International Journal of Computational Intelligence Research
ISSN 0973-1873 Volume 13, Number 2 (2017), pp. 161-173

http://www.ripublication.com

Top-k Equities Pricing Search in the Large Historical data set of NYSE

Ganesan Senthilvel
Research Scholar,
PG & Research Dept. of Computer Science,
Dr. Ambedkar Arts College, Chennai, India

A. Murugan
Associate Professor,
PG & Research Dept. of Computer Science,
Dr. Ambedkar Arts College, Chennai, India

Abstract

Stock exchange is a stock market, where brokers and traders can buy and sell stocks. Stock exchange often functions as continuous auction markets. In this paper, we describe the need to look into large data for analyzing the recorded financial activities in the stock exchange. The study of historic data is very crucial. As the size of the data huge we need better methods to retrieve data efficiently. Hadoop framework is leveraged to resolve this problem.

Source data is ingested from NYSE (New York Stock Exchange) web site into Hadoop file system. Perceived historical data is queried by the user with Top-k stock details based on its closing price. We are using two techniques of querying, Hive and Impala. Though the technical effort is completely built using the existing framework like Hadoop, Hive, Impala, etc, Top-k query is constructed. Top-k query is developed using in Java to extract the expected result set.

The objective is to compare and analyse the execution and its performance of Top-k Big Data query between disk based Hive and memory based Impala.

Keywords: Stock Market, New York Stock Exchange, Equities, Top-k pricing.

1. INTRODUCTION

Stock market is a place where brokers and traders can buy and sell stocks. Stock exchange often functions as continuous auction markets. There are ups and downs in the stock market. There are possibilities for an investor to experience losses due to many factors. We can reduce the loss by studying historic data prior to investing. It is not enough to just look at a stock's volatility from day by day.

2. PROBLEM STATEMENT

The problem statement pertains to the area of Top-k pricing search from the equity's historical stock exchange data streams. In this case, the popular New York Stock Exchange (NYSE) data has been considered.

2.1 Definition

On analyzing the stock exchange site, Top-k pricing search on equity market is not around. Though, other external sites tried to produce Top-k in terms of activity not by pricing [10]. In this scenario, the paper discussed how to retrieve Top-k pricing result set based on user driven date.

2.2 Data Layout

Daily data feed is shipped in the various data formats; and the target is CSV file. CSV (for historical stock data) file layout is as follows:

Table 1. Sample NYSE Equity

Ticker	Date	Open	High	Low	Close	Volume
IBM	2015 0728	159.91	160.19	158.50	160.05	272100

2.3 Equity

In general, the definition of equity can be represented with the accounting equation:

Equity = Assets - Liabilities

An equity investment generally refers to the buying and holding of shares of stock on a stock market by individuals and firms in anticipation of income from dividends and capital gains, as the value of the stock rises. Typically, equity holders receive voting rights for the board of directors' selection.

3. PROPOSSED WORK

In this paper, two areas are addressed (1) Top-k equity pricing search @ business domain (2) disk based versus memory based @ technology domain. On analyzing the stock exchange site(s), Top-k pricing search on equity market is not around and so it is kind of new entry. In terms disk vs memory based Hadoop frameworks (hive vs impala), there are few industry white papers available to consume.

4. HADOOP FRAMEWORK

In the fundamentals of the computing, two key factors drive the system, namely (1) storage and (2) process. In Hadoop eco system, HDFS represents storage and MapReduce for process. Hadoop is a framework written in Java for running applications on large clusters of commodity hardware and incorporates features similar to those of the Google File System (GFS) and of the Map Reduce computing paradigm [1].

Hadoop's HDFS [1] is a highly fault-tolerant distributed file system and, like Hadoop in general, designed to be deployed on low-cost hardware. It provides high throughput access to application data. Also, it is suitable for applications that have large data sets. HDFS architecture is represented in the Figure 1.

