Big Data Project Report

Shenwei Liao, Anthony Ou, Jacqueline Terlaan
CMPUT 391
Winter 2014

April 23, 2014

1 Goals and Objectives

- To successfully implement a distributed database system which will quickly and reliably store several terabytes of random data into large tables with many columns (attributes).
- To distribute the data in the above tables across 8 "nodes" (servers), which form our "cluster".
- To generate additional tables distributed throughout the cluster to be optimally configured for the queries to meet their own requirements (see below).
- To query the above tables such that all of the following requirements are met by five queries.
    1. Four of five queries must retrieve data across from all of the distributed nodes.
    2. At least one of the queries must contain at least ten atomic conditions (formulas) in the WHERE clause.
    3. At least two of the queries must utilize both the GROUP BY and ORDER BY clauses. Since Cassandra does not yet support GROUP BY aggregation, the same functionality must be achieved in another way.
    4. At least three of the five queries must be range queries which specify an upper and lower bound on some ordered attribute that is used for the purposes of querying.
    5. None of the five queries can be trivial. That is, there can be no simple key searches or anything that is of little practical interest for measuring how a database system can deal with computationally expensive requests.
- To record the execution time of each query.

2 Methodology, Tools and Equipment Used

8 server instances were used, each with 32GB virtual memory and 1TB secondary storage

3 Implementation

4 Shell Script

A shell script was written to wrap all functionality of the project, including server setup for all nodes, table creation and population and executing the queries. To run it, cd into the directory containing group3.sh, and call "sh group3.sh", and follow the prompts interactively.

4.1 Configuration Changes

Many different aspects of Cassandra cluster and the table were tuned so that we can achieve a high write availability and high performance for generating a randomly generated dataset. Here are some of the more notable configuration changes that we applied:

- Cluster-wide changes (in cassandra.yaml):
    1. Use of RoundRobinScheduler over NoScheduler. Having a request scheduler allowed for requests to be assigned to nodes faster by the coordinating node, speeding up concurrent reads and writes.
    2. Reducing commit log sync frequency. Writing to the commit log must happen for every transaction, as is necessary for write durability. For the purposes of this project, we can sacrifice some durability to go faster.
    3. Concurrent writes increased.
    4. Memtable flush writers increased. Memtable is the in-memory data structure cassandra uses to hold batched writes.
    5. Disabling automatic backups and snapshots of the keyspace.
    6. Removed the throttling on compaction throughput.
- Table-specific changes:
    1. Replication factor set to 3.
    2. Durable writes are disabled. The commit log doesn't need to be written to just for table generation. This and the next setting are only off because we are randomly generating data and there isn't such a thing as a wrong random value.
    3. Consistency level for inserts was set to 0 for maximum write availability.

4. Compression is disabled to keep CPU utilization low.
5. Removing column keys that had low cardinality.
6. Making indexes after the bulk insert.

4.2 Table Schemas

The cdr table follows the assignment specification's schema, with 470 columns. It and 16 column keys, indexed on MONTH_DAY and MOBILE_ID_TYPE. The role of the last two tables is to make the substitute aggregate queries both possible. These tables are not generated in a sparse manner, every column will have an entry of the appropriate type.

1. "CDR" The large 470 column family with 16 column keys and a UUID as the partition key. This helps with distributing across nodes when the partitioner is random or byte ordered.

2. "CDR_ALT" Contains the same 470 columns as cdr, but the last 6 column keys are no longer column keys. These 6 columns were never queried upon but in the CDR table for testing.

3. "SMALLCDR" This table has the last 235 columns removed but has the same 10 column keys as CDR_ALT, an experiment in partitioned table.

4. "GROUP_BY_MONTH"
    - Keys are between 1 and 31.
    - UUID of the partition key in CDR is kept as a column key
    - Reduces data to facilitate completion of a query equivalent to one in SQL containing 'GROUP BY MONTH_DAY'.
    - Ordering is implicitly done by Cassandra

5. "GROUP_BY_MOBILE_ID_TYPE"
    - Keys are between 0 and 7.
    - Again the UUID of the partition key in CDR is kept as a column key
    - Similar to "GROUP_BY_MONTH", but instead the assignment is based on its insertion number modulo 8. This helps us identify if insertions are consistent since this number should not variate much.
    - Ordering is implicitly done by Cassandra

6. Indexes An alternative to creating a table for grouping by.
    - On table CDR, column month_day
    - On table CDR, column mobile_id_type

4.3 Programs

We wrote two main Python 2.7 scripts: generate.py and query.py. These programs are meant to run in the background uninterrupted. The generate program is designed such that it can run many times as a different process with a different seed, so that data generation doesn't collide. A benefit is that now a generator process can be run on each node to maintain a balanced CPU load. Query.py runs the queries with facilities to measure real time spent running.

