A Method of SQL Processing Data in NoSQL

Wumin Pan
North China Institute of Science and Technology
Beijing, China
e-mail: aquarius@163.com

***Abstract*—For data processing, there has been lot of work in that area. Most of the work intends to provide a simple or conducive way of processing (storing and accessing) the data stored in NoSQL stores. They aim to have a SQL like approach in manipulating the data, since most end users might find that way to be familiar. We will stress more on the the advance SQL features that has been implemented and find a method to provide SQL-like commands to manipulate this data .**

Keywords-SQL implementation, NoSQL, MapReduce.

I. INTRODUCTION

This With the advent of "BigData" in recent age, current and potential application might find Relational Database Systems not suitable for storing and accessing their data. This might be due to its unstructured nature, size or the need for high scalability and availability. These requirements makes it paramount to look into alternative ways of storing and processing this data.

In this their work entails processing structured data in NoSQL using MapReduce. So basically RDBMS is transformed in to a NoSQL structure and they manipulate the data using series of MapReduce functions which can further be integrated into some framework to provide SQL like query capabilities.

We will dwell on the major optimization that we have done regarding Cross-Join and SET Operations. It will also cover the JackHare framework that enabled us to use normal ANSI SQL Queries to process the data stored in the NoSQL data store. This work focuses mostly on the back-end component of this framework which can be integrated with an SQL client called Squirrel to act as the front-end. Our main contribution is in the back-end is providing our own MapReduce codes to solve different ANSI-SQL Queries.

II. ADVANCED SQL IMPLEMENTATION

Wherever Times is specified, Times Roman or Times New Roman may be used. If neither is available on your word processor, please use the font closest in appearance to Times. Avoid using bit-mapped fonts if possible. True-Type 1 or Open Type fonts are preferred. Please embed symbol fonts, as well, for math, etc.

III. STRUCTURE

A. *JackHare and our Work*

Firs JackHare is an OpenSource project which aims to provide an ANSI-SQL compliant JDBC API implementations for most useful operations on noSQL DB. At the time of writing this masters thesis the NoSQL DB supported by this platform is HBase-0.90.4 and Hadoop-0.20.203. So with the architecture of JackHare we can write and issue custom MapReduce programs and HBase calls to process our BigData stored in our NoSQL stores. With MapReduce platform we can make the best use of parallelism to process really huge data.

One of the major changes we have done is to completely parallelize the ANSI-SQL queries that we are working on; instead of using just the native HBase API calls as in the original JackHare implementation. JackHare has various components which plays different independent roles as illustrated in the Fig1 The figure illustrate JackHare’s high level architecture. There are two main parts and that is Front End and Back End. The front end consist of an SQL Client, SQL Translator and

Database Connection and the back end consist of NoSQL database (HBase) and API/MapReduce calls to to implement SQL behaviour.

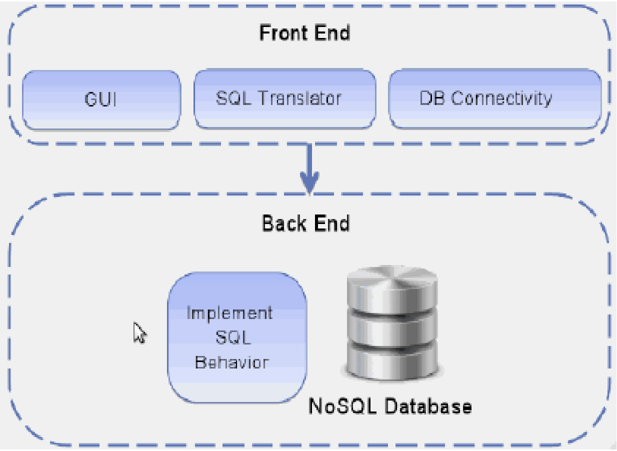


Figure .1: High Level JackHare Architecture

B. Advance MapReduce to SQL Implementation

Majority of our queries are advance queries, which involves one, two or sometimes three MapReduce jobs. Mostly the more the jobs the more complicated the implementation. During optimization a reduction in these jobs can significantly increase performance.

Order By is one of the most challenging problem we have faced during the implementation of this work. The main reason being we are working with different data types which are stored differently in HBase. Integers, Strings and floating point numbers are stored in different ways in HBase. Therefore you must be cautious of which data type you are dealing with for a proper order by result. Fig.2 shows all the improvements for the ORDER BY clause.

We will do a multi column order by with different data types. We will go through our approach in solving these problems. The key to our approach is building our RowKeys out of our the order by columns, since we know if the RowKey are sorted we will have a sorted values too.

