Big Data Project Report

Xiaohui Ma, Shenshu Zhou, Yue Yin
CMPUT 391 Group 4
April 1, 2016

1. Object

- To implement a NoSQL distributed database system using Cassandra to be able to store all of call detail record (CDR) with 2 terabytes random data into a large table and over 400 columns.
- To develop a C program to populate random data for the table
- To specify five SQL queries which satisfying the requirements
  - four of the five queries will retrieve information from all distributed nodes
  - one query must contain at least 10 conditions (atomic formulas) in the where clause;
  - two queries must contain both the group by and order by clauses
  - at least three queries are range queries, i.e., a query retrieving all rows where some column value is between an upper and lower boundary; and
  - none of the queries are non-trivial, i.e., a simple key-search.
- To evaluate and compare the results of these five queries

2. Bash Scripts for the Process:

stop_remote_sever.sh – to stop Cassandra server on all nodes

initialize.sh – to reset database, all data in keysapce ‘group4’ will be removed; then upload all necessary files and the configuration file to all nodes

check_storage_space.sh – to check the current storage status

start_remote_sever.sh - to start Cassandra on all nodes

create_table.sh – to drop the table and re-create it

check_server_status.sh – to check the running status of all nodes

3. Performance Tuning on Cassandra Configurations

concurrent_writes - set this to a high value to allow more concurrent I/O

thrift_framed_transport_size_in_mb - this is the maximum message length

auto_snapshot - set to false to disable auto snapshot to reduce amount I/O

compaction_throughput_mb_per_sec - set this to 0 disables throttling

request_scheduler - set the scheduler to RoundRobinScheduler

throttle_limit - set to a higher value to increase the number of in-flight requests per client

4. Table Structure

Our cdr table follows the the provided scheme (setup1.sql), with 470 columns. And it has 6 primary keys to ensure all queries can work properly. The cdr table has its specific structure by following the assignment required. It has 470 columns and 5 column keys, partitioned on MOBILE_ID_TYPE.

Since Cassandra does not have the feature of GROUP BY, we use the User-Defined Aggregate Function (UDA) to achieve the GROUP BY feature. This is defined in setup.cql when creating the table.

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

5. Programs

We wrote our generator and insertion part separately. The benefit of it that is the whole CPU can focus on generate data first, it can generate 40MB/SECOND on average. We wrote a python program to generate data at the beginning, but it worked inefficiently, so we rewrote a C program which provides a better performance. The generator program is designed such that it can generate number of days data with what we input. All datas will be written into a csv file with date on it. And we wrote a Python program to insert all the data into our database. We have also tried to use a sstableloader that was written in Java to load datas directly, but the result is unreasonably slow. So we finally decided to use our original python inserter.

6. CQL Queries

Query 1

```
SELECT SERVICE_NODE_ID
FROM CDR
WHERE MOBILE_ID_TYPE IN (1,2,3,4) AND
CITY_ID >= 1 AND
CITY_ID <= 5
LIMIT 100;
```

Satisfying requirement: Ranged query and order by. We use order by when we create the table, which order by CITY_ID. We restricted MOBILE_ID_TYPE if it is either 1,2,3 or 4 and restricted CITY_ID by a range of 1 to 5. And it has a limit of 100 selections. Also MOBILE_ID_TYPE is a partition key, we do not need to use ‘ALLOW FILTERING’, since we use partition key and ordered key.

Potential use: In real life, some companies might want to know that which node is used the most in some specific mobile id type with a range of cities. It is useful if they are considering which node’s capacity is the most necessary to be extended.

Query 2

```
SELECT COUNT(SEQ_NUM)
FROM CDR
WHERE MOBILE_ID_TYPE = 1 AND
CITY_ID =5 AND MONTH_DAY = 14 AND
MSC_CODE > 1 AND
MSC_CODE < 1500
LIMIT 100
```

Satisfying requirement: Ranged query and order by. CITY_ID was ordered in the table, and since Cassandra only allow the last restriction to be ranged, so we have to use '=' or 'in' to restrict before.

Potential use: In some case, companies may want to know how many record was made by using specific mobile id type in specific city with specific date and range.

Query 3

```
SELECT SEQ_NUM
FROM CDR
WHERE MOBILE_ID_TYPE IN (1,3,4,5) AND
(CITY_ID, MONTH_DAY, MSC_CODE, SERVICE_NODE_ID, SEQ_NUM) >
(1,2,10,5,100) AND
(CITY_ID, MONTH_DAY, MSC_CODE, SERVICE_NODE_ID, SEQ_NUM) <
(7,20,50,9,400)
LIMIT 100
```

Satisfying requirement: 10 conditions and range query. We use restriction to every primary key, and then we do not have to use 'ALLOW FILTERING'

Potential use: There might be some company want to know the sequence number of record which is under these specific conditions, it can make the searching more accurate. Then they can use the sequence number to look for more detail.

