IJCRT.ORG ISSN : 2320-2882

INTERNATIONAL JOURNAL OF CREATIVE RESEARCH THOUGHTS (IJCRT)

An International Open Access, Peer-reviewed, Refereed Journal

Query Optimization For Performance Gain With Index Joins In Hive

Sowmya Chunduru, Dr.M.Akkalakshmi

[#]*Authors designation*

Department of Computer Science & Systems Engineering

GITAM University

Abstract --Index joins is crucial for efficiency and adaptability when processing queries in big data. HIVE is a batch oriented big data management engine which is well accepted for data analysis applications and for an OLAP. For very "selective" queries whose outputs are tiny fraction from the data contribution, there the brute- force suffers with poor performance because of dispensable disk I/O operations or lead to initiations of added map operations. Here, An attempt is made and propose to speed up the query process and integrate it with index joins technique in Hive by mapping our design to the ideational optimization flow. To evaluate the performance, we generate and estimate test queries on datasets generated using TPC-H benchmark. And the results indicate notable performance gain over proportional large data sets and /or highly select queries having a two-way join and a single join condition.

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

I. INTRODUCTION

With the origin of web 2.0, roles of the users and web applications went through a uprise. The unassertive view- is that the users have become content creators. The chance to interconnect over the internet granted to users, get shut of all the data from social media, social sites, videos and other web.2.0 technologies to web sites has caused in increase of loads to the already assembled large amount of data on servers. This change demands in the innovation of solutions to store the huge amount of data and support coherent querying over it. The raw data to extract the worthwhile information need to be queried from it. This gives new horizons for the development of novel algorithms, tools, and services to process queries over this vast amount of data in a equitable time frame.

Hive is a data warehouse software suites for OLAP caseload to handle and the query over huge volume of data to be conferred in a distributed storage. The HDFS is the ecosystem in which Hive sustains the data dependability and get through from hardware failures. The only SQL-like relational big data warehousing process developed on top of Hadoop to the best of our knowledge is Hive. A high-level programming model, called *mapreduce* [3], erect on top of *Hadoop* [1] empower it to stream the data at a high bandwidth and perform massive manipulation of data.

As joins are salient operations in databases, which depend on the predicate, data, etc., granting information "combined" from different relations. It also gives us more data analysis and mining tasks which are important in the factor of business intelligence for finding gripping and useful patterns in large amount of data. Therefore, upgrading various join operations can result in consequential performance improvement. In relational databases, the join operations are through indexing or external sort techniques, without which the brute-force scan of the entire table is incompetent for large data. This is more crucial in particular when a small fraction of the tuples participate in the join operation.

The major aspects impact the performance gain in Hive with index joins contains very large data volume and low index maintenance cost.

Though Hive is anticipated to work well with huge amount of data, indexing can further upgrade the performance by lessen the amount of data retrieves from the contributing tables. Having few updates, as a characteristic of big data, makes the cost of index maintenance of less importance or affordable. Additionally, the index types proposed and developed in Hive take up a pretty small space.

The output of this paper gives a procedure to perform join with MapReduce operations, over huge

sets of data stored in a Hadoop-based cloud. Estimating analytically the performance of the suggested approach, gives a recent indexing feature in Hive to improve staging over non-indexed queries. The go on paper is sorted as follows. Section II details Hive Architecture and Section III evaluates related work. Query Optimization using Index joins is given in Section IV, and its experimental evaluation and outputs are presented in Section V. Ending remarks and subsequent work are discussed in Section VI.

II. HIVE ARCHITECTURE

Hive system architecture contains of several parts and their relations, and the Hadoop Map-reduce framework. The high effective view of the data-warehouse architecture is shown in Figure 1.

At the end of Figure 1, we can see the Hadoop system. At the start of Figure 1, the highlighted part of Hive is placed in associate with its fundamental elements. A brief of these elements and their usage are as follows:

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

For example, in index joins, we need to know if a table given in the query is indexed or not. Or we have to know if the index rounds all the parts of a table. Such details are stored in the meta-store and is pleaded by the query compiler only once, but the vital part of this data is sent to several trackers.

- **Driver:** The component that obtains the query, once it is received by the UI from the user, and directs the existance of a query inside Hive. It also executes the conviction of session controls and reclaim the session statistics.

In Figure 1, the Driver comprises of three main components, Compiler, Optimizer, and Executor. The compiler interprets HiveQL into a DAG (Directed Acyclic Graph) of mapreduce tasks that are implemented by the executor or effect engine in the order of their addictions. The optimizer occupies at some point in between the compiler and executor to enhance the performance. The index joins the algorithm with the compiler and optimizer modules, which will be described in more detail.