4.4 CQL Queries

1. Query 1

```
SELECT count (*) as ten_atomic
FROM ONE OF [CDR,CDR_ALT,SMALLCDR]
WHERE
 (MSC_CODE ,CITY_ID ,SERVICE_NODE_ID
 ,RUM_DATA_NUM ,DUP_SEQ_NUM ,SEIZ_CELL_NUM ,
     FLOW_DATA_INC ,SUB_HOME_INT_PRI ,CON_OHM_NUM,
 SESS_SFC)
> (10000,10000,10000,3,10000,1
            ,10000,10000,10000,10000)
AND (MSC_CODE ,CITY_ID ,SERVICE_NODE_ID
 ,RUM_DATA_NUM ,DUP_SEQ_NUM ,SEIZ_CELL_NUM ,
     FLOW_DATA_INC ,SUB_HOME_INT_PRI ,CON_OHM_NUM,
 SESS_SFC)
< (150000,150000,150000,30,150000,6
            ,150000,150000,150000,150000)
LIMIT 4000
ALLOW FILTERING ;
```

Cluster-wide range query with 10 atomic formulas. Searches on every column key of the CDR table, as this is the only conceivable way to properly execute a range query. Returns results from each node, satisfies the range query requirement and has 10 atomic formulas.

Potential uses: many 'outliers' are eliminated from the selection because they are above and below the minimum and maximum values of all the columns respectively, by approximately 10%. In other words, any 'extreme' data points in the CDR table can be ignored for statistical analysis.

Query 1 ('Alternate')

Performs the same cluster-wide query as Query 1, but over the table cdr_alt.

Query 1 ('Small')

This query is intended to take less time than the previous query, because fewer columns have to be read into memory in order for the atomic conditions to be evaluated.

2. Query 2

```
SELECT count (*) as range_city_id
FROM cdr
WHERE
MSC_CODE > 5000 AND MSC_CODE < 90000
LIMIT 4000
ALLOW FILTERING;
```

   Satisfies two requirements: cluster-wide and range query. Retrieves the number of cdr entries for cities with IDs strictly greater than 5000 and strictly less than 90000.

   Potential uses: In some cases companies may wish to know how many calls were made in a given range of cities, based on certain CITY_IDs. While those cities may not have any major features in common, this query can be used in cases where only a small sample of all cities is required.

3. Query 3

```
SELECT count(*) as range_DUP_SEQ_NUM
FROM cdr
WHERE
 (MSC_CODE  ,CITY_ID ,SERVICE_NODE_ID ,RUM_DATA_NUM  ,
     DUP_SEQ_NUM  )
 >(0 ,0 ,0 ,0 ,10000) AND
 (MSC_CODE  ,CITY_ID ,SERVICE_NODE_ID  ,RUM_DATA_NUM  ,
     DUP_SEQ_NUM)
 <(9900 ,9900 ,9900 ,9900 ,30000)
LIMIT 4000
ALLOW FILTERING;
```

   Satsifies two requirements: cluster-wide and range query. Counts the number of cdr entries such that DUP_SEQ_NUM is strictly greater than 30000 and strictly less than 300000. All other atomic conditions in the range query are used to ensure that all of the results fall within a valid range and have no null-valued entries for CITY_ID, SERVICE_NODE_ID, and so on.

   Potential uses: Suppose that statistical data is to be generated for rows in the CDR table for which there are no null-valued entries in the columns CITY_ID, SERVICE_NODE_ID, RUM_DATA_NUM, MONTH_DAY and DUP_SEQ_NUM, and that DUP_SEQ_NUM falls within a more specific range, that is 30000-300000.

