INDEX-BASED JOIN OPERATIONS IN HIVE

MAHSA MOFIDPOOR

A THESIS

in

THE DEPARTMENT

of

COMPUTER SCIENCE AND SOFTWARE ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE

CONCORDIA UNIVERSITY

MONTREAL, QUEBEC, CANADA

APRIL 2013

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: *Mahsa Mofidpoor*

Entitled: *Index-based Join Operations in Hive*

and submitted in partial fulfillment of the requirements for the degree of

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

N. Tsantalis	Chair
Dr. D. Goswami	Examiner
Dr. R. Jayakumar	Examiner
Dr. N. Shiri V., Dr. T. Radhakrishnan	Supervisors

Approved by Dr. H. Harutyunyan

Chair of Department or Graduate Program Director

Dean of Faculty Dr. Robin A. L. Drew

Date *22 Apr 2013*

ABSTRACT

INDEX-BASED JOIN OPERATIONS IN HIVE

MAHSA MOFIDPOOR

The exponential growth of data being generated, manipulated, analyzed, and archived nowadays introduces new challenges and opportunities for dealing with the so called big data. Hive is a batch-oriented big data software, well suited for query processing and data analysis. 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. We investigate such techniques for join operations in Hive and develop an index-based join algorithm 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.

In this work, we implement the proposed index-based technique and integrate it in Hive. To add our extension, we obtain Hive architecture details by reverse engineering the code and map our design to the conceptual optimization flow.To evaluate the performance, after setting up the environment, we run relevant test queries on datasets generated using the industry standard benchmark, TPC-H. Our results indicate significant performance gain over relatively large data or highly selective queries.

Acknowledgments

My sincere gratitude goes to my supervisors Dr. N. Shiri and Dr. T. Radhakrishnan. I am deeply grateful since they have granted me the opportunity and confidence to explore my own research interest without opting out of offering guidance and support. Special thanks to Dr. Shiri for his constant engagement throughout the project and him encouraging me persistently to proceed with my research.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the advent of Web 2.0, roles of the users and web applications went through a 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. 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. This opens new horizons for development of novel algorithms, tools, and services to process queries over this huge amount of data in a reasonable time frame.

In this regard, a bunch of terms or even buzzwords have appeared in the related literature. We begin by trying to clear the distinctions and relations among big data, cloud computing, and Not Only SQL (NoSQL) from a technical viewpoint. We will then introduce our chosen use case study, Hive [5], to set up the stage for our work in this research.

1.1 Big Data

Big data spans three dimensions: Volume, Velocity, and Variety, mostly known as three Vs [27], which answer how big the data is, how fast it grows, and how different the structure is, respectively.

1. **Volume**: The data being stored in the World Wide Web (WWW) every day is exploding. The size of this data was reported to be 8×10^{20} bytes and predicted to reach 35×10^{21} bytes by 2020. Twitter generates more than 7 terabytes (TB) of data every day. Facebook does the same at 10 TB level [15: Part I, Characteristics of Big Data].

   A great portion of this gigantic amount of data is not analyzed or even "used," but that does not save us from drowning into it.
2. **Velocity:** Velocity in the data context is the pace at which data is arrived, stored, retrieved, and flown or streamed. Scrutinizing 5 million trade events created each day for fraud detection or analyzing 500 million daily records of call details in real-time to predict customer churn faster gives a flavour of how fast processing is required.
3. **Variety**: Organizations have to draw insights from structured, semi-structured, and unstructured raw data from a variety of sources to make the best decisions. Both traditional and non-traditional data are crucial for enterprises to make decisions. The non-traditional data indeed spans a greater part of the world data [15: Part I, How Fast is Fast? The Velocity of Data].

1.1.1 Big Data Solutions Application

All the issues a big data solution can overcome fascinatingly do not make it as an all-purpose magical replacement for the existing establishments. There are key principles on which a big data solution is determined [15: Part I, When to Consider a Big Data Solution]. For example, the following situations explain the big data solution is the right choice:

1. When sampling of data to analyze the whole data is either ineffective or impossible to achieve,
2. When we need iterative or exploratory analysis of data,
3. When it is profitable,
4. When traditional databases can not solve challenges at hand

Big data tools by no means do not eliminate or replace traditional database technologies. In fact, conventional DBMS technologies are relevant and vital part of an overall effective solution to problems in big data management.

1.2 Cloud Computing

"*Cloud Computing is the long dreamed vision of computing as a utility, where users can remotely store their data into the cloud so as to enjoy the on-demand high quality applications and services from a shared pool of configurable computing resources.*" [26]

Cloud computing is a service-oriented way of computing with abstract infrastructure. Among all the marketing buzzwords describing cloud computing, we consider the following definition from the USA National Institutes of Standards and technology (NIST) that points out five essential characteristics for cloud computing [18].