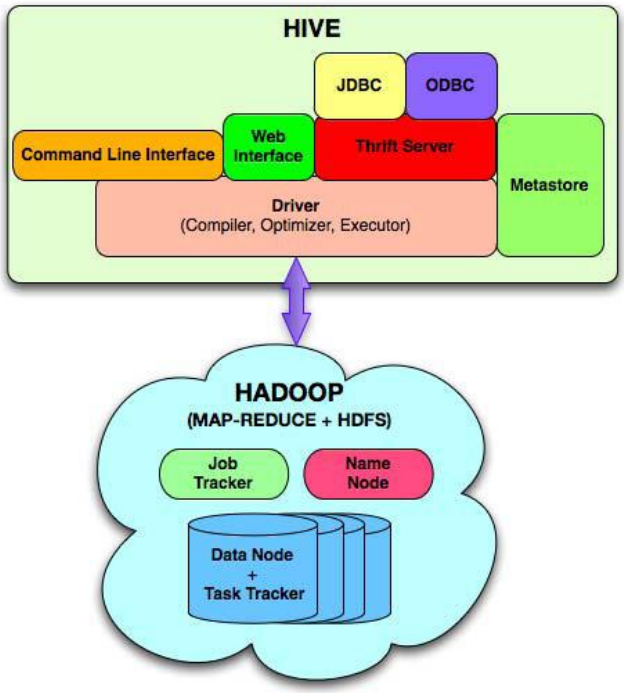

Figure 1 Hive System Architecture [7].

- **Hive Server:** Hive server or Thrift Server permits approach to Hive with a single port, like, it allows access to Hive remotely. And it provides means to joins Hive with other applications.

Thrift is a efficient cross-language service development framework; or clearly, a binary communication protocol. Clients in various programming languages can communicate logically with Hive using the "thrift interface".

- **JDBC/ODBC:** JDBC (Java Database Connection) and ODBC (Open Database Connection) which are applied on top of Thrift sever are some more access points to Hive. These Application Programming Interfaces (API) gives access to Hive from other applications. JDBC is committed to provide access to Java applications.

- **Command Line Interface/Hive Web Interface:** CLI and HWI, are the Used to issue a query (usually by a human user) to Hive. CLI is the most significant way to use Hive which can work both collectively and with a batch of scripts.We used CLI in our executions.

How the components of Hive architecture interacts?

A user presents the query via Hive CLI/Hive web Interface, JDBC/ODBC, or Thrift interface. The Driver gets the query and process it to the compiler. Compiler do the typical parsing, type checking, semantic analysis, and signals the meta-store if needed. At last it generates a query plan that is sent to the optimizer. The optimized query plan is transformed to a DAG of

mapreduce jobs. The user runs these jobs in the order of priority on Hadoop.

III. Related Work

Before going to the related work, it is needed to point the generic optimization flow in Hive, as many optimizations abides to it.

The DAG of operator goes to the optimizer to opt for the best feasible order of operations on the original data in the query plan. Many RDBMSs today profits from a cost-based query optimizer. Hive offers a clear yet rule-based optimizer where the operator tree is recursively traversed and further divided into series of mapreduce sterilisable tasks, each summarizing a part of the query plan, acceptable to be executed on HDFS. The plan even has the essential samples/partitions such are by the query itself. Hive optimization contains a chain of variations in which the DAG results one of the transformation step is fed as an input to the next. Start to change the optimizer or adding new optimization algorithm is the Transform interface. For this, one should applies the Transform interface using their logic for the chain of optimizations in Hive Optimizer. Hive optimizer invoking all the transformations, one after other, to modify the query plan. Start to change the optimizer or adding new optimization algorithm is the Transform interface. For this, one should applies the Transform interface using their logic for the chain of optimizations in Hive Optimizer. Hive optimizer invoking all the transformations, one after other, to modify the query plan.

Below is the description of modules and their roles[7].

