Project 2: Cassandra

Mahsa Rezaei Firuzkuhi
mrezaeif@cougarnet.uh.edu

Problem Description

In part one, we are utilizing open source, distributed, NoSQL database named Cassandra in order to compare its performance with utilizing the CSV file format for our pyspark application from homework 3. The CSV file format is suboptimal in many aspects. So, another goal is to compare it with Cassandra in terms of timing. We use the same dataset as project one, the “2013_data_jan.csv” and we need to import the January dataset into the Cassandra NoSQL database. So the fundamental goal is to import the dataset into the Cassandra and then in part 2 of the assignment to compare the performance of the pyspark code that we have developed for homework 3 using the CSV input file versus the Cassandra database while having 1, 2,4 and 8 number of executors in part 2 of the assignment. We need to consider how to import data from the CSV file into the Cassandra database properly to achieve the best performance and compare it with the case of reading from csv file. In this problem we have a large sensor data set in the comma separated value (CSV) format in which each sensor is reporting a measurement every five minutes for various concentration parameters of the pollutants and we need to identify the hourly average of only one of the concentrations (O3) based on the input data set that we have. This problem is challenging because the dataset consists of actual data collected from the sensors and there is many data points or entries that cannot be used for information processing and on the other hand there are various data point values that are missing (the sensor data was not collected every 5 minutes). Every single data has a flag indicating the availability of the data point value. Finding the actual hourly average of each concentration on each site is a highly challenging task because we are interested only in O3 concentration hourly average value information, and we are not interested in any other pollutant or any other measurement. We need to identify which values we can use to calculate the hourly average and then performing the actual calculation as we did this part in homework 3. If we need to calculate the hourly average of a specific concentration for every hour of a year in Pyspark code, the key should contain month, day and hour values corresponding and the measurement after ensures that the reducer will get all the data points for each of these intermediary keys (this part is completed in homework3). I have considered the following two goals for my data modeling when importing the data into the Cassandra table: spreading data evenly around the cluster and minimizing the number of partitions to be read. So, we need to pick a good primary key for the Cassandra table and when we issue a read query, we have to read rows from as few partitions as possible and as a result find the best trade-off between these two goal and then compare the best achieved performance with performance we get from the pySpark code which reads from the CSV dataset file format. So, we used Spark technology to be able to do the operation in parallel form. In the following section we briefly introduce the spark technology and data-frame and Cassandra NoSQL database and how we employ their features in order to speed up the hourly average calculation of O3 concentration.

Solution Strategy

Apache Spark is an open-source cluster-computing framework. Spark provides an interface for programming entire clusters with implicit data parallelism and fault-tolerance. Apache Spark provides programmers with an application programming interface centered on a data structure called the resilient distributed dataset (RDD), a read only multi-set of data items distributed over a cluster of machines, that is maintained in a fault-tolerant way. Spark uses shared memory to facilitate communication between resources and this fact makes it faster compared to other technologies that do not use shared memory like Hadoop. Data-Frames are distributed collection of rows under named columns and conceptually like a table in a relational database.

The design goal of Cassandra is to handle big data workloads across multiple nodes without any single point of failure. Cassandra has peer-to-peer distributed system across its nodes, and data is distributed among all the nodes in a cluster. Keyspace is the outermost container for data in Cassandra which is a list of one or more column families. A column family, in turn, is a container of a collection of rows. Each row contains ordered columns. Column families represent the structure of the data. Each keyspace has at least one and often many column families.

A schema in a relational database model is fixed. Once we define certain columns for a table, while inserting data, in every row all the columns must be filled at least with a null value. In Cassandra, although the column families are defined, the columns are not. We can freely add any column to any column family at any time. Generally, column families are stored on disk in individual files. Therefore, to optimize performance, it is important to keep columns that we are likely to query together in the same column family. Another difference is that relational tables define only column, and the user fills in the table with values. But, In Cassandra, a table contains columns, or can be defined as a super column family.

Cassandra is optimized for high write throughput, and almost all writes are equally efficient. We need to perform trade-off between extra writes and the efficiency of the read queries. I have considered the following two goals for my data model:

1. Spread data evenly around the cluster: we need that every node in the cluster to have roughly the same amount of data. In Cassandra, rows are spread around the cluster based on a hash of the partition key, which is the first element of the PRIMARY KEY. So, I needed to pick a good primary key.

2. Minimize the number of partitions read: partitions are groups of rows that share the same partition key. The goal in a read query is to read rows from as few partitions as possible. Since each partition may reside on a different node, the coordinator will generally need to issue separate commands to separate nodes for each partition we request. This adds a lot of overhead and increases the variation in latency. Furthermore, even on a single node, it's more expensive to read from multiple partitions than from one due to the way rows are stored. These two goals often conflict, so we need to try to balance them.