In the HDFS architecture, NameNode resides one per cluster. Meta data's centralized server to provide a global picture of the file system's state. NameNode stores its file system metadata on local file system disks in a few different files, but the two most important of which are fsimage and edits [1]. DataNode is designed to be multiple per cluster, which stores block data (contents of files). Data nodes regularly report their status to the NameNode in a heartbeat mode with the default setting of 3 minutes. It sends Block Report to NameNode periodically with the default of 60 minutes. Block report is the list of all usable blocks of DataNode disks [1].

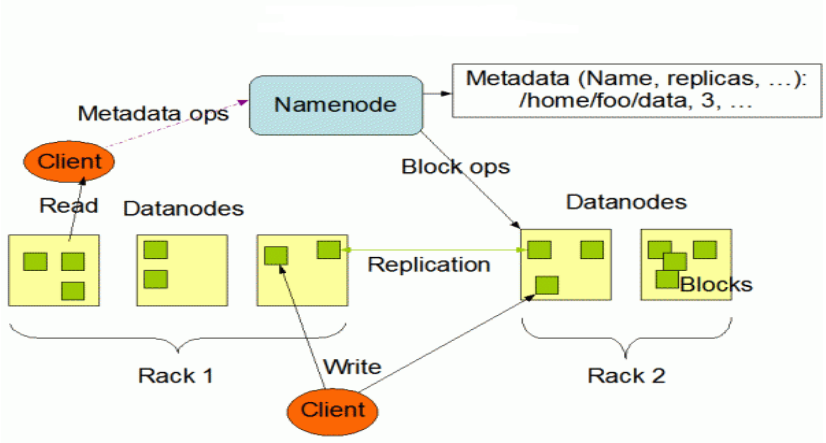

Figure 1: Hadoop Distributed File Structure Architecture

On submission of the job by the User, Hadoop initiates the Job Tracker process at Master Node. Internally, the execution undergoes three major tasks. They are Map, Shuffle Sort and Reduce as depicted in the sample.

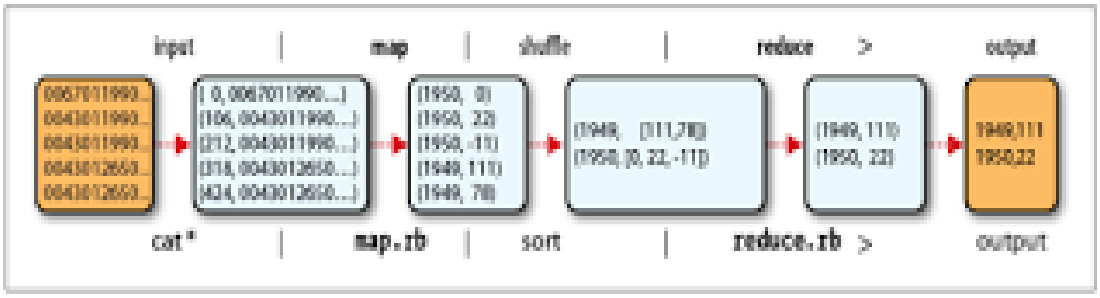

Figure 2: Map Reduce Processing Logic

Map task takes the input data to create the logical mapping. Next the mapped data is shuffled to sort it appropriately. Reduce task groups the result set based on the requirement.

5. APPROACH

Top-k query processing plays an important role in data retrieval to give an answer to a user quickly. As the size of a database is lager, the database is stored in a distributed network, and it requires the parallel processing. We address a parallel top-k query processing using Google's Map Reduce programming model.

Our experiment with the synthetic dataset is derived from the authentic historic data from NYSE [4]. So, the financial instrument-equity and its related data are meaningful.

Key advantage on leveraging Map Reduce framework is processing the large datasets in the parallel mode across the cluster of processing nodes [3]. Elastic computing is achieved using scaling out architecture.

5.1 Strategy

The strategy is to go through the list once, keep a list of the top k elements that you found so far. To do this efficiently, the system have to always know the smallest element in this top-k, so you can possibly replace it with one that is larger. The heap structure makes it easy to maintain this list without wasting any effort. It is like a lazy family member who always does the absolute minimum amount of work. It only does enough of the sort to find the smallest element, and that is why it is fast.