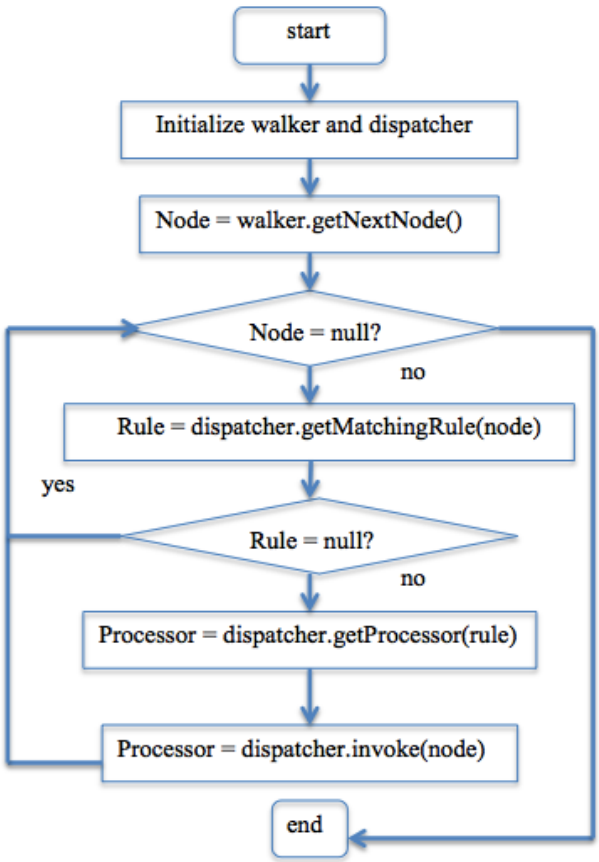

Figure 2 Hive optimization flow [2]

- **Node:** As the input/output of the optimizer is in a tree form, This is representing the elements of a tree called *nodes*.

- **GraphWalker:** GraphWalker can be said as mechanism to traverse the tree fed to the optimizer. This is GraphWalker and it picks up the nodes for visiting and maintains track of the already visited ones.

- **Rule:** Rule is a pattern in the query which is Used for regular expressions notation. As the elements in the DAG are operators, the very basic tokens used here in regular expressions are also of the same type.

- **Dispatcher:** Dispatcher is generally the rule matching and, some cases a certain rule is equal with a Node, it calls for the corresponding processor.

- **Processor:** The processor simply, defines and includes the optimization logic.

Figure 2 The optimization flow in Hive is as below: The optimizer module gets the query plan in the form of a DAG of operators. The GraphWalker brings a node and the dispatcher checks for any rule is similar with a node, If it is so, it calls for the appropriate processor and gets the next node.

Any optimization in Hive observes to the above flow.Below, we will discuss a few optimization techniques associates to indexes in Hive.HIVE-1644 [4] is the application of processing the `WHERE` clause with the index.

```
SELECT column_list
FROM table_name
WHERE predicate;
```

and re-writes it into:

```
INSERT INTO intermediate_file_name
SELECT _BUCKETNAME, _OFFSETS
FROM table_index
WHERE
relevent_part_of_the_predicate;
```

The new query returns the table with the index table and looks for the address of the values. The appropriate part of the predicate is the part which can be processed by the indexes, is a conjunction of the binary expressions .The revised query is compiled and the processed root tasks are added to the raw query root tasks. Then the initial query runs over the transitional, where results are produced from the revised query. All column references in HIVE-1644 should refer to the same table (no joins or sub-queries). Another optimization in HIVE-1694 [5] speed up the queries with GROUP BY clauses. It re-writes the below query:

```
SELECT COUNT (key)
FROM TABLE
GROUP BY k
WHERE predicate;
```

into the following query:

```
SELECT SUM (_COUNT_OF_key)
FROM index_table
GROUP BY key
WHERE predicate;
```

1.Sppeding up a query with a Where clause with index

HIVE-1644 [4] is the execution of a query consists a `WHERE` clause that holds the index to get the tuples. The important questions are: when/where the optimization is applied, how it is applied, what the constraints are, and how it can be triggered? HIVE-1644 is a physical query optimization. As mentioned in the Hive architecture, the optimizer receives an operator DAG and performs the enabled or possible optimizations. This means optimization is applied at the end of or during the logical plan generation stage. The case for HIVE-1644 is slightly different. As a physical optimization it happens more precisely after the logical plan operation when the complete operator tree is being transferred to the tree of tasks, but Hive optimizer and physical optimizer have the same components we already discussed and consequently the tree goes through similar steps.
The physical plan optimizer invokes all the physical optimizations in turn.

2. Acclerating a query with a GROUP BY clause using index

The goal of HIVE-1694 is to accelerate queries containing `GROUP BY` clauses.As in HIVE-1644, it uses query re-writing technique,but its core design is not limited to re-writing only.Though this optimizatio seem intuitively as physical, in the code it is not organized in the physical optimization package, and as a result its optimizer implements the Transform interface.