3. Determining what type of query, we need to support, and model based on that query patterns. Following are some example of query patterns:

    - Grouping by an attribute
    - Ordering by an attribute
    - Filtering based on some set of conditions
    - Enforcing uniqueness in the result set

    In our problem since we need to calculate the hourly average of O3, so the query is to have access to a group of pollutant data points which are in groups and we want to get all those data points in a group (in other words, we want to get the full pollutant info for every pollutant in a particular group).

4. In practice, we should use one table per query pattern. And if we need to support multiple query patterns, we need more than one table. So here we designed one table for our query pattern which is kind of grouping by an attribute. In other word, if we store the data in the table in the order that we have plan to use them afterwards, data access would be very fast.
5. In Cassandra disk space is considered cheap and if we have a certain type of analysis (query pattern) that you need to run over and over, it is better to store the data twice for different type of analysis and using two different set of primary key/partition key instead of storing the data only one and degrade the performance

In the following command, the PRIMARY KEY has 7 components: "year", which is the partitioning key, and the rest which are called the clustering keys (non-partition keys or column grouping keys because they will indicate the order in which the elements that have the same partition key are being store on a server node). With the following combination of the primary key columns, I achieved the best performance and correct results. The non-partition keys are used for column grouping and the following elements are the columns we used for column grouping in homework 3.

```
PRIMARY KEY(year, month, day, hour, min, cams, parid) )
```

This satisfies the goal of minimizing the number of partitions that are read, because we only need to read one partition. However, we need to check if it does well with the first goal of evenly spreading data around the cluster or not. If we have thousands or millions of small groups with hundreds of datapoints in each, we will get an even spread. In our case, we have small groups of pollutants records in one hour which we need to collect and calculate the average of those, so with selecting the above primary key, we get an even spread.

But if there is one group only with millions of datapoints in it, the entire burden will be shouldered by one node. In that case, we need to spread the load more evenly using a few strategies. The basic technique is to add another column to the PRIMARY KEY to form a compound partition key.

Every single data point of the data set contains 12 columns: year, month, day, hour, and minute of when the measurement was performed, and measurements has been taken every five minutes by sensors. Some of the measurement parameters like “region” are not useful in calculation the hourly average of the data in each site. Then we have the parameter name and identification which we are interested in to identify the name “o3” only. The next two columns are the location of the sensor (site) and sensor ID which we are interested in because a single site can have potential multiple sensors. The last two value are the most important columns including the actual sensor measurement value (which we are going to evaluate the average of that in every hour) and the flag which indicates whether that sensor value is valid or not. If the flag is empty (null) it indicates the value is valid otherwise that data point should be discarded and not considered in calculation of the average. The solution for solving this problem is described in homework 3. We need to implement a Pyspark code solution which can calculate the hourly average of O3 value for each “site”. In the first step we read all the single data points for all pollutants in one year and in different sites in a data-frame structure by following command and then filter the data frame with O3 concentrations:

```
Df=sqlContext.read.format('com.databricks.spark.csv').options(delimiter=',',n
ullValue='').load("/home2/input/2013_data_jan.csv",schema=schema)
```

Description of how to run the code

“hw4.py” is answering both parts of the assignment. To change the input and output dataset directories we need to change those values inside the source code. All other parameters can be set using the following command. For example, we must change the number of executors, it means changing the value of total-executor-cores in the spark-submit command (we consider the value 2 for the number of executors per core in all the measurements):

```
spark-submi --packages com.datastax.spark:spark-cassandra-connector_2.11:2.3.0

--master  spark://crill:28959  --total-executor-cores  8  --executor-cores  2
./hw4.py
```

For 1 executor we insert --total-executor-cores 2

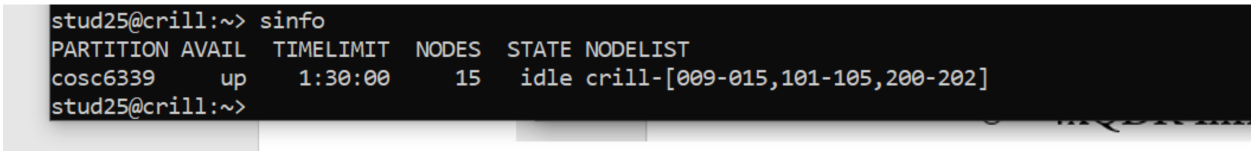
For 2 executor we insert --total-executor-cores 4