5.2 Parallel Algorithm

5.2.1 Top-k Query Processing

A top-k query finds the top-k answers with the highest grades on the given query. In the recent years, many assorted algorithms have been proposed to support these top k queries [9]. In this paper, we evaluate the existing algorithms for top-k queries using disk based hive and in memory Impala search methodologies.

5.2.2 Role of MapReduce Algorithm

Our approach adopts Google's MapReduce as the parallel programming framework for the top-k query processing [9]. MapReduce gives a programmer simple interface for parallel programming, that is, the programmer does not need to consider the parallel issues. Top-k query processing algorithm aims to minimize the number of probing predicates. The main contribution of this work is to design a parallel top-k query processing model using MapReduce [1]. In this process, we evaluate the algorithms through experiments and propose a new approach to improve its performance. The signatures of map and reduce [3] are as follows:

map (k1, v1) → list (k2, v2)

reduce (k2, list (v2)) → list (v3)

Based on Vector mode, map task with k1 keys and v1 values are shuffled into k2 keys and v2 values. Reduce task consolidates the intermediate result into v3 values as the result.

5.2.3 Parallel Top-k Processing

This section explains, the idea to parallelize top-k query processing using MapReduce.

We propose an approximation algorithm which estimates the number of top objects to be drawn in each node. The idea is same with the above ideal case; we want to find only necessary top objects. In order to do that, we can estimate the number of top objects from the result set. We compress a large database to a small database using wavelet and conduct data parallel query processing with the small database. From the result, we identify the ratio of top-k objects in each node and conduct the query processing with the large database and the ratio. That is, we retrieve the different number of necessary top objects in each node according to the ratio.

5.2.4 Algorithm

As the query technology Hive and Impala doesn't have the built in Top N feature, it is

essential to write a customized [9] Top N query with the following approach,

Step 1: Divide the data by the ranking key

Step 2: Sort each group by user and value

Step 3: Within each group, assign rank order to each record. This is achieved by custom rank function. The rank function keeps track of last user key and simply increments the counter. As soon as it sees a new user, it reset counter to zero. Since the data is already sorted by user and is in descending order of value, we know for sure that all records related to a single user will be sent to the same node and they will be grouped together and also sorted by value.

Step 4: Pick Top N categories. Note since our index starts with 0, we only need to categories from 0 to N-1.

5.2.5 User Defined Function in Java

As SELECT query uses rank() method, we need to write the custom rank function using Java routine as below:

public final class Rank extends UDF{ …… }

As the next step, compile Rank class to generate Rank.jar library. On starting the query tool Hive, add jar command is executed inclusive of this custom library. It will expose as User Defined Function (UDF) to write within the query statement.

6. EXECUTION

6.1 Cluster Setup

The experiments run on a 4-node cluster with 4 cores, 4GB of RAM running Ubuntu 14.04 operating systems. All algorithms were implemented using Java Compiler of version 1.7.x. In terms of Hadoop eco system, Cloudera Distributed version 5.2 has been setup. The relevant version of Hive-0.13 and Impala-2.0 are leveraged in this effort. We measure the performance in term of execution time as well as speedup and scale up between disk-based Hive and in memory based Impala.

6.2 Disk based Hive

Apache Hive is a data warehouse infrastructure built on top of Hadoop for providing data summarization, query, and analysis. HiveQL, which is an SQL-like language provided by Hive, provides a mechanism to project structure onto the data and query the data. Also this language allows traditional map/reduce programmers to plug in their custom mappers and reducers when it is inconvenient or inefficient to express

this logic in HiveQL. Its execution is depicted in the following Figure 3.