In its optimizer called RewriteGBUsingIndex.java, it first checks if the query meets all the constraints such as:

- The presence of the index over the join key
- Validation of the index
- Coverage of the index over partitions (if any)
- Having only one table (no joins) in the query
- Having a single `COUNT (index_key)` function in the query
- Addressing barely the columns that are in the index key

3. Using indexing over mapreduce

Hive merges all required facilities need to perform queries over mapreduce. This means one can issue a query without Hive by writing their own map and reduce methods and managing the query lifecycle themselves.

A recent work integrated the index into mapreduce framework, which tries to reduce the number of maps generated to access the initial data using an index with random access. The index structure is a B+-tree, which is not built using a conventional create-by-insert in a top-down fashion. Instead, since the data and accordingly the index is not supposed to be updated, the data is read in batch-mode using the mapreduce framework itself; afterwards it is sorted on the (index_key,offset) pairs and written sequentially to a file. These sets form the leaf nodes for the index tree. In the next step, all the leaf nodes are scrutinized and the half way index nodes are created in a bottom-up manner. In a conventional B+-tree, pointers connect the leaves while this method keeps all the leaves in a consecutive space.

4. Query optimization using statistics

Statistics play a vital role in the factor of query optimization. Statistics either help the optimizer to select the more economical plan such as join reordering or work for the query output like the `COUNT(*)` clause in a query. Hive provides table and partition level statistics as well as column-level statistics.

A current work proposed storing column-level meta data in Hive tables to get benefit during query execution.Column-level statistics or more distinctively, histograms that indicate value distribution within a table gives more exact information than just the table size to estimate the output size. A new table is added to Hive meta-store that carries the number of distinct values, number of null values, min and max values and most periodic values as its fields.

There are few defects about their work as follows:

- Collection of meta data at any levels like partition, table or column put out extra above in terms of time and space for the database management system though it is not often updated.
- The executed elements are rather a different element than a detailed embedded constituent in Hive. Other optimization techniques need more advanced implementation that requires additional comprehensive knowledge over the architecture and dependencies.
- The time taken to extract the statistics, done by supplies direct queries to Hive, is totally abandoned.

5. Hive joins

Hive has distinct join operation implementations: Common Join, Map Join, Bucket Map Join, Sort Merge Bucket Map Join and Skew Join. Common Join is the basic implementation of the join which reads the entire tables and has the greatest number of comparisons. Based on the data distribution, tables' sizes, and being sorted, one of the another implementations enhances the best choice to manage the join. We decided to execute our tests and differentiations using the Common Join because (1) using the index reduces the number of comparisons and Common Join has the greatest number of it. (2) Other implementations are necessarily built and used for data with specific features.

IV. PROPOSED INDEX JOINS

The already used indexes in Hive are built only for single tables. Please note that the already used index is different than"Join index", which would be an erection of an index built over more than one table that continues pairs of identifiers of tuples from two or more relations that equals in case of a join [9][10].

This work speeds up a two-way join query indicates 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 discretionary. All our changes are clear to the user and the syntax of the query remains flawless. For the sake of instance we review only two tables, but our implementation works easily for multiple tables as well.

The synopsis is, given two tables A and B with B having been indexed and a query to join these two tables, execute the join by search then whole A and for each row in A analysis the index on B. This is available by revise the above query into:

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

This optimization flow observes to the regular optimization flow reported in Section II. Our implementation utilizes the ideas in HIVE-1694 and handles the internal data structures in the query processor; even so, to adjust it to process joins we included the extension presented in Fig. 3. As the first step shown in the figure, the optimizer explores for a JoinOperator. If this step is excluded, the optimization is allowed for any query.

The reason the JoinOperator gets first is, based on the different operators, various design decisions have to be made. A query contains a WHERE clause uses a separable different design to benefit from the index from the one contains a GROUP BY does. Then, the optimizer searches the query for a two-way join.

Our technique can be simply extended to support multiway joins, by exit this check out, but since we have controls over the SELECT column list we get to represent our work for a two-way join. In the other step we obtains the TableScanOperator which points to the table it should handle. We have to see that the table has an index and the index is correct. An index is logical if (1) it is of type compact (2) it encloses all the partitions of the table. The index sustainability check returns true if a table is not partitioned, or if it has partitions and they are not refered in the WHERE clause. If and only if it has partitions and they are mentioned in the WHERE clause, it restores true if all the mentioned partitions are enclosed by the index. After this step the optimizer seeks to revise 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 restored by its corresponding index table. This means that table must be deleted from every internal data structure in the DAG of operators and the new table has to be added. Other data structures does not equals with the new DAG of operators. Even so, since there is no dependency on them, this is not an issue when the query performs. Since the table is revised, the schema is also revised. This requires the modification.