For 4 executor we insert --total-executor-cores 8

Results

Description of resources used

We have utilized 15 compute nodes of the “Crill” cluster for our project. The properties of these nodes are listed below:



```
stud25@crill:~> sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
cosc6339     up    1:30:00     15   idle crill-[009-015,101-105,200-202]
stud25@crill:~>
```

2 Appro 1326G4 nodes (crill-101 - crill-102), each node with

- two 2.2 GHz 12-core AMD Opteron processor (24 cores total)
- 32 GB main memory
- four NVIDIA Tesla M2050 GPUs (448 cores each) (not operational as of today)
- 4xQDR InfiniBand HCA

3 Sun Microsystems X2200 nodes (crill-103 - crill-105), each node with

- two 2.2 GHz 4-core AMD Opteron processor (8 cores total)
- 32 GB main memory
- 4xQDR InfiniBand HCA

3 HP DL 160 Gen 8 nodes (crill-200 - crill-202), each node with

- two 2.4 GHz quad-core Intel Xeon E5-2665 processor (12 cores/24 threads total)
- 16 GB main memory
- 4xQDR InfiniBand HCA

Network Interconnect

- 288 port 4xInfiniBand QDR InfiniPath Switch (donation by TOTAL)
- 24 port 4xInfiniBand SDR switch (I/O switch to the SSD storage)
- 48 port Netgear GE switch

Storage

- 4 TB NSF /home directory
- 18 TB distributed storage operated as OrangeFS (/pvfs2) or BeeGFS (/beegfs)
- 2 TB RamSan 620 SSD storage (/pvfs2-ssd) (not operational)

Description of measurements performed

We imported the time library and used time.time() to clock the execution time. In the following three section we will describe the details of the measurements. We present the average of the four different running of the code in the following tables. The sections one and two of the assignment is about doing the performance evaluation and changing the various parameters of the code.

Part 1:

I started cqlsh session using the command cqlsh as shown below. It gives the Cassandra cqlsh prompt as output. We need to first load the data in the Cassandra table. I have opened a cqlsh shell with the following command and since there was already a keyspace for my account (stud25), I have started using it with below commands. Assignment of server for connecting to Cassandra for student id stud25 is 192.168.1.105.

```
stud25@crill:~> cqlsh 192.168.1.105 -u stud25 -p stud25
```

CQL Data Definition Commands: Cqlsh has a few commands that allow users to interact with it. The commands are listed below. Then I created table named “project-2-table” using below commands.

USE − Connects to a created KeySpace.

CREATE TABLE − Creates a table in a KeySpace.

```
stud25@cqlsh>  use stud25;
```

```
stud25@cqlsh:stud25>  CREATE TABLE project-2_table (year int, month int, day
int, hour int, min int, region int, parid int, parname text, site int, cams
text , value text, flag text, PRIMARY KEY(year, month, day, hour, min, cams,
parid) );
```

The primary key is the column that is used to uniquely identify a row. Therefore, defining a primary key is mandatory while creating a table. A primary key is made of one or more columns of a table.

COPY: this command copies data from a file to Cassandra table which we already created. Loaded data from file "2013_data_jan.csv" to table "project-2_table" using below command:

```
stud25@cqlsh:stud25> COPY project-2_table(year, month, day, hour, min,
region, parid, parname, site, cams, value, flag) FROM
'/home2/input/2013_data_jan.csv' WITH HEADER=FALSE;
```

Verification: the select statement will give us the schema. I verify the table using the select statement as shown below.

```
stud25@cqlsh:stud25> SELECT * FROM project-2_table;
```

Cassandra Connector: Spark Cassandra Connector allows to pull data from Cassandra to Spark and do the analytics from the data. We can load a huge chunk of data (or entire table) from Cassandra and perform all sorts of complex transformations, aggregations.

```
spark-submit
--packages com.datastax.spark:spark-cassandra-connector_2.11:2.3.0
--master spark://crill:28959
--total-executor-cores 8
--executor-cores 2 ./hw4.py
```

Spark session: next I need to create SparkSession with my Spark application. When creating the session, I'm specifying Cassandra host and keyspace as configuration parameter. Starting Apache

Spark 2.0.0 the entry point to Spark Job is changed from SparkContext to SparkSession. SparkSession gives combined customization options of SparkConext and SparkSQL.

