ISSN 2348 – 9928
IJAICT Volume 6, Issue 10, October 2019 Doi:01.0401/ijaict.2016.12.01 Published on 05 (10) 2019

# QUERY OPTIMIZATION USING INDEX JOINS FOR PERFORMANCE GAIN IN HIVE

**Ms. O. Mrudula**
***Research Scholar***

**Mr. Karteek KVLN**

**Dr. A. Mary Sowjanya**
***Assistant Professor***

**Department of Computer Science & Systems Engineering,
College of Engineering (A),
Andhra University, Visakhapatnam, India**

***Abstract***— ***The Index joins are crucial for efficiency and scalability when processing the queries over big data. Hive being a batch oriented big data management engine that is well suited for data analysis application and for OLAP. For every "selective" query whose output sizes are small fraction from the contributing data, there the brute-force suffers from poor performance because of redundant disk I/O operations or lead to initiations of extra map operations. Here in this paper an attempt is made and propose index join technique to speed up the query process and integrate it in Hive by mapping our design to the conceptual optimization flow. To evaluate the performance, we create and evaluate test queries on datasets generated using TPC-H benchmark. The results indicate significant performance gain over relatively large data sets and/or high selective queries having a two-way join and a single join condition.***

***Keywords — Indexing Techniques, Map and Reduce functions, Join Operation, Hive, and Hadoop.***

## I. INTRODUCTION

The exponential growth of data being generated, manipulated, analyzed, and archived now a day's introduces new challenges and opportunities for dealing with the so called big data. Hive[1] is batch-oriented big data software, best suited for OLAP workloads and well suited for query processing and data analysis. Hive originally developed by Facebook in 2009 and now under the Apache Software Foundation, Hive is gaining popularity for its SQL like query language HiveQL and for supporting majority of the SQL operations in relational database management systems (RDBMS).

Being the expensive operation in RDBMS, join has been the focus of many query optimization techniques to improve performance of database systems. Investigating such techniques for join operations in Hive an index-based join algorithm has been developed for queries in HiveQL. When a query requires only a small subset of data selected by a predicate in the WHERE clause, the brute-force method which scans the entire tables results in poor performance for redundant disk I/Os, and irrelevant maps initiation in case the query is issued using the mapreduce[2], which is built on top of Hadoop[3] enables it to stream the data at a high bandwidth and perform massive manipulation of data.

In this work, an index-based join technique has been proposed, designed, implemented and integrated in Hive. The Hive architecture details have been extended by reverse engineering the code and mapping the design to the conceptual optimization flow.

## II. PROBLEM STATEMENT

With the advent of web 2.0, roles of the users and web applications went through a revolution. With the advent of Web 2.0, roles of the users and web applications went through revolution. The passive view-only users have become the content creators. The chance to interact over the Internet granted to users, dumped all the data from social media, blogs, videos and other web.2.0 technologies to web sites has caused increased loads to the already accumulated massive pile of data on servers.

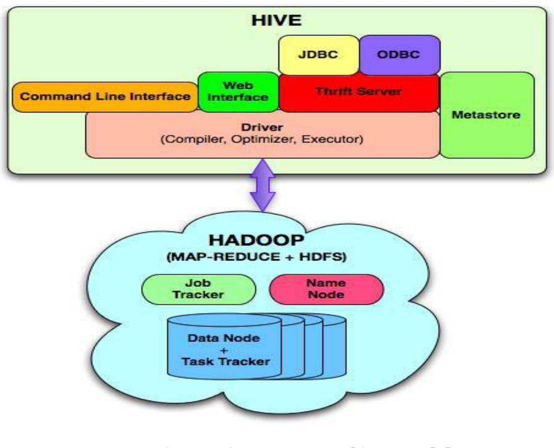

***Fig 1 : Hive System Architecture [1].***

This change demands innovative solutions to store this vast amount of data and support efficient querying over it. The raw data has to be queried to extract the worthwhile information from it.