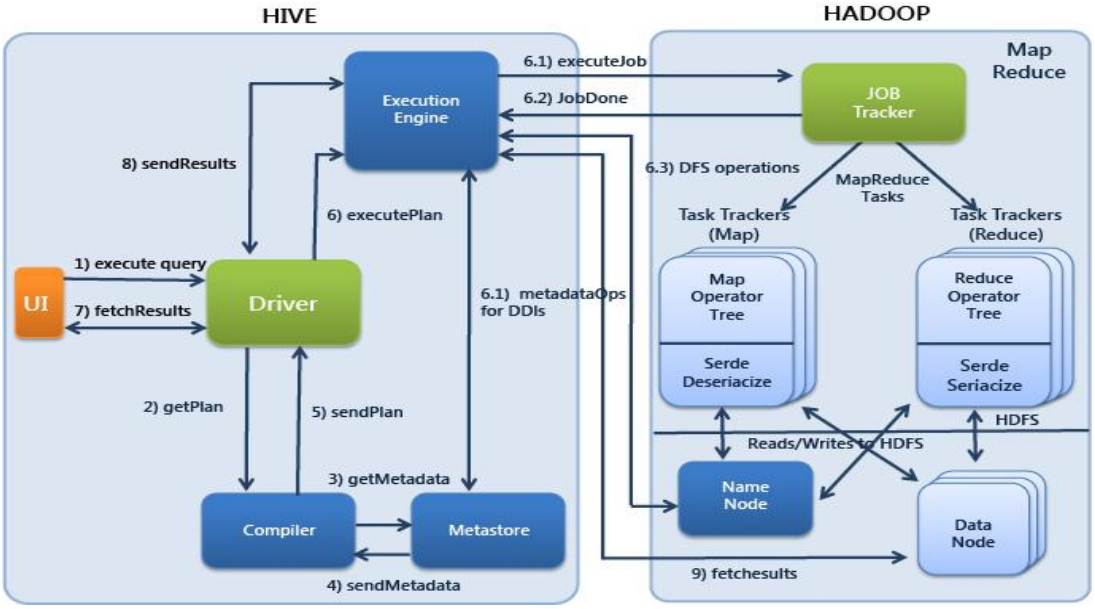

Figure 3: Hive Query Execution over Hadoop Eco System

As shown in Figure 3, the data storing in Hive databases is in fact a part of Hadoop data directory only. HDFS is a super set where all data including Hive databases is stored.

6.3 Memory based Impala

6.3.1 Motivation

Impala doesn't even use Hadoop at all. It simply has daemons running on all your nodes which cache some of the data that is in HDFS, so that these daemons can return data quickly without having to go through a whole Map/Reduce job.

The reason for this is that there is certain overhead involved in running a Map/Reduce job, so by short-circuiting Map/Reduce altogether to get some pretty big gain in runtime

6.3.2 Design

Impala is designed to scale with three key daemons.

1. Impala – collocate the data nodes
2. State Store – confirm the healthy node to accept the new work
3. Catalog – broadcast meta data changes to all Impala daemons.

It is represented in the following Figure 4.