Query 4

```
SELECT GROUP_AND_COUNT(CITY_ID)
```

```
FROM CDR
WHERE MOBILE_ID_TYPE = 1 AND
CITY_ID IN (0,1,2) AND
(MONTH_DAY, MSC_CODE) >= (1,10) AND
(MONTH_DAY, MSC_CODE) <= (31,20)
```

Satisfying requirement: ranged and both group by and order by. We give range to city id, month_day and msc_code to reduce the range that need to be searched.

Potential use: some companies might want to know how many records of using specific mobile id type were made in their city under some conditions that they want to add.

Query 5

```
SELECT GROUP_AND_COUNT(MONTH_DAY)
FROM CDR
WHERE MOBILE_ID_TYPE = 2 AND
(CITY_ID, MONTH_DAY, MSC_CODE, SERVICE_NODE_ID, SEQ_NUM) >=
(1,1,30,7,100) AND
(CITY_ID, MONTH_DAY, MSC_CODE, SERVICE_NODE_ID, SEQ_NUM) <=
(1,31,50,9,200)
```

Satisfying requirement: ranged and both group by and order by and 10 condition. We use many condition to restrict the searching of the query, since it will take much more time if it is a count query. Limit does not work for counting. To avoid it go through the whole database, we reduce the range of every condition.

Potential use: It is always a good idea to know it got the most record in which day of the month under some other condition.

7. Experimental Results

Generation - Our generator took 15 hours to generate about 2 TB in total, which is 700 GB for each node. The average speed of generation is 40 MB/SECOND.

Insertion - The insertion was not as fast. We only inserted about 500 GB data.

Query - In our query 4 and query 5, we used both group by and order by, and it speeds up a lots comparing to the version we used before. And since we did not use ‘ALLOW FILTERING’ for these two queries, it took lesser time to go through data. Instead we use ‘MOBILE_ID_TYPE’ as the partition key, it groups all data by their MOBILE_ID_TYPE, 0,1, 2, …, 9.

Here is the result of each queries in different configuration:

Default Configuration	Our Configuration
Query 1 finished in 4.237885952 seconds.	Query 1 finished in 2.11831212044 seconds.
Query 2 finished in 0.794775009155 seconds.	Query 2 finished in 0.28423500061 seconds.
Query 3 finished in 1.74859213829 seconds.	Query 3 finished in 1.41924905777 seconds.
Query 4 finished in 5.93631505966 seconds.	Query 4 finished in 5.50283384323 seconds.
Query 5 finished in 2.10646605492 seconds.	Query 5 finished in 2.91639900208 seconds.

8. Discussion

Using partition key in queries can make the selection much more efficient. When we doing a selection, it can find the specific group that we need in a shorter period than normal ways. In normal way, it takes a lot of time to go through every node for choosing the specific data.

Using GROUP BY and ORDER BY can also speed up the speed of selection, but using ‘ALLOW FILTERING’ is not always a good idea for doing searching, it will take more time than usual way.

9. Conclusion

By working in this project, we learned some basic knowledge of NoSQL database system. We gained some hands-on experience by implementing the Cassandra database system and explore the differences between a tradition SQL and a NoSQL database system.

10. Future improvement

Even though we have achieve all requirements that we needed in this project, but there are still some aspect of implementation that can be improved:

- Speed of Generation - it might be able to speed up by using some efficient algorithm.

- Speed of Insertion - it might have connections with configuration of server or Cassandra itself. Using the sstableWriter library in Java might be a better choice for bulk data loading.

- Efficiency of retrieves - it might be able to improve by finding out a better configuration and table structure.

Reference

1. https://10kloc.wordpress.com/2012/12/27/cassandra-chapter-4-data-partitioning/
2. http://www.tanzirmusabbir.com/2013/06/cassandra-performance-tuning.html
3. http://shareitexploreit.blogspot.ca/2012/02/cassandra-bulk-loader.html
4. http://www.datastax.com/dev/blog/basic-rules-of-cassandra-data-modeling
5. http://christopher-batey.blogspot.ca/2015/05/cassandra-aggregates-min-max-avg-group.html
6. http://www.datastax.com/dev/blog/a-deep-look-to-the-cql-where-clause