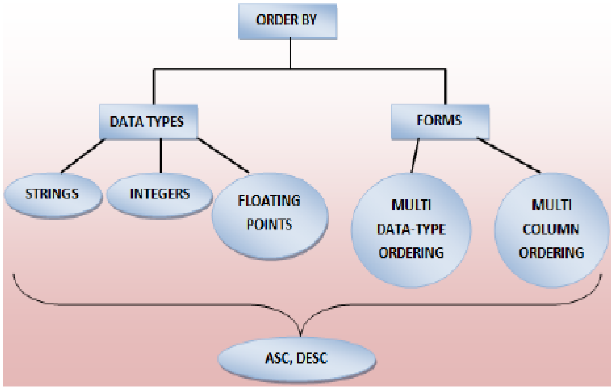
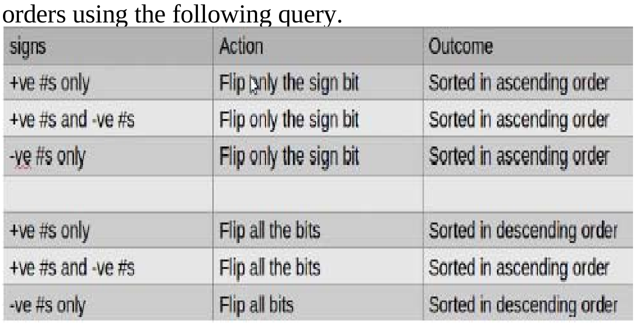


Figure.2: Major improvement in ORDER BY clause

C. Descending and Ascending Ordering

By default HBase stores string and integers in ascending order. But we can change this by simply flipping all the bits in the byte array. Fig.3 is a table summarizing the above explanations. For Descending and Ascending, consider in an example where you want to sort student marks in descending orders using the following query.



signs	Action	Outcome
+ve #s only	Flip only the sign bit	Sorted in ascending order
+ve #s and -ve #s	Flip only the sign bit	Sorted in ascending order
-ve #s only	Flip only the sign bit	Sorted in ascending order
+ve #s only	Flip all the bits	Sorted in descending order
+ve #s and -ve #s	Flip all the bits	Sorted in ascending order
-ve #s only	Flip all bits	Sorted in descending order

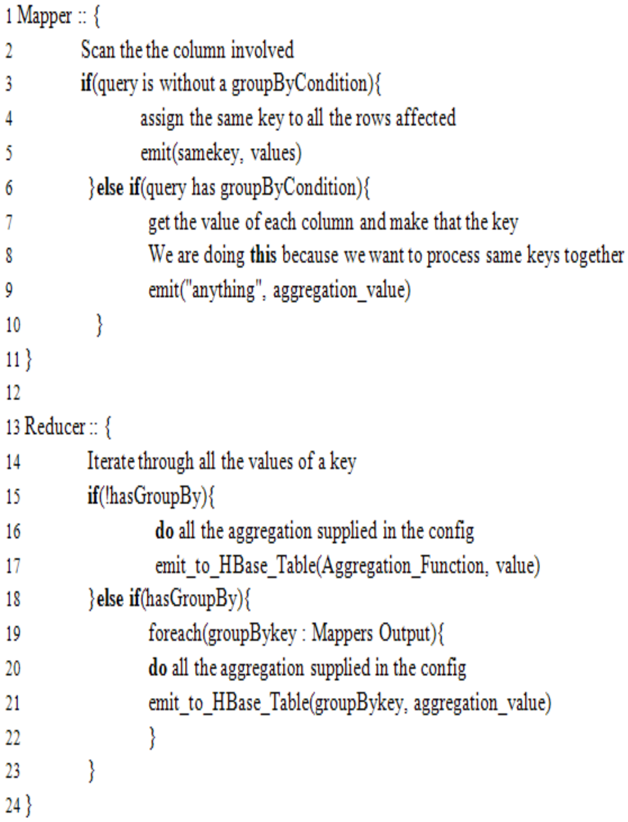
Figure. 3: Order By Summary +ve = Positive Numbers, -ve = Negative Numbers

*Select * FROM STUDENT_MARKS ORDER BY MARKS DESC*

HBase sorts by the RowKey so a manipulation of the RowKeys by shifting the bits can help us sort in either ascending or descending order. We have implemented several aggregation functions which includes SUM, COUNT,MIN, MAX, AVG and some statistical functions like Standard Deviation, Variance. These implementations are done using a mapper and a reducer. Consider the Query:

Select [OPTIONAL GROUP BY COL], SUM(COL1), COUNT(COL1), MIN(COL1), MAX(COL1),STD(COL1) FROM TABLE [OPTIONAL: (WHERE CONDITION) (GROUP BY CONDITION)]

In Mapper and Reducer we do the following.



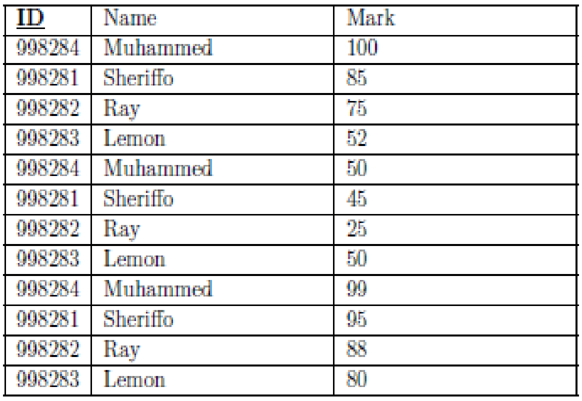
```
Mapper :: {
        Scan the the column involved
        if(query is without a groupByCondition){
                assign the same key to all the rows affected
                emit(samekey, values)
        }else if(query has groupByCondition){
                get the value of each column and make that the key
                We are doing this because we want to process same keys together
                emit("anything", aggregation_value)
        }
}

Reducer :: {
        Iterate through all the values of a key
        if(!hasGroupBy){
                do all the aggregation supplied in the config
                emit_to_HBase_Table(Aggregation_Function, value)
        }else if(hasGroupBy){
                foreach(groupBykey : Mappers Output){
                do all the aggregation supplied in the config
                emit_to_HBase_Table(groupBykey, aggregation_value)
                }
        }
}
```