© 2019 IJAICT **(www.ijaict.com)**

The main contribution of work is an extension of the query processing in Hive query language for performing index-based join operations, without user's interference. The proposed extension is incorporated in the Hive source code and checked for correctness of the implementation and efficiency. The results of the experiments show effectiveness of the proposed index-based join technique.

## III. HIVE ARCHITECTURE

Hive system architecture consists of several components and their interactions, and the Hadoop Map-reduce framework. The high level view of this data-warehouse architecture is depicted in Figure 1 taken from[1]. At the bottom of Figure 1, we can see the Hadoop system. At the top of Figure 1, the elevated part of Hive is placed in consort with its fundamental elements. A brief description of these elements and their roles are as follows:

- **Meta-store:** Hive system catalog contains schemas, tables, columns, and their types, tables' locations, statistics and other information essential for data management. Since meta data should be available fast, Hive uses a traditional RDBMS (e.g., Derby SQL Server, MySQL Server, etc.) to manage meta data rather than using the HDFS.

- **Driver:** The component that receives the query, after it is received by the UI from the user, and manages the lifespan of a query inside Hive. It also implements the notion of session handles and retrieves the session statistics.

- **Hive Server:** Hive server or Thrift Server allows access to Hive with a single port, that is, it allows programmatically access to Hive remotely. Therefore it provides means to integrate Hive with other applications.

- **JDBC/ODBC:** JDBC (Java Database Connection) and ODBC (Open Database Connection) which are implemented on top of Thrift sever are other access points to Hive. This Application Programming Interfaces (API) provides access to Hive from other applications. JDBC is dedicated to provide access to Java applications.

- **Command Line Interface/Hive Web Interface:** Shortly CLI and HWI, are the points to issue a query (usually by a human user) to Hive. CLI is the most popular way to use Hive that can work both interactively or with a batch of scripts. We have used CLI in our experiments.

How the components of Hive architecture interact with each other? A user submits the query via Hive CLI/Hive web Interface, JDBC/ODBC, or Thrift interface. The Driver receives the query and passes it to the compiler. Compiler does the typical parsing, type checking, semantic analysis, and pings the meta-store if needed. Finally it generates a logical query plan that is sent to the optimizer. The optimized query plan is converted to a DAG of mapreduce jobs. The executor executes these jobs in the order of dependency on Hadoop.

4. Proposed Index Joins

The existing indexes in Hive are built only over single tables. Please note that the existing index is different than "Join index", which would be an assembly of an index built over more than one table that maintains pairs of identifiers of tuples from two or more relations that match in case of a join [9][10]. This work speeds up a two-way join query expressed in HiveQL as below:

```
SELECT column_list
 FROM table1 JOIN table2
 ON (table1.col1 = table2.col1)
 WHERE ...]
 [GROUP BY ];
```

in which WHERE and GROUP BY clauses are optional. All changes are transparent to the user and the syntax of the query remains intact. For the sake of illustration we considered only two tables, but the implementation works effortlessly for multiple tables as well.

The scenario is, given two tables A and B with B having been indexed and a query to join these two tables, perform the join by scan then whole A and for each row in A probe the index on B. This is obtainable by re-writing the above query into:

```
SELECT column_list
FROM table1_index JOIN table2
ON (table1.col1 = table2.col1)
[WHERE ...]
[GROUP BY ..];
```

Our implementation uses the ideas in HIVE-1694 and manipulates the internal data structures in the query processor; however, to adjust it to process joins we added the extension presented in Fig. 3. As the first step shown in the figure, the optimizer searches for a Join Operator. If this step is omitted, the optimization is enabled for any query.

ISSN 2348 – 9928
Doi:01.0401/ijaict.2016.12.01 Published on 05 (10) 2019

The reason the Join Operator is fetched first is, depending on the different operators, different design decisions have to be made. A query containing a WHERE clause uses a distinguishably different design to benefit from the index from the one containing a GROUP BY does. Then, the optimizer examines the query for a two-way join.