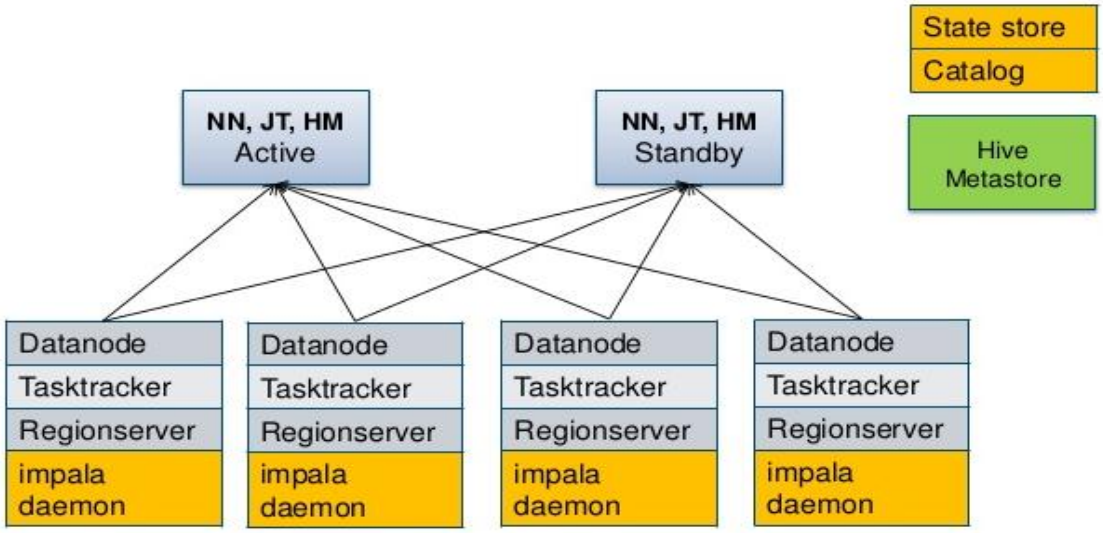

Figure 4: Key domains of Impala design

6.4 How Impala is faster

To avoid latency, Impala circumvents MapReduce to directly access the data through a specialized distributed query engine that is very similar to those found in commercial parallel RDBMS.

In the industry experiments, it is clear that Impala has outperformed Hive by '6x' to '10x' times, which is depicted in Table 2.

Table 2. Impala performance over Hive

#	Query Category	X times faster
1	Interactive	6x
2	Reports	8x
3	Deep Analytics	10x

The result is order-of-magnitude faster performance than Hive by the architecture itself.

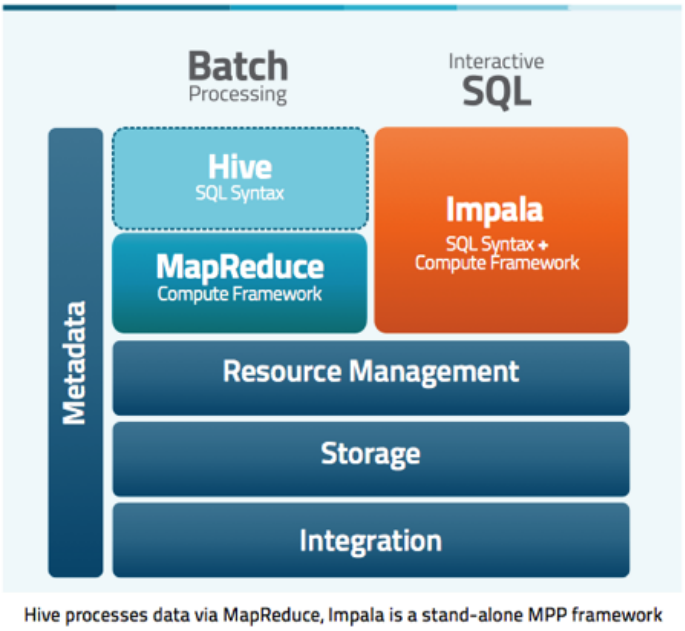

Figure 5: Impala and Hive design on common framework

That is the reason Impala is heavily used Cloudera, MapR, Amazon Web Services, Oracle, etc. The common framework for Impala and Hive is given the Figure 5.

6.5 Ranking Function

User-defined functions are written for processing column values during the Hive/Impala query.

Depending on the use case, one can write all-new functions, reuse Java UDFs that have already written for Hive, or port Hive Java UDF code to get higher-performance native Impala UDFs.

It is possible to code either scalar functions for producing results one row at a time, or more complex aggregate functions for doing analysis across. In this paper, ranking algorithms is developed not only weight based but also importance of the data set.

6.6 Sample Output

On executing Top-3 query on the particular date, the sample output is shown below:

Top	Listed-Stock	Volume	Price	Chg
1	Bank of America (BAC)	68,867,607	15.97	0.19

2 Ambev ADR (ABEV) 23,578,088 5.90 0.04

3 Whiting Petroleum (WLL) 11,441,585 7.74 0.05

7. RESULTS

This section reports the evaluation of the proposed system, top-k equity pricing query processing using MapReduce algorithm. Evaluation process involves two methodologies (1) disk based Hive (2) memory based Impala.

