<file>' `WITH DELIMITER=',' AND HEADER=TRUE;`
- Exporting: `COPY` <table> (columns) `TO` '<path>/<file>' `WITH DELIMIETER=','`
- We have to import all the key-columns, yet we do not have to import all the data from the file. We cannot import data to store it in non-existent columns.
- We can export a subset of columns to a file.
- Importing JSON requires using Bulk Loader or using `INSERT INTO` <table> `JSON` '{ <data> }'

Cassandra and programming languages

Cassandra is supplied with drivers (e.g. DataStax) allowing us to use many of the most popular programming language

- ODBC/JDBC
- python (2.5+ and 3.0+)
- C++
- C#
- Node.js
- Spark
- Go (API extension for Java drivers)
- PHP
- Ruby

Note that as with SQL we need to install the proper drivers before running the server and establishing the connection

Simple example – Python

```
from cassandra.cluster import Cluster

c1=Cluster() #or Cluster(<list of IPs>, port=<connection port>)

session = c1.connect('csKS') #keyspace name from Lecture 11
#since we created the table with a typo we keep the names
session.execute("INSERT INTO StudendsByID (indexNo, lastName, firstName, major) VALUES (123098,'Smith', 'Alice', 'BDA')")
session.execute("INSERT INTO StudendsByID (indexNo, lastName, firstName) VALUES (123999,'Bean', 'Sean')")
selects= session.execute("SELECT FROM * StudendsByID")
for s in selects:
    print s.indexNo, lastName, firstName
session.execute("DELETE FROM StudendsByID WHERE indexNo=123999")
selects= session.execute("SELECT FROM * StudendsByID")
for s in selects:
    print s.indexNo, lastName, firstName
```

Simple example – Go

```
package main

import (
    "fmt"
    "log"

    "github.com/gocql/gocql"
)

func main() {
    cluster := gocql.NewCluster("172.102.1.1", "172.102.1.4", "192.168.1.0")
    cluster.Keyspace = "csKS"
    cluster.Consistency = gocql.Quorum
    session, _ := cluster.CreateSession()
    defer session.Close()

    if err := session.Query("INSERT INTO StudendsByID (indexNo, lastName, firstName, major) VALUES (?, ?, ?,?)",
        100999, 'Smith', 'Hugo', 'BDA').Exec(); err != nil {
        log.Fatal(err)
    }
    var fn string
    var ln string
    if err := session.Query(`SELECT firstName, lastName FROM StudendsByID WHERE indexNo =? `,
        100999).Consistency(gocql.One).Scan(&id, &text); err != nil {
        log.Fatal(err)
    }
    fmt.Println("Student:", fn, ln)

    students := session.Query(`SELECT firstName, lastName FROM StudendsByID`).Iter()
    for iter.Scan(&id, &text) {
        fmt.Println("Student:", fn, ln)
    }
    if err := iter.Close(); err != nil {
        log.Fatal(err)
    }
}
```

Simple example – Java

```
import com.datastax.driver.core.Cluster;
import com.datastax.driver.core.Session;

public class Batch {

   public static void main(String args[]){

      //Creating Cluster object
      Cluster cluster = Cluster.builder().addContactPoint("127.0.0.1").build();

      //Creating Session object
      Session session = cluster.connect("csKS");


      //query
      String query =" BEGIN BATCH INSERT INTO StudendsByID (indexNo, lastName, firstName, major) VALUES "
         + (100654, 'Hook','James','IKw');"
         + "INSERT INTO StudendsByName (lastName, firstName,startingYear, email, indexNo, major,tutor) VALUES "
         + " ('Hook','James',2020, 'jh@bestmails.world', 100645, 'IKw','Greg Michels');"
         + "UPDATE StudendsByID SET major = 'IB' WHERE indexNo =100999;"
         + "DELETE major FROM StudendsByID WHERE indexNo =123098;"
         + "APPLY BATCH;";


      //Executing the query
      session.execute(query);

      System.out.println("Changes done");
   }
}
```

Modeling

Relational approach:

- We analyze the problem
- We determine the main entities
- We describe the attributes of the data
- We design the tables
- We normalize, design the queries and joins
- We design the application

Modeling

Cassandra approach:

- We start with the application – designing queries that will be needed for the app's functionality
- We design the tables with queries in mind – we have to remember that no JOINs are available and that the conditions should be made only on the key
- Instead of normalization we optimize for reads so that minimal number of queries may be needed and efficiency of the queries in distributed environment
- We create CQL schema – we design tables, views and indexes using 'query first' approach

Modeling

Designing steps:

1. We create an ERD of the project.
2. We design the workflows of the project – actions that will determine the queries connected with each entity – the access patterns.
3. We identify the queries and assign priorities to them (based on their frequency and importance) .
4. We select the partition key – so that the data is split in a possibly even manner across the nodes and that a single part is neither too large nor (if it is not required) unique.
5. We (optionally) select clustering columns to speed up the reads (allows selection of a particular row or a range of data).

Modeling

6. We create Cassandra tables with the selected key (partition key and clustering column).
7. We consider adding additional fields to the table.
8. We determine if some data may be simplified by user defined type.
9. We exclude counter types from the base tables (we create counter tables).
10. We bucket the data with high resolution.
11. If the schema needs changing (esp. the key) during the lifetime, create a new table, move the data in the background using data broker, use shadow writes, and switch API to the new table after the data is synchronized. Data migration tools may be used, e.g. described here.

Example

Let us consider the app for which our `csKS` keyspace may be used. Let us assume that we are creating JSOS-like web app with student's courses, grades and so on.

- We need a separate table for logging in: e-mail as a partition key, password (clustering column for speeding up the selects), uuid (a field to return on successful login – we may use timeuuid for keeping ids similar for students from the same year and major).
- We need a table for users (containing their data): uuid (or timeuuid – the same as above), firstName, lastName, indexNo, e-mail, major etc. – if the most often used access pattern will by indexNo, we could use StudendsByID, if by name, StudendsByName (possibly with indexNo added as a clustering column).
- We need a table (or tables) for classes – possible partition keys may include the course code, the lecturer, the major, the time of the classes, ...

Example

- We need a table for student classes – a map-like (many-to-many relation in SQL) classes containing information connecting students with their classes. It may be realized as two tables (organized by classes and by students) with atomic fields, composite (user defined) field for the students and collections for either students or classes to store signed-ins.
- Similarly to the student tables, we would require tables for teachers.
- How do we keep the tables synchronized? We should keep separate API for each access scenario and bind them with `BATCH` inserts, updates and deletes so that all tables are always (eventually) synchronized.

Things to remember

- Cassandra is designed with distributed systems in mind.
- We have to design the schema with the 'query first approach'
- There is Cassandra-specific language, similar to SQL in syntax, yet having major differences – CQL.
- Cassandra tables are similar to the ones from SQL yet they are stored in different way, moreover we specify partition key and clustering columns and cannot or should not perform some of the operations known from SQL.
- No JOINs are possible.
- The types do not have to be atomic and we can create our own user defined types.
- The nodes of Cassandra are organized in a ring architecture, which determines what data is stored where.
- We can manually determine the replication multiplicity, strategy and the consistency type.

What we did not cover

- technicalities behind the actual data storing – SSTables, MemTable, CommitLog, ...
- read and write paths
- compaction strategies
- synthetic sharding
- protocols of data dissemination and gossiping
- Bloom filters
- types of indexes
- utilizing Kafka