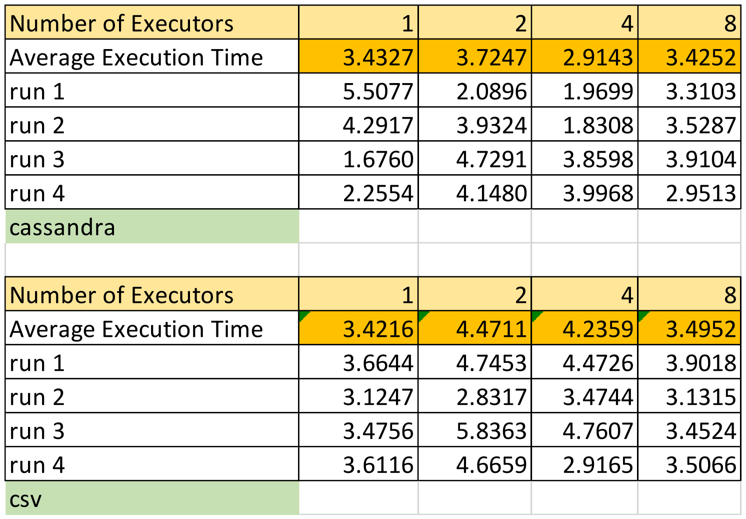
```
spark=SparkSession.builder
.config('spark.cassandra.connection.host','192.168.1.105')
.config("spark.cassandra.auth.username",'stud25')
.config("spark.cassandra.auth.password",'stud25')
.getOrCreate()
```

Load DataFrame: the data on Cassandra table can be loaded as DataFrame as well. Then we can do the operations as normal DataFrame.

```
df=spark.read.format('org.apache.spark.sql.cassandra').load(table="project-
2_table", keyspace="stud25")
```

Part 2:

We need to change the number of executors that we are using for measurements. We consider January database file and running the Spark code with one, two, four and eight number of executors and evaluate how does the performance will change for both cases of reading the data-frame from a CSV file and from Cassandra database table. We consider two cores per executors and keep this value fix in all the measurements. We expect that the more Spark executors we are using, the smaller the execution time becomes. This performance improvement is dependent on the extra hardware employment for execution. We have measured the execution time and provide the results in the following table for different number of executors and for each number of executors, we ran the code four times and provide the average execution time. On the other hand, we are comparing the execution time of the O3 hourly average calculation code while reading the dataset from CSV file formats and loading data-fame from the Cassandra database for the same number of executor or even different number of executors.



Number of Executors	1	2	4	8
Average Execution Time	3.4327	3.7247	2.9143	3.4252
run 1	5.5077	2.0896	1.9699	3.3103
run 2	4.2917	3.9324	1.8308	3.5287
run 3	1.6760	4.7291	3.8598	3.9104
run 4	2.2554	4.1480	3.9968	2.9513
cassandra				

Number of Executors	1	2	4	8
Average Execution Time	3.4216	4.4711	4.2359	3.4952
run 1	3.6644	4.7453	4.4726	3.9018
run 2	3.1247	2.8317	3.4744	3.1315
run 3	3.4756	5.8363	4.7607	3.4524
run 4	3.6116	4.6659	2.9165	3.5066
csv				

Table 1: Average execution time for different number of executors for January datasets

Finding:

Result shows that, on average, in case of Cassandra, increasing the number of executors from 1 to 2 and then to 4 executors, is slightly decreasing the execution time (very small amount) and increasing the executors from 4 to 8 had slightly negative effect on the execution time. We need to consider that increasing number of executors reduces amount of time spent on each node but, it will increase shuffling overhead so increasing number of executors is not helpful in all the cases. In our example, the amount of input data was not big enough to utilize extra computing power available with higher number of executors. One important way to increase parallelism of spark processing is to increase the number of executors on the cluster. However, knowing how the data should be distributed, so that the cluster can process data efficiently is extremely important.

As we can observe, on average, in case of CSV input file, increasing the number of executors from 1 to 2 and then to 4 and 8 executors, has zero or negative effect on the execution time (very small amount).

On the other hand if we compare the execution time of the code in case of the Cassandra and CSV for a single number of executors (for one executor), on average, the execution time did not improved while with increasing the number of executors (for example 2 or 4), the execution time in case of Cassandra database has improved significantly in comparison to CSV, which indicates the successful assignment of the primary key set of the columns of the January dataset into the Cassandra table and as a consequence, improvement in performance. In other word, if we store the data in the table in the order that we have plan to use them afterwards, data access would be very fast. Also, we observe that this performance improvement has increased with higher number of executors. In other words, in case of 4 executors, performance improvement of Cassandra relative to CSV is higher than the case in which the number of executors is 2 or one. But, in case of number of executors to be 8, on average, we do not observe any significant performance improvement from Cassandra over the CSV. In our case, we have small groups of pollutants records in one hour which we need to collect and calculate the average of those so higher than 4 executors do not help to increase the performance through increasing number of tasks (more executors means more tasks and each task is executed in parallel).