If any of these context does not met the flow defined in Fig. 3, the process ends in "Exit" which then leads that the performance proceeds as usual without using the index. It is necessary to mention that, since there is no longer any permit to the base table, there is no permit to all of its columns either. Rather, a subset of the attributes (the ones that are indexed) is accessable after the revise. This controls the queries that can be maintained to only queries refers those specific columns. Our executions and outputs are described next.

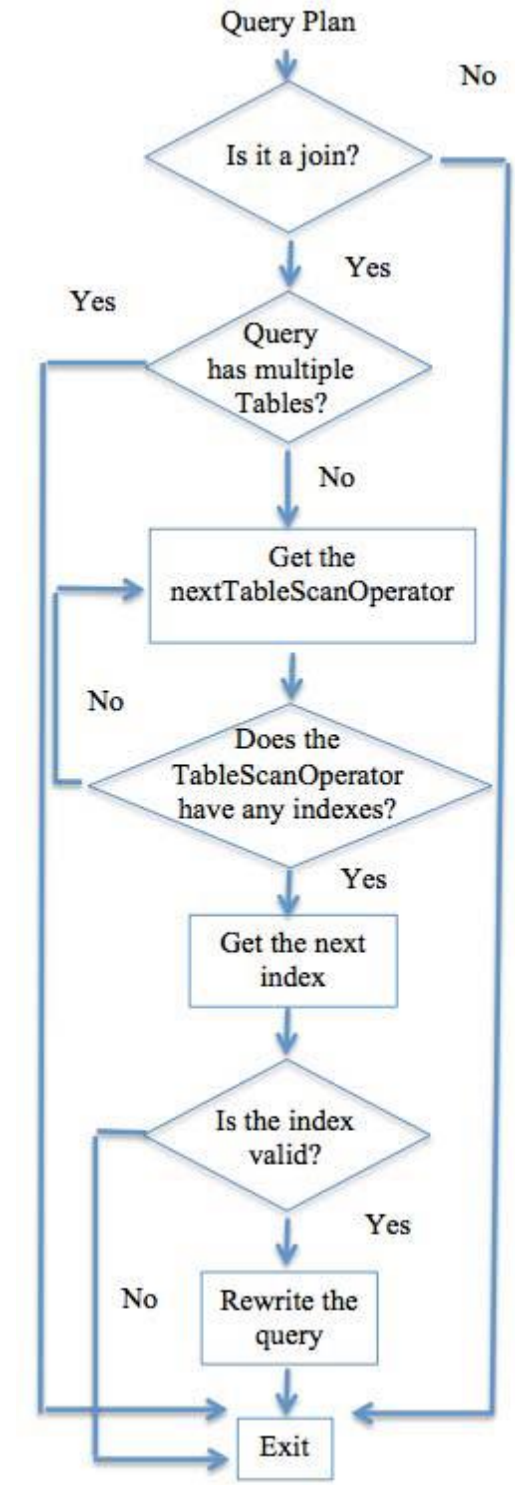

Figure 3. Optimization flow for index-based join

V. EXPERIMENTS AND TEST RESULTS

A. Environment

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

B. Test data

We pre owned the standard benchmark TPC-H version 2.14.4[8] to execute data used in our experiments. We have taken only the *lineitem* and *orders* tables. We generated database smaples of various sizes reaching from 1GB to 20GB for Experiments 1, and 1GB to 90GB for Experiments 2.

C. Test queries

We execute a two-way join with optional WHERE and GROUP BY clauses. The cause for this is,such clauses are the nodes of the TableScanOperator.