7.1 Performance Evaluation

We test few months of multi-dimensional datasets with varied dataset size in our experiments. Specially, we retrieve several synthetic with varying the number of data records from 10,000 to 200,000 from NYSE historical download site.

In our experiments, we evaluate the effects of the dataset size ‘n’, the result number ‘k’, and the number of machines ‘s’ using Hive and Impala. Our use case has the variation of ‘s’ from 2 to 4. ‘n’ is based on the pre-loaded historical date wise records.

7.2 Observation

As the source data file is downloaded from NYSE historical data site, it is loaded into Hadoop storage. Top-k Query is executed based on the input criteria (usually date wise) given by the end user. It is executed in two shells (1) Hive (2) Impala.

On experimenting, the results of Hive execution time are observed in the below Table 3.

Table 3. Hive Execution in sec

Node(s)	100 million	200 million	300 million
1	6.31	11.17	16.30
2	6.10	10.81	15.43
3	5.82	9.87	15.01
4	4.80	9.12	14.69

The same data set D is executed using Impala for Top-k equity pricing query Q. Impala execution results are observed in Table 4

Table 4. Impala Execution in sec

Node(s)	100 million	200 million	300 million
1	0.84	1.21	1.81
2	0.75	1.10	1.49
3	0.61	1.03	1.15
4	0.54	0.91	1.02

Key observation is the parallel execution and linearly scalable against data set volume and number of nodes.

It is completely depicted in the observed result table and graph as attached in Figure 6.

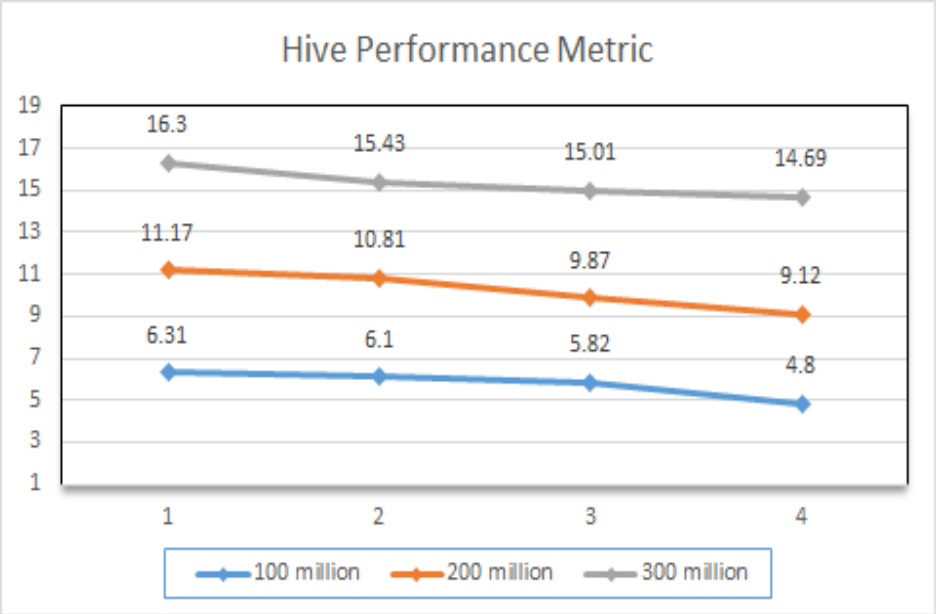

Figure 6: Performance metric using Hive

In the performance metric chart, number of Hadoop nodes (1, 2, 3, 4) is recorded in X axis, whereas the execution time is placed in Y axis. In similar way, Impala's execution time is represented in Figure 7.