Our technique can be easily extended to support multiway joins, by leaving this check out, but since we have limitations over the SELECT column list we chose to represent our work for a two-way join. In the next step we get the TableScanOperator which points to the table it should manipulate. We have to check that the table has an index and the index is valid. An index is valid if (1) it is of type compact (2) it covers all the partitions of the table. The index validity check returns true if a table is not partitioned, or if it has partitions and they are not mentioned in the WHERE clause. In case it has partitions and they are mentioned in the WHERE clause, it returns true if all the mentioned partitions are covered by the index. After this step the optimizer attempts to re-write the query. Final query looks like:

```
 SELECT column_list
 FROM index_table JOIN table2 ON
(table1.col1 = table2.col1)
[WHERE ..]
[GROUP BY ];
```

The first or the second table (whichever that has the index) is replaced by its corresponding index table. This means that table must be removed from every internal data structure in the DAG of operators and the new table must be added. Other data structures do not match with the new DAG of operators. However since there is no dependency on them, this is not of an issue when the query executes. Since the table is changed, the schema is also changed. This requires adjusting the de-serializes.

If any of the conditions is not met in the flow described in Fig. 3, the process ends in "Exit" which then implies that the execution proceeds as usual without using the index. It is important to mention that, since there is no longer any access to the base table, there is no access to all of its columns either. Instead, a subset of the attributes (the ones that are indexed) is available after the re-write. This limits the queries that can be handled to only queries referencing those specific columns. Our experiments and results are described next.

***Merits:***

1. Reduction in time complexity.
2. Load balancing
3. Indexes can be partioned depending on the size of the data.
4. Increase in speed of query lookup on certain columns of the table.
5. Indexing can be done on partioned external tables.

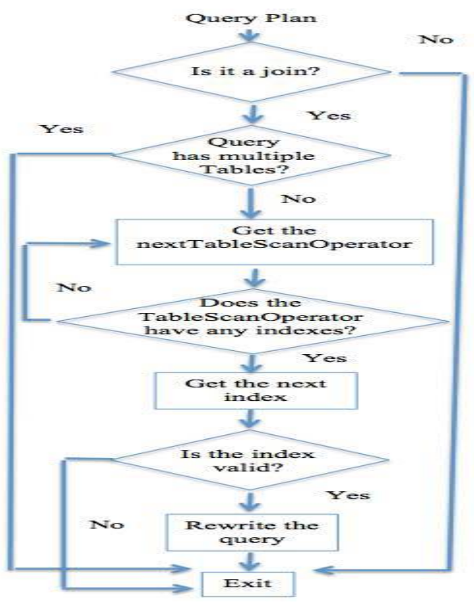

*Fig 2 : Optimization flow for index-based join*

## IV. EXPERIMENTS AND TEST ANALYSIS

### *4.1 Environment*

The test environment includes a two-node Hadoop cluster, each node having Intel Core i5-2400 3.10GHz 6MB Quad Core, 250GB SATA HDD and 8GB of RAM. Both machines were running Ubuntu v10.04 as the OS.

### *4.2 Test data*

We used the standard benchmark TPC-H version 2.14.4[8] to generate data used in our experiments. We considered only the *lineitem* and *orders* tables. We created database instances of various sizes ranging from 1 GB to 20GB for Experiments 1, and 1GB to 90GB for Experiments 2.

### *4.3 Test queries*

We perform a two-way join with optional WHERE and GROUP BY clauses. The reason for this choice is because such clauses are the children of the TableScanOperator.

Since we manipulate the TableScanOperator in our proposed solution, we have considered queries 2-4 to make sure that our approach does not affect any of the potential dependents of TableScanOperator. Here are the queries:

1. SELECT DISTINCT o.O_ORDERKEY, o.O_TOTALPRICE, o.O_ORDERDATE FROM orders o JOIN lineitem l ON o.O_ORDERKEY =l.L_ORDERKEY;
2. SELECT DISTINCT o.O_ORDERKEY, o.O_TOTALPRICE, o.O_ORDERDATE FROM orders o JOIN lineitem l

© 2019 IJAICT (www.ijaict.com)