4. Query 4

```
prep = session.prepare("select count(*) from 
    group_by_MOBILE_ID_TYPE where MOBILE_ID_TYPE = ?"
    )
prep.consistency_level = Consist_Level

for i in range (8):
    print str(i) + " : " , session.execute(prep.bind
        ([i]),timeout=None)
```

Satisfies two requirements: group-by and order-by clause, and cluster-wide query.

A table of UUIDs aggregates the keys of the MOBILE_ID_TYPE columns. This query has to be in a python loop to execute a query on each group.

Additionally this is a idempotent transaction, this insert can be done multiple times and it will not cause inconsistencies.

Inserting into the table is done in the following way:

```
insert into group_by_MOBILE_ID_TYPE
(MOBILE_ID_TYPE, id) values (?,?)
```

Potential uses: Most likely for the generation of histograms, but other basic kinds of statistical analysis can be performed on these small MOBILE_ID_TYPE 'buckets' as well, such as calculating arithmetic averages, etc.

5. Query 5

```
prep = session.prepare("select count(*) from 
    group_by_month where month_day = ?")
prep.consistency_level = Consist_Level
for i in range (1,32):
    print str(i) + " : " , session.execute(prep.bind
        ([i]), timeout=None)
```

Satisfies the group-by and order-by clause requirements. 'CLUSTERING ORDER BY (MONTH_DAY)' ensures that the GROUP_BY_MONTHs entries are ordered by MONTH_DAY, and GROUP BY functionality is achieved via aggregating UUIDs of each row in CDR.

Potential uses: finding out which day(s) of each month of the year have the highest frequency of entries added to the CDR table can be of use for marketing to customers, optimizing billing rates, and so on. Similar queries can be written for different times of the day, months or weeks of the year, by aggregating CDR row counts into hourly, monthly or weekly groupings for more specific results which could be of use in billing plans with calls at different times incurring different costs.

5 Experimental Results

5.1 Table Population

The biggest table to generate took two and a half days, generating an average of 380GB per node for a total of just over 3TB of data, out of the 8TB that was available on the cluster, with roughly 8.8 million columns. The average speed of insertions to the big table was roughly 15MB/s.

A separate keyspace was used for some postliminary tests. With the additional smallcdr table to mimic a sparse table or a partitioned table and cdr_alt table changed to sort ascendingly. Insertion of about 700000 entries — resulted in about 40GB of data per node which took 3 hours.

5.2 Query Results

When it comes to searching very large data sets like the 400GB node or smaller sized nodes with a potentially inconsistent read. Then quick reads generally came from maximizing column keys usage like in query 1 or minimizing column key usage like in query 2, this held true in both data sets.

Query 3 only did a range on half the column keys, and this happened to be the slowest possible query for the 400GB data set, while in the other set it became the quickest query overall with respect other queries at the same consistency level.

Doing consistent reads was simply slower, and queries that tend to use more column keys were generally faster. For the 40GB data set we experimented with a partitioned cdr table called the smallcdr table, this table led to the fastest and consistent query one.

The GROUP BY queries executed the fastest, because map reduction was already done at insertion time. On lower read consistency levels, queries on the group by tables had reduced query time.

	400GB CsistLvl=3	400GB CsistLvl=1
Query 1	3.032994934	2.200000453
Q. 1 Alt,desc	6.137618132	4.19432343245
Query 2	4.171009902	2.12728600105
Query 3	9.964161499	4.75602963368
Query 4	0.9116	0.6132
Query 5	0.8965	0.5538

Fig 1. Queries on initial keyspace with a 400GB load per node and a replication factor 3 with units in minutes. Queries 1-3 return 4000 results.

	40GB CsistLvl=3	40GB CsistLvl=1
Query 1	3.76711789767	1.394323281
Q. 1 Alt,asc	5.64526818196	4.377174052
Query 1 Small	2.46129718224	2.025322433
Query 2	4.20441953341	1.726343501
Query 3	1.50994411707	0.5942962646
Query 4	0.1503	0.1344
Query 5	0.1674	0.1395

Fig 2. Second keyspace made with cdr_alt sorted ascendingly and the smaller cdr table.

6 Discussion

What is not obvious from query one and derivatives, is as columns count increases, so should the number of column keys to keep query time from growing exponentially. We saw the move from 470 columns with 16 column keys in the cdr table to 470 columns with 10 column keys in cdr_alt, and this resulted in a longer query time by over a factor of two, when reducing the amount of column keys by over 50%.