Since we handle the TableScanOperator in our present solution, we have taken queries 2-4 to make sure that our process does not affect any of the likely relates 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 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;
```

D. Run-time parameters

The parameter mapred.map.tasks runs the number of map tasks and mapred.reduce.tasks handles the number of reduce tasks. In our executions, these parameters were set to 20 and 4, respectively.

E. 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.

F. 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 7×10^6 to 150×10^6. Figures 4 to 7 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 singlenode 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 4 to 7, 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 approaches.

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 8 and 9 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. 8, 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.

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. 9, 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.

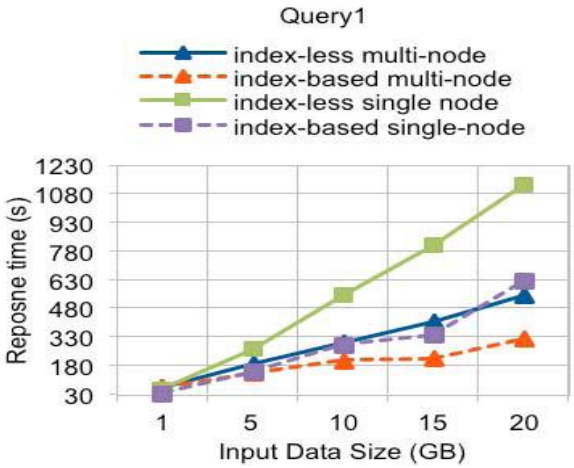

Figure 4. Query 1 response time with/out index on multi-node and single-node setups

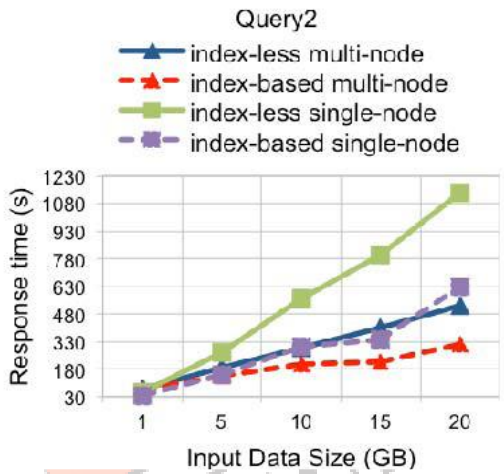

Figure 5. Query 2 response time with/out index on multi-node and single-node setup

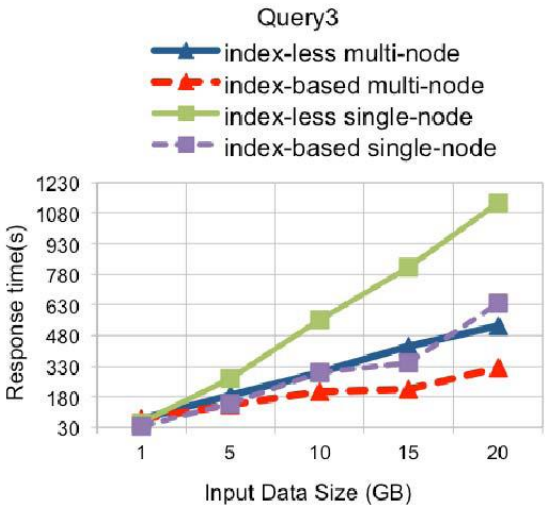

Figure 5. Query 3 response time with/out index on multi-node and single-node setups

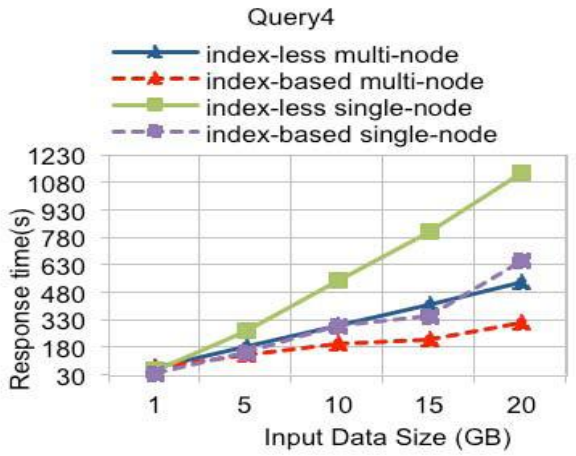

Figure 7. Query 4 response time with/out index on multi-node and single-node setups

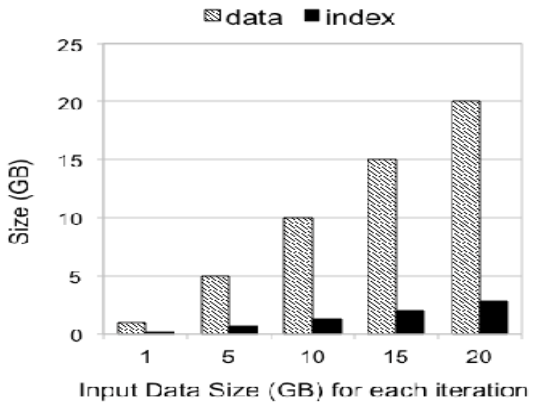

Figure 8. Index size vs. data size

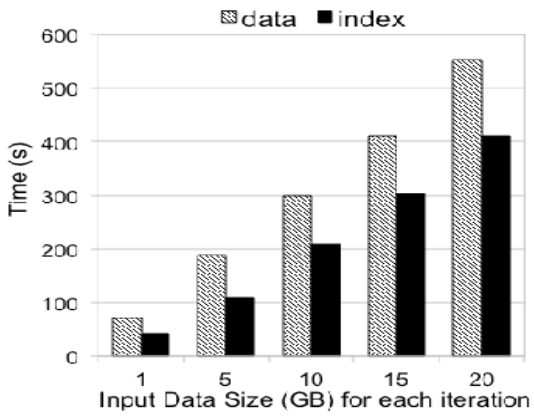

Figure 9. Index creation time vs. query response time

G. Experiments 2

The second set of executions we manages for performance measurement gives different value for the query discrimination ratios. For this, we used Query1 over the tables *orders* having a specific size of 164 MB with 15 $\times 10^5$ tuples and also table *lineitem* of size reaching from 0.71 GB to 90.6 GB and with the number of tuples reaching from 6×10^6 to 7×10^8. After expands the specificity, the *lineitem* different join key or the output size of the query was kept at 1,500,000 while the data is two times each time. In this execution, we were specified to get the point at
which our index-based proceedure works detectable better than the index-less proceedure on our current multi-node setup.

Fig. 10 describe the graphs for average response times sustained. As we move from case 1 to 8 in this figure, the index-less proceedure grows uncertainly, while the indexbased proceedure remains less or more at an average of about 87 seconds. In case 7, with 45GB of data and 0.3% as query specificity, the index-based proceedure is an order of magnitude speeden than the index-less proceedure. The next iteration, case 8, with double query specificity (0.1%) and double data size (90GB), our proceedure is 20 times faster than the index-less method. The epidemic efforts of the