ISSN 2348 – 9928
Doi:01.0401/ijaict.2016.12.01 Published on 05 (10) 2019

```
    ON o.O_ORDERKEY = l.L_ORDERKEY
    WHERE o.O_TOTALPRICE >15000;

 3. SELECT o.O_ORDERKEY,
    o.O_TOTALPRICE, o.O_ORDERDATE
    FROM orders o JOIN lineitem l
    ON o.O_ORDERKEY = l.L_ORDERKEY
    GROUP BY o.O_ORDERKEY,
    o.O_TOTALPRICE, o.O_ORDERDATE;


 4. SELECT o.O_ORDERKEY,
    o.O_TOTALPRICE, o.O_ORDERDATE
    FROM orders o JOIN
    lineitem l ON o.O_ORDERKEY =
    l.L_ORDERKEY WHERE
    o.O_TOTALPRICE > 15000 GROUP BY
    o.O_ORDERKEY, o.O_TOTALPRICE,
    o.O_ORDERDATE;
```

### *4.4 . Run-time parameters*

The parameter mapred.map.tasks controls the number of map tasks and mapred.reduce.tasks holds the number of reduce tasks. In our experiments, these parameters were set to 20 and 4, respectively.

### *4.5. Evaluation metrics*

In all of our experiments, we measure performance using the query response time in seconds(s). In Experiments 2, we measure performance by also considering query selectivity since it becomes important in the presence of indexes.

### *4.6. Experiments 1*

Experiments 1 includes execution of the 4 query types, each one is executed 5 times, on a multi-node and a singlenode Hadoop cluster using 5 different dataset sizes 1GB, 5GB, 10GB, 15GB, 20GB with *lineitem* holding almost 5/6 of the total data and number of tuples ranging from about $7\times10^6$ to $150\times10^6$. Figures 3 to 6 depict the average response time for each data size.

In the multi-node setup, moving from 1GB of data to 20GB, in all steps our index-based approach outperforms the existing one. The larger the data are, the bigger the gap between the index-less and index-based approaches becomes. Our index method is almost two times faster than the index-less approach in all graphs.
In the single-node setup, we see the same behavior; for each data size, our proposed method outperforms the normal one and the larger the data are, the bigger the gap between the index-less and index approaches becomes. The index method is almost about two times faster than the index-less approach.

Comparing the results from both setups, we note that the single-node setup works faster than the multi-node setup for the data size 1GB in both approaches. For the data size of 5GB, the multi-node setup is slightly faster than the single node case. Afterwards, multi-node is almost two times faster than the single-node. The performance difference between the two setups indicates the networking overhead only pays off when the data size is relatively big. In our experiments, the data size over 5GB is suitable for the multi-node setup. We say 'relatively' because this measure depends on the hardware configuration of the computers as well as the networking equipment.

Experiments showed that repeating the same query over the same dataset does not lead to significantly different response times. The reason is, Hive does not cache the query plan and starts from scratch for each query. This causes the first response time not to be always the longest one. With the growth of data size, the deviation from the average response time in each step grows.

To better study the performance of our technique, in the rest of Experiments 1, we conduct the same test with different queries, which are extensions of query1.

Looking at Figures 3 to 6, the graphs show similar curves, using which we concluded that the 4 types of queries have almost the same behavior and they did not lead to significantly different response times in neither approach.

The most expensive operator in all the queries is the JOIN. Neither WHERE nor GROUP BY, which where extra clauses added to queries 2-4, initiates a new mapreduce job. The number of mapreduce jobs in all the queries is equal to 1. As a result, in the rest of the experiments we only use Query 1.

We also studied the cost of index creation in terms of time and space to decide whether or not to use index. Figures 7 and 8 compare the size of the index with the size of the data and the time taken for creating the index with the average time taken for an index-less Query1 execution on multimode setup respectively.

As shown in Fig. 7, the size of the index is less than 15% of the input dataset size, which is relatively small. This is due to the simple tiny structure of indexes in Hive which only stores pairs of values and their relative locations from the beginning of the index file.

© 2019 IJAICT **(www.ijaict.com)**

However, the index size can vary based upon the number of columns on which the index is created. In all our tests, the index had been built over the join attribute, L_ORDERKEY.