In addition we saw the move from the cdr_alt table with 470 columns and 10 column keys to the smallcdr table with 235 columns and 10 column keys which reduced the query time by over 50%.

With this information in mind, if one wanted to denormalize their data and the data characteristics was not very sparse, it could be a valid strategy to put in dummy columns keys that is never sparse at the end of the column key (at the end because matters when declaring column keys). In addition we also saw how querying up until the middle of the column keys was incredibly quick or incredibly slow, if this could be consistently leveraged it would could half read time.

Searching on indexes made on the CDR column to facilitate a group by query could have been done, but it was obvious such a procedure would be incredibly slow. A procedure like this would be viable for redoing an entire schema.

7 Conclusions

The bulk insertion phase of the test had to be tuned in server side and in schema related ways, to maximize insertions. The most important server side changes that affected insert speed was using the round robin scheduler to have more nodes handle requests in parallel, to disable the commit log/durable writes in order to get more disk IO, and disabling compression of tables helped reduce CPU utilization allowing for more write threads. Schema configurations that were found to be bottlenecks were putting low carnality columns as column keys and making indexes after the bulk insertion.

There is no clear winner on having wide rows vs partitioned rows, either can be viable. A wider row was faster with inconsistent reads as opposed to a partitioned table, but this relation was inverted when it came to consistent reads. It comes down to the characteristics of the data and what makes sense for maximizing the applications of the data.

In general denormalizing the data is not viable without increasing column keys or ensuring the data was sparse. It might be reasonable to assume that wide sparse tables inherently achieves normalized schema characteristics since nulls take up no space in Cassandra, and this works in favor of a column oriented database. A guess as to why increasing column keys helps speed up queries, is that columns keys can help buffer earlier column keys and prevent Cassandra from reading a mix of column keys and regular columns. Of course that doesn't mean all column keys are good, it is possible a column key is polluting another column key because it is unrelated to the query. This explains how going into the middle of the column key list can be very slow for query 3 depending on the data, or very fast if the data is aligned correctly. The biggest variance in query times happens at middle of the column keys likely, because these columns are never empty and a slight misalignment can snowball into many more IOs.

8 Notes on Possible Improvements

Cassandra offers the ability of store entire tables into memory, using this feature on the group by tables would have likely led to a large speed up, and would have been interesting to explore, but a pure implementation of this would probably not be big data.

It is possible to run the server with a custom scheduling, so with this in mind it could be possible to write a scheduler that is optimized for bulk insertions. Cassandra also offers trigger support which could automatically reduce inserts into appropriate tables. These previous two features aren't well documented but could be useful in future works.

9 References Used

Agilent Technologies Inc (28 October, 2014), "Mobile Station Reported Pilot Information". http://wireless.agilent.com/rfcomms/refdocs/cdma2k/c2kla_gen_ms_pilot_meas_report.html Accessed: 10 March 2014.

Apache Software Foundation (2004), "The Apache Licence, Version 2.0". http://www.apache.org/licences/LICENCE-2.0 Accessed 7 March, 2014.

Apache Software Foundation (10 March 2014), "Cassandra Query Language (CQL) v3.1.5". http://cassandra.apache.org/doc/cql3/CQL.html Accessed: 10 March 2014.

Apache Software Foundation (15 November 2013), "UUID". http://wiki.apache.org/cassandra/UUID Accessed: 13 March 2014.

DataStax Documentation (2014), "Cassandra Storage Basics". http://www.datastax.com/documentation/cassandra/2.0/cassandra/dml/manage_dml_intro_c.html Accessed 7 March, 2014.

DataStax Documentation (2014), "The cassandra.yaml Configuration File". `http://www.datastax.com/documentation/cassandra/2.0/cassandra/configuration/configCassandra_yaml_r.html` Accessed 7 March, 2014.

DataStax Documentation (2014), "CLI keyspace and table storage configuration". `http://www.datastax.com/documentation/cassandra/2.0/cassandra/reference/referenceStorage_r.html` Accessed: 10 March 2014.