index-less graph in Fig. 10, started at iteration 6 with 0.7% as the query specificity. If the curve has the same, our index-based proceedure can perhaps be 2 orders of magnitude faster than the index-less proceedure at 45TB of data with very specific (0.0007%) queries.

As shown in Fig. 12, the index size gently drops from 18% of the data size to 9% over the 8 iterations. The Hive index size highs or decreases proportional to the data size . In Experiments 2, the lower index rate is due to the data dispensation, as at each iteration, the number of different values of all attributes, was kept the same while the volume of data was doubled.

In regard to index, in Fig. 12, we can observe that, up to iteration 5, index creation time is a bit less than the performing of Query 1 without index, and surpasses the query run-time later.

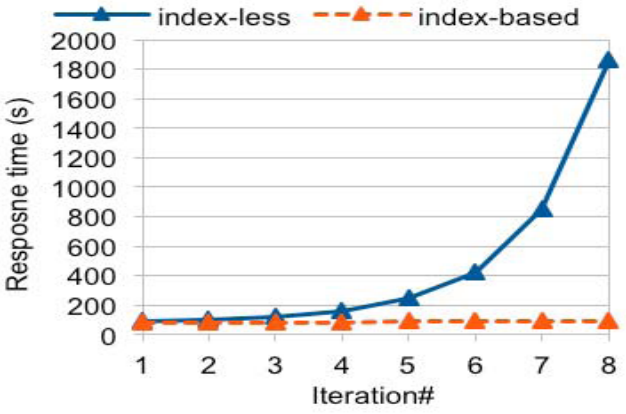

Figure 10. Query 1 response time with/out index on multi-node and single-node setup (Experiments 2).

VI. CONCLUSION AND FUTURE WORK

Indexes have been throughout for long time and the profit of using them is well known. Though, deciding when to use indexes in a scenario requires huge estimation and commutation between its cost and execution. In this exeperiment, we used the present Hive indexing structure to faster up join queries. From researches 1, we have seen, larger the data are, larger the performance gain becomes. Our proceedure grew straight in all cases shown in Figures 4 to 7. In Experiments 2, we expand the sizes of the datasets with increasing specificity ratios. The outputs of these executions said that our proceedure is mounting faster than the present Hive
approach.

We observed in Fig. 8, that the index size was fixed at only 15% of the data size in Experiments 1; and in Fig. 11, it has taken an average of 12% of the data in execution 2. Even though the index size hangs on the data grouping and the number of attributes for indexing, our executions presented the Hive index space usage is logical. Index constructing time graphs illustrated in Figures 9 and 12 has shown the time necessary on building an index rely on the data distribution, the more equivalent tuples gives the output in a minimal index creation process became. In Fig. 11, the maximum time case (iteration 8) index creation took almost twice the query execution time. Index creation contains of reading the whole data, classifying it, and remove the duplicates, which is a bit lengthy process. Till the data in the base table is unwanted, any group of queries that have the right to make use of the index, nevertheless the index construction cost is only sustained once.