Depending on the dataset size, the index creation time increases as the data size grows. As shown in Fig. 8, the time grows from 60% to 75% of the time taken for executing the query itself. This is because processing the query and creating the index scan the entire dataset for both which takes the major part of the process. This scan operation is considerably reduced for the queries when base table is replaced by the index table. Recall that indexes are built only once, and its cost is amortized over many executions of queries using the index.

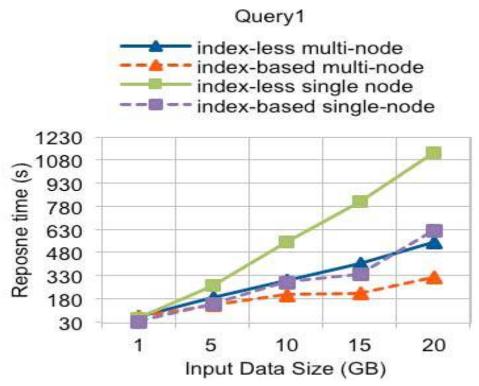

*Fig 3 : Query 1 response time with/out index on multi-node and single-node setups*

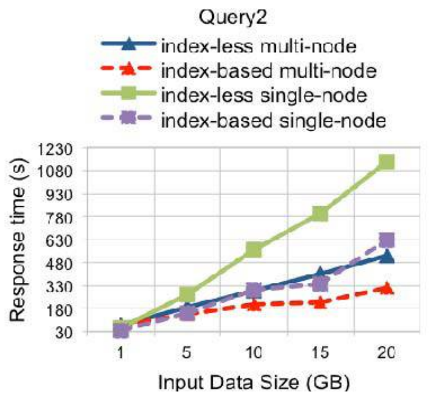

*Fig 4 : Query 2 response time with/out index on multi-node and single-node setup*

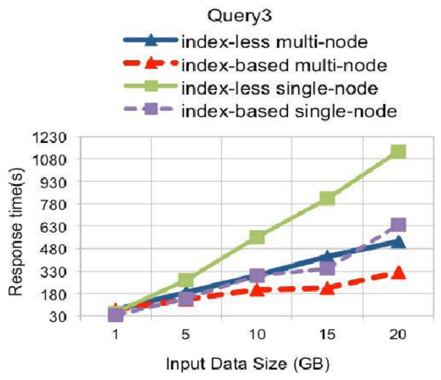

**Fig 5 : Query 3 response time with/out index on multi-node and single-node setups**

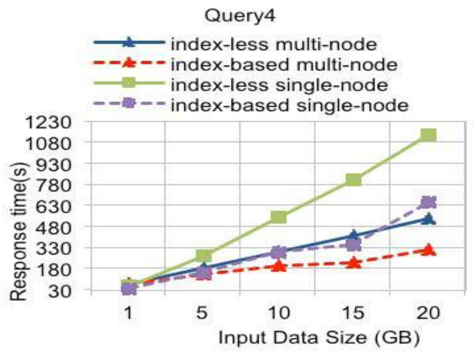

*Fig 6 : Query 4 response time with/out index on multi-node and single-node setups*