Figure. 4 Aggregation Functions

A classical example will be accumulating the sum of student test marks in a course as depicted in Fig. 5 using the following query

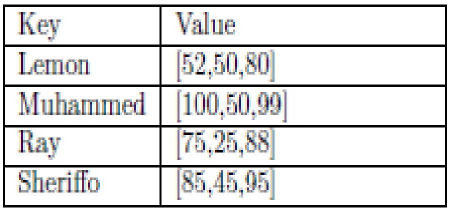
Select ID,Name,SUM(Mark) FROM TEST_RECORDS GROUP BY Name .

The function of the mapper will set the GROUP BY column as the new RowKey this will enable the reduce to process or aggregate one student mark together. The data passed to the reducer will be in key-value format as in Fig. 6 Finally the reducer will just iterate through the keys and add the marks together as the final output for each student. This can also work with DISTINCT.



ID	Name	Mark
998284	Muhammed	100
998281	Sheriffo	85
998282	Ray	75
998283	Lemon	52
998284	Muhammed	50
998281	Sheriffo	45
998282	Ray	25
998283	Lemon	50
998284	Muhammed	99
998281	Sheriffo	95
998282	Ray	88
998283	Lemon	80

Figure.5 Test Records



Key	Value
Lemon	[52,50,80]
Muhammed	[100,50,99]
Ray	[75,25,88]
Sheriffo	[85,45,95]

Figure. 6 Mapper Output

D. *JOIN Implementation and Cross Join Optimization*

Joins are one of the most researched areas in NoSQL, because NoSQL was not designed handle join operations. Nonetheless we can do our own joins but since we might be dealing with huge data this process might be very expensive and time consuming.

Our approach involves using cell versions to avoid cross joins for Equal-Join. We have a considerably great performance improvement with this mapping model. Our implementation follows the steps below:

Proper table design: HBase has no strict schema requirements like in RDBMS but following best practices can lead to improve performance. Looking at our schema in Figure 2.5 you will see, we have duplicated data and stored related data together using one RowKey; to achieve faster data access. Avoid Cross Product: We avoid cross product which serves as a perquisite for most inner joins. Doing a cross product is very expensive as you have to read each key from one table and merge its data with all the keys in table two. E.g. if we have 10,000 rows in table A and 1,000,000 rows in table B to build B x A, we would need 10,000 x 1,000,000 the output table of the cross table will be 10 billion rows; in fact in reality we store much more data in NoSQL DB that the example above.

E. *Set operations and it's Optimization*

The Finally we have also implemented some set operations, e.g. UNION, INTEREST, ALL. For UNION and INTERSECT we used two ways to solve them, one is for the general problem where we prawn a separate job for each of the queries, store the results in HBase and read those results again and store them in the final results table. This will take us minimum three MapReduce jobs. The second one which is not for general UNION or INTERSECT cases depends on the format of the where clause. Consider the example.

SELECT NAME FROM EMPLOYEE WHERE ID > 7
UNION
SELECT NAME FROM EMPLOYEE WHERE ID< 100

With such queries we can get great performance improvement by first modifying the query and instead of running 3 or more MR jobs we can just run one job to get our final result. In our implementation we can simply use HBase filters to filter ID > 7 and ID < 100 in a single job. This approach will also work if the comparators (>, <) are the same a UNION or JOIN query.

ACKNOWLEDGMENT

Covered are the advance implementations of the queries that we solved and also optimization that have been done for specific kind of queries. Also explained is the role JackHare played in the implementation by providing us with the capability to use SQL client like Squirrel to query our data stored in HBase.

REFERENCES

[1] Fey Chang, Jeffrey Dean et al. *BigTable:* A Distributed Storage System for Structured Data

[2] *HBase:* http://hbase.apache.org/

[3] *NoSQL:* http://nosql-database.org/ [10] Meng-Ju Hsieh, et al. *SQLMR:* A Scalable Database Management System for Cloud Computing

[4] K GIGACOM 2012 *Facebook Data:* http://gigaom.com/data/facebook-iscollecting-your-data-500-terabytes-a-day/

[5] Biswapesh, et al. *Tenzing:* A SQL implementation On The MapReduce Framework.

[6] *Number of Column Family recommendations:*http://hbase.apache.org/book/number.of.cfs.html

[7] *JackHare:* http://sourceforge.net/projects/jackhare/.

[8] D *Hive:*https://cwiki.apache.org/confluence/display/Hive/Home