With reference to approach the index, present Hive indexes do not give an immediate approach to values, which unquestionalbly comes with heavy space projected. What they provide rather is, examine a huge amount of data that replaces with a highly small set of it that handles the values that are desired for. The cost of detecting a value in the present index Hive is *O(n)*, where n is the number of tuples. Let's suppose a Hive table of n tuples and its index with entries, permitting a particular value in the index is decreased from *O(n)* to *O(m)* with *m* much lesser than *n*.

Hive index maintenance cost is considerably low, Noticing the few updates and batch-mode data insertion as the specifications of big data. If new data are inserted into a new parting of a base table, indexes can be inserted statically for that parting and kept individually without any requirement to execute update operations.

The indexing proceedure in Hive is instead new and the progress has been controlled to present index structure and also the query life cycle. There are a number of optimization ideas to additional raise Hive index-based joins, including:

- Plotting a cost-based optimizer, which can estimate a query plan to help determine to use indexes or not, likely by using column level statistics.

- Auto-indexing or the capable for the compiler to construct indexes inside if demonstrated to be more logical than the brute-force scanning of the data.

- Index creation in which the best index out of all of the presented ones is selected to be used. The best index could be the lesser or the one with the optimal set of attributes. Present Hive naively selects the first relevant index to execute a query plan.

- Not taking index creation time by establish the index when inserting the data into a table. Noticeably, in Hive managed tables data are read twice. One for copying it to the base table and one for creating the index. The previous can be removed if the index can created in the background while inserted data into a table.

- Execution of a hash-based index at the bucket level. Buckets, the smallest data model units in Hive, are probable candidate for the fast hash-based index structure.

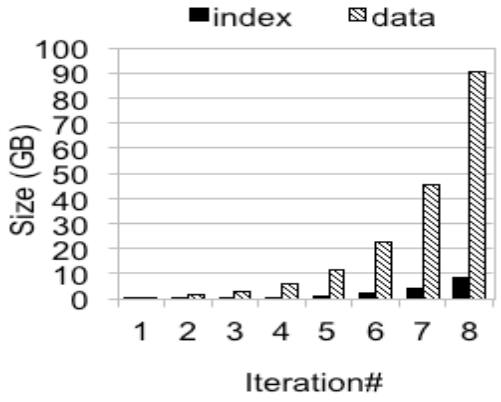

Figure 11 . Index size vs. data size

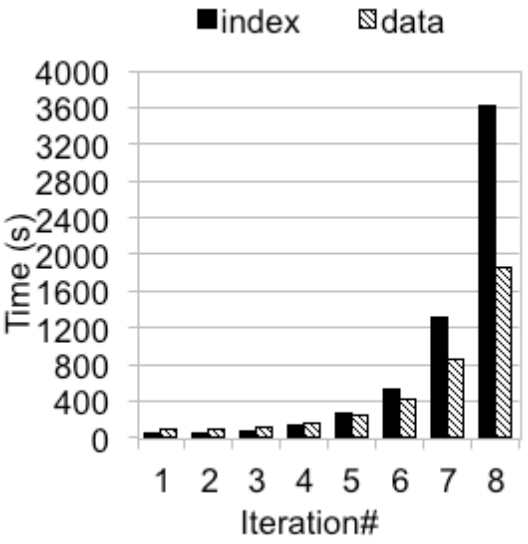

Figure 12. Index creation vs. query response Times

ACKNOWLEDGMENT

This work is supported in part by Department of Computer Science & System Engineering, GITAM University, Hyderabad.

REFERENCES

[1 Apache Hadoop [Online]. Available: http://hadoop.apache.org/

[2] 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

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

[4] 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

[5] 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

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

[7] Valduriez, P. "Join Indicies," in ACM Trans. Database Systems (TODS), vol. 12, issue 2, 1987, pp. 218-264

[8] 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.1–24

[9] 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