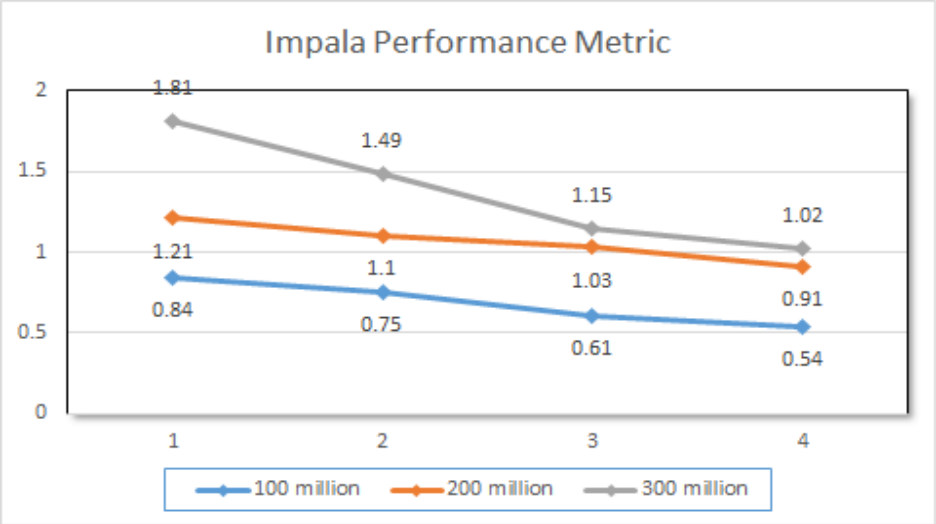

Figure 7: Performance metric using Impala

Based on the above analysis, it is inferred that disk based Hive processing is taking 8 times fasther than of memory based Impala.

A linearly scalable application can scale just by adding more machines and/or CPUs, without changing the application code. In our observation, adding nodes is linearly scalable against the performance of the execution process.

The above listed metric clearly indicates the performance trends between disk based Hive tools against in-memory based Impala of Big Data platform.

8. CONCLUSION AND FUTURE WORK

This work studies parallel top-k equity queries over large multidimensional data using disk based and memory based Hadoop framework. Key step in this proposal is to write the custom rank function for Big Data Query tools like Hive, etc.

By experiment results, this study shows the better effectiveness and scalability of in-memory based Impala than disk based Hive framework.

In future, the existing analysis pattern can be extended to the rest of commonly available financial stock exchange data.

REFERENCES

[1] Tom White, “Hadoop: The Definitive Guide”, O'Reilly Media, Inc., 3rd Edition, May 2012.

[2] Jimmy Lin and Chris Dyer, “Data-Intensive Text Processing with

MapReduce", Morgan & Claypool Publishers, 2010.

[3] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein and Khaled Elmeleegy, Russell Sears, "Paper on MapReduce Online", UC Berkeley Paper.

[4] New York Stack Exchange End of Day data downloader site: http://eoddata.com/download.aspx

[5] Dehua Chen, Changgan Shen, Jieying Feng and Jiajin Le, "An Efficient Parallel Top-k Similarity Join for Massive Multidimensional Data using Spark", International Journal of Database Theory and Application, Vol 8(3), pp 57-68, China, 2015.

[6] Thusoo A, Sarma J, Jain S, Shao N, Chakka Z, Anthony P, Liu S and Wyckoff H, "A warehousing solution over a Map-Reduce framework in VLDB", 2009.

[7] David Kotz and Nils Nieuwejaar, "Dynamic file-access characteristics of a production parallel scientific workload", The Pennsylvania State University.

[8] Ritesh Agarwal, "Extract Top N records in each group in Hadoop/Hive", wordpress blog, 18 Nov 2011.

[9] Y. Kim and K.Shim, "Parallel top-k similarity join algorithms using MapReduce", Data Engineering (ICDE), 2012 IEEE 28th International Conference on. IEEE, (2012)

[10] The Wall Street Journal on most active stocks in NYSE http://www.wsj.com/mdc/public/page/2_3021-activnyse-actives.html

.