DataStax Inc. (2014), "UUID and timeuuid". `http://www.datastax.com/documentation/cql/3.0/cql/cql_reference/uuid_type_r.html` Accessed: 13 March 2014.

DataStax Documentation (2014), "What's new in Cassandra 2.0". `http://www.datastax.com/documentation/cassandra/2.0/cassandra/features/features_key_c.html` Accessed: 10 March 2014.

DataStax Documentation (2014), "The Write Path to Compaction". `http://www.datastax.com/documentation/cassandra/2.0/cassandra/dml/dml_write_path_c.html` Accessed 7 March, 2014.

Fowler, M. (9 January 2012), "NosqlDefinition". `http://martinfowler.com/bliki/NosqlDefinition.html` Accessed: 13 March 2014.

Fowler, M. (16 November 2011), "PolyglotPersistence". `http://martinfowler.com/bliki/PolyglotPersistence.html` Accessed: 13 March 2014.

Fowler, M. (7 January 2013), "Schemaless Data Structures". `http://martinfowler.com/tags/noSQL.html` Accessed: 14 March 2014.

Grinev, M. (9 July 2010), "A Quick Introduction to the Cassandra Data Model". `http://maxgrinev.com/2010/07/09/a-quick-introduction-to-the-cassandra-data-model/` Accessed: 10 March 2014.

Grinev, M. (12 July 2010), "Do You Really Need SQL to Do It All in Cassandra?". `http://maxgrinev.com/2010/07/12/do-you-really-need-sql-to-do-it-all-in-cassandra/` Accessed: 10 March 2014.

McFadin, P. (13 November 2012), "Cassandra Data Modeling Talk". `http://www.slideshare.net/patrickmcfadin/data-modeling-talk` Accessed: 13 March 2014.

McFadin, P. (3 May 2013), "The Data Model is Dead, Long Live the Data Model!". `http://www.slideshare.net/patrickmcfadin/the-data-model-is-dead-long-live-the-data-`
Accessed: 14 March 2014.

McFadin, P. (16 May 2013), "Become a Super Modeler". `http://www.slideshare.net/patrickmcfadin/become-a-super-modeler` Accessed: 14 March 2014.

McFadin, P. (24 June 2013), "The World's Next Top Data Model". `http://www.slideshare.net/patrickmcfadin/the-worlds-next-top-data-model` Accessed: 14 March 2014.

McFadin, P. (11 October 2013), "Cassandra 2.0: Better, Stronger Faster". `http://blog.imaginea.com/consistency-tuning-in-cassandra/` Accessed: 14 March 2014.

McFadin, P. (17 February 2014), "Time Series with Apache Cassandra". `http://www.slideshare.net/patrickmcfadin/time-series-with-apache-cassandra-strata` Accessed: 13 March 2014.

Oracle Inc. (Copyright 1996-2005), "Datatype Comparison Rules". http://docs.oracle.com/cd/B19306_01/server.102/b14200/sql_elements002.htm#i55214 Accessed: 10 March 2014.

Oracle Inc (No date), "GROUP BY clause". http://docs.oracle.com/javadb/10.6.1.0/ref/rrefsqlj32654.html Accessed 7 March, 2014.

Oracle Inc (No date), "ORDER BY clause". http://docs.oracle.com/javadb/10.6.1.0/ref/rrefsqlj13658.html Accessed: 10 March 2014.

Oracle Inc (No date), "SelectExpression". http://docs.oracle.com/javadb/10.6.1.0/ref/rrefselectexpression.html#rrefselectexpression Accessed: 10 March 2014.

Oracle Inc. (Copyright 1996-2005), "SYSDATE". http://docs.oracle.com/cd/B19306_01/server.102/b14200/functions172.htm Accessed: 10 March 2014.

Oracle Inc. (Copyright 1996-2005), "SYSTIMESTAMP". http://docs.oracle.com/cd/B19306_01/server.102/b14200/functions173.htm Accessed: 10 March 2014.

Oracle Inc. (Copyright 1996-2005), "TO_DATE". http://docs.oracle.com/cd/B19306_01/server.102/b14200/functions183.htm Accessed: 10 March 2014.

Saxena, S. (4 September 2013), "Consistency Tuning In Cassandra". http://blog.imaginea.com/consistency-tuning-in-cassandra/ Accessed: 14 March 2014.