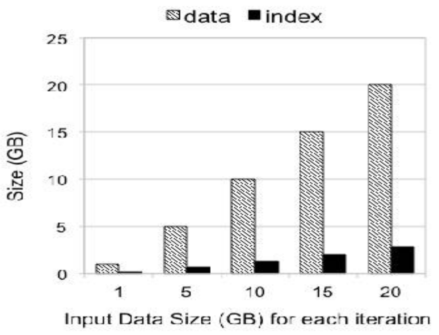

*Fig 7 : Index size vs. data size*

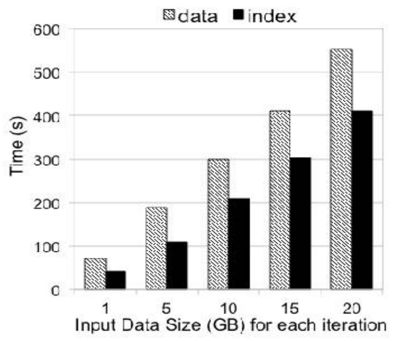

*Fig 8 : Index creation time vs. query response time*

© 2019 IJAICT **(www.ijaict.com)**

ISSN 2348 – 9928
Doi:01.0401/ijaict.2016.12.01 Published on 05 (10) 2019

### *4.7. Experiments 2*

The second set of experiments we conducted for performance measurement considered different value for the query selectivity ratios. For this purpose, we used Query1 over the tables *orders* having a fixed size of 164 MB with 15 $\times 10^5$ tuples and also table *lineitem* of size ranging from 0.71 GB to 90.6 GB and with the number of tuples ranging from $6\times10^6$ to $7\times10^8$. In order to increase the selectivity, the *lineitem* distinct join key or the output size of the query was kept at 1,500,000 while the data was doubled each time. In this experiment, we were interested to find the point at which our index-based approach works noticeably better than the index-less approach on our current multi-node setup.

Fig. 9 shows the graphs for average response times measured. As we move from case 1 to 8 in this figure, the index-less approach grows non-linearly, while the indexbased approach remains more or less constant at an average of about 87 seconds. In case 6, with 45GB of data and 0.3% as query selectivity, the index-based approach is an order of magnitude faster than the index-less approach. The next iteration, case 8, with double query selectivity (0.1%) and double data size (90GB), our approach is 20 times faster than the index-less method. The exponential behavior of the index-less graph in Fig. 9, started at iteration 6 with 0.7% as the query selectivity. If the curve keeps the same trend, our index-based approach can possibly be 2 orders of magnitude faster than the index-less approach at 45TB of data with very selective (0.0007%) queries.

As indicated in Fig. 9, the index size gradually drops from 18% of the data size to 9% over the 8 iterations. The Hive index size grows or shrinks proportional to the data size or distribution. In Experiments 2, the index decreasing rate is due to the data distribution, as at each iteration, the number of distinct values of all attributes, was kept the same while the volume of data was doubled.

In regard to index construction time, in Fig. 9, we can see that, up to iteration 5, index creation time is slightly less than the execution of Query 1 without index, and exceeds the query run-time afterwards.

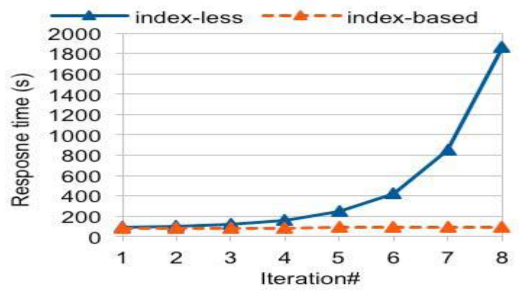

***Fig 9. Query 1 response time with/out index on multi-node and single-node setup (Experiments 2).***

## V. CONCLUSION

Indexes have been around for long time and the benefit of using them is well known. However, deciding when to use indexes in a situation requires extensive evaluation and trade-off between its cost and performance. In this research, we used the current Hive indexing structure to speed up join queries. From Experiments 1, we observed, in general, larger the data are, larger the performance gain becomes. Our approach grew linearly in all cases shown in Figures 3 to 6. In Experiments 2, we increased the sizes of the datasets with growing selectivity ratios. The results of these experiments indicated that our approach is exponentially faster than the current Hive approach.

We saw in Fig. 7, that the index size was almost fixed at only 15% of the data size in Experiments 1; and in Fig. 9, it took an average of 12% of the data in Experiment 2.Though index size depends on the data distribution and the number of attributes for indexing, our experiments showed the Hive index space utilization is reasonable. Index creation time graphs depicted in Figures 7 and 9 showed the time required on building an index depended on the data distribution, the more duplicated tuples resulted in a slower index creation process became. In Fig. 9, the worst case (iteration 8) index creation took almost twice the query execution time.

Index construction comprises of reading the whole data, sorting it, and eliminating the duplicates, which is a quite lengthy process. Until the data in the base table is untouched, any types of queries that have the privilege to utilize the index can use the index, nevertheless the index creation cost is only incurred once.