1. On-demand/self-service: The customers are provided with computing capabilities as per their needs without requiring separate human interactions with the service provider.
2. Broad network access: Services are available over the Internet and can be accessed from a variety of devices (e.g., PCs, laptops, mobile phones, etc.).

3. Resource pooling: Computing resources are pooled to be assigned to multiple customers dynamically using a multi-tenant model.
4. Rapid elasticity: The customers can assume the available resources are unlimited.
5. Measured service: The customers pay only for the exact usage.

The three well known service models of cloud computing are Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). IaaS is about outsourcing all the equipment including storage, servers and network components required for the operations. PaaS goes one level higher and delivers operating systems and other associated system services without download or installation. Finally, SaaS hosts customers' applications through a vendor or service provider.

Cloud is an abstract infrastructure or computing setup, which makes big data accessible to customers. As an example, cloud can host a big data solution, often called Database as a Service (DaaS).

1.3 NoSQL

NoSQL concept literally emerged in 2009 when the designers of Web. 2.0 services realized that RDBMS are best fit for either small yet frequent read/write transactions or for large batch transactions with few write accesses [35]. The term NoSQL is used in contrast to relational databases, to refer to a database system which is distributed, *may not* require fixed table schemas, *usually* avoids join operations, *typically* scales horizontally, *may* not expose a SQL interface, and *may* be open source [35].

Traditional databases, generally, import data slowly into a native representation before they are ready for querying. This makes them not suitable for management and processing of streams of data.

Moreover, traditional databases do not often scale up for “big data” which is at least petabytes of data in volume.

NoSQL databases promise to bring high performance and high availability in the mentioned non-traditional situations. The data model typical to NoSQL databases (discussed in the following section) partially justifies its success.

1.2.1 NoSQL Data Models

Features of the NoSQL data model are as follows:

- Key-Value: The simplest data model, in which data elements’ values are retrieved, is by a (unique) key. Values are un-interpreted byte arrays, which are independent and separated from each other. Therefore, the key-value model is in a way schema-free.

  Definition, management, and interpretation of any data model over key-value format should be done at the application level. Dynamo [2] is a popular example of key-value structured database.

- Column family: Column family does not use the conventional row-store. Instead, an arbitrary number of key value pairs are stored within rows. Multiple versions of values are stored in chronological order to support versioning and more precise consistency.

  Column family stores its data in a table, but the entries are columns instead of rows. Column family does not allow table associations since the value part of the table is meaningless to the system. A popular example of column family solutions is Hbase [6].

- Document: A document is the key-value structure with semantic value part, which means the value part is a meaningful object to the system. The value component, stored in JSON or XML, can hold data types and complex data structures, and hence can be queried as well. Document stores are pretty convenient in data integration and schema migration tasks because they allow multi attribute search on records, which may have completely different kinds of key value pairs. MongoDB [29] is a popular example of document-oriented databases.
- Graph: In graph databases, an example of which includes Neo4j [30], a graph with its nodes, edges, and properties is used to store the data. The main characteristic of graph databases is that adjacent elements of a node are accessed directly without indexes. The strength of this model is efficient management of heavily linked data. More specifically, to perform the recursive join of multiple relations, it uses tree traversals, which are less expensive, compared to the traditional approach like nested loops.

NoSQL databases plan to achieve their goals at the price of weakening the transactions' properties of being Atomic, Consistent, Isolated, and Durable (ACID) that are assumed in traditional relational databases. According to CAP theorem [17], a distributed system cannot have Consistency, Availability, and Partition tolerance simultaneously; it can only support two at a time. Consistency means nodes of a distributed system see the same data at the same time; availability guarantees receiving a response (fail/success), and partition tolerance indicates the system as a whole continues to operate despite a partition(s) failure.

Among all the big data solutions already introduced to the technology globe, we consider working on Hive, a non-NoSQL, relational big data solution, for query optimizations. The next section introduces Hive.

1.4 Hive

Hive is a data warehouse software best suited for OLAP (OnLine Analytical Processing) workloads to handle and query over vast volume of data residing in a distributed storage. The Hadoop Distributed File System (HDFS) [1] is the ecosystem in which Hive maintains the data reliably and survives from hardware failures [12]. Hive is the only SQL-like relational big data warehousing approach developed on top of Hadoop to the best of our knowledge.