With respect to accessing the index, current Hive indexes do not provide an instant access to values, which undoubtedly comes with heavy space overhead. What they offer instead is, scanning a huge amount of data is replaced with scanning a drastically small set of it that holds the desired values. The cost of finding a value in the current index Hive is $O(n)$, where n is the number of tuples. Assuming a Hive table of n tuples and its index with mentries, accessing a specific value in the index is reduced from $O(n)$ to $O(m)$ with $m$ much smaller than $n$.

The indexing technique in Hive is rather new and the progress has been limited to current index structure and also the query life cycle. There are a number of optimization ideas to further improve Hive index-based joins, including: designing a cost-based optimizer, which can evaluate a query plan to help decide to use indexes or not, probably by using column level statistics and auto-indexing or the ability for the compiler

© 2019 IJAICT **(www.ijaict.com)**

ISSN 2348 – 9928
IJAICT Volume 6, Issue 10, October 2019 Doi:01.0401/ijaict.2016.12.01 Published on 05 (10) 2019

## References

[1] Antony, S., Chakka, P., Jain, N., J., Liu, Murthy, R.Sarma, J. S., Thusoo, A., Zhang, N "Hive – A Petabyte Scale Data Warehouse Using Hadoop," IEEE 26th Intl.Conf. Data Engineering (ICDE), Long Beach, CA, 2010, pp. 996 – 1005

[2] Dean, J., Ghemawat, S. "MapReduce: Simplified Data Processing on Large Clusters," Mag. Commun . ACM 50th anniversary, vol. 51, issue 1, 2008, pp.107-113

[3] ApacheHadoop[Online].Available: http://hadoop.apache.org/

[4] HIVE-1644[Online].Available: https://issues.apache.org/jira/browse HIVE- 1644

[5] HIVE-1694[Online]. Available: https://issues.apache. org/jira/ browse / HIVE- 1694

[6] Y. Jia and Z. Shao. A Benchmark for Hive, PIG and Hadoop, 2009. https://issues.apache.org/jira/browse/HIVE-396

[7] The Apache Software Foundation. Hadoop MapReduce. http://hadoop. apache. org/.

[8] The Apache Software Foundation Hive. http://hive. apache.org

[9] Hive Wikipedia. http://wiki.apache.org/hadoop/Hive/.

[10] An, M., Wang, W., Wang, Y., "Using Index in the MapReduce Framework,", 12th Intl. Asia Pacific Web Conf. (APWEB), Beijing, China, 2010, pp. 52-58

[11] K. Mythili and H. Anandakumar, "Trust management approach for secure and privacy data access in cloud computing," Green Computing, Communication and Conservation of Energy (ICGCE), 2013 International Conference on, Chennai, 2013, pp. 923-927.doi: 10.1109/ICGCE.2013.6823567

[12] Chansler, R., Kuang, H., Radia, S., Shvachko, K."The Hadoop Distributed File System," in Proc. IEEE Conf. Mass Storage Systems and Technologies (MSST), Incline Village, NV,2010, pp. 1 – 10.

[13] TPC-H[Online]. http://www.tpc.org/tpch/

[14] Li, Z., Ross, K. A. "Fast joins using join indices," in The International Journal on Very Large Data Bases, vol. 8, issue 1, 1999, pp.24

[15] Gruenheid,A. Mark, L.Omnecinski,E."Query Optimization using column statistics in Hive," in Proc.15th Symp. Intl. Database Engineering & Applications (IDEAS), Lisbon, Portugal, 2011, pp. 97-105, 2011

## *Authors*

O.Mrudula has done her B.Tech in Information Technology from M.V.G.R College of Engineering and M.Tech in Computer Science from Andhra University.

Karteek KVLN has done his B.Tech in Computer Science & Engineering and M.Tech in Computer Science from Andhra University.

Dr.A.M.Sowjanya has done her B.Tech and M.Tech in Computer Science. Her Ph.D is in Incremental clustering. She is at present working as an Assistant Professor in College of Engineering (A), Andhra University. Her research interests include Data mining and Big Data.

© 2019 IJAICT **(www.ijaict.com)**