A high-level programming model, called *mapreduce framework*, on top of Hadoop enables it to stream the data at a high bandwidth and perform massive computation. HiveQL as described in Section 2.1, is an SQL-like query language for expressing queries in Hive. After a query is issued through an interface in Hive, it undergoes several processing phases including parsing, semantic analysis, logical plan generation, physical plan generation, etc; ultimately the plan is transformed to a sequence of mapreduce operations which are then executed over Hadoop.

1.5 Motivation

Relational Database Management Systems (RDBMS) have robust and established qualities for managing and querying relational data. SQL is the de facto standard language to conveniently query relational data. Ironically, SQL is the biggest missing piece of NoSQL [33: chapter 1, Challenges of RDBMS].

In the context of databases and data warehousing, to achieve the highest level of efficiency in processing, first the primitive operations should be improved. Join as the most expensive operation, makes a critical performance improvement if enhanced.

The join operation allows information from various relations to be “combined.” This facility provides more analysis opportunities to the user.

One of the techniques to accelerate the join computations is using the indexes. Without indexing, the brute-force scanning of the entire data is prohibitive in general for large data. This is especially important when a small fraction of tuples participate in a join operation.

Two major factors influence the indexing approach in Hive to speed up joins:

1. Incredibly high data volume
2. Low index maintenance cost

Hive as a host for handling big data is supposed to work well with vast amount of data. The more data we have, the better the performance becomes. As a result, indexing can promisingly make a huge difference in response time.

Having infrequent updates, is another characteristic of big data, which makes the index maintenance simple and affordable. Additionally, the index types proposed and developed for data in Hive take up a pretty small space, as we will observe in Section 5.5, Experiment 1 and in Section 5.6, Experiment 2.

1.6 Contributions

Our aim in this research is to accelerate join queries with the assistance of suitable indexes. On this matter the contributions of this thesis are summarized as follows:

1. We provide a clear, concise description of the Hive architecture and query life

cycle (Chapter 3). This also provides a basis for better understanding the Hive framework, useful for its further extensions and improvements deemed required.

2. The main contribution of our work is an extension of the query processing in Hive query language (Chapter 4) 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 our experiments show effectiveness of the proposed index-based join technique.

1.7 Thesis Outline

The rest of this document is organized as follows. Following the first chapter in which we introduced Hive, in Chapter 2, we study HiveQL plus Hive indexing component. Afterwards, we will cover Hadoop/mapreduce as the background and foundation of Hive. Chapter 3 explicates the technical details about Hive architecture focusing on the parts that are relevant to this research. Chapter 4 introduces our proposed technique along with the related work on Hive join algorithms and optimizations. Chapter 5 is dedicated to experiments and analysis of the results followed by conclusion and future work in Chapter 6.

Chapter 2

Background

In this chapter, we first review Hive Query Language and Hive data model. We then provide a brief description of the data in Hive including data types, file formats, and loading data into Hive tables. This part can be used as an exposition of our experiments described in Chapter 4.

In the second half of this chapter, we review Hadoop, as it is the underlying system for Hive. We then introduce mapreduce and its implementation of relational algebra operators. We also study join algorithms on mapreduce.

2.1 HiveQL

HiveQL is an SQL-like language, but not fully conforms to it and it is used to express queries over Hive. Unlike standard SQL engines, Hive does not support update, delete, row-level insert operations and does not support transactions.

Hive, as the Hadoop client, is supposed to function with infrequently updated and batch-mode inserted data. These are the characteristics of the underlying system of Hive.

Hive is an OnLine Analytical Processing (OLAP) data warehouse solution and is best suited for large data mined for insights, reports, etc. For Online Transaction Processing (OLTP) features working with big data, one should consider a NoSQL database, good example of which would be Hbase [6].

To start with, we explore the Hive data model. The next section provides an introduction to the Hive Data Definition Language (DDL) followed by a review of the Hive Data Manipulation Language (DML).

As a guide throughout this chapter, using any clause/parameter in square brackets means an optional feature and any parentheses is for more clarity.

2.2 Hive Data Model

2.2.1 Databases

The notion of database in Hive is a catalog or namespace to organize tables. When a database is created, a directory with a user-given name ending with .db is created. All such directories are created under a top-level directory which by default is */user/hive/ warehouse.*

A database can be easily created in Hive using the following command.

```
CREATE (DATABASE|SCHEMA) [IF NOT EXISTS] database_name
[COMMENT database_comment]
[LOCATION hdfs_path]
[WITH DBPROPERTIES (property_name = property_value,...)];
```

A description of the parameters and options in the above command are as follows:

`LOCATION` specifies a custom location other than the default one.

`WITH DBPROPERTIES` are user-defined parameters that can contain any arbitrary meta data about the database. As an example “`created_by = Mahsa`“ stores the name of the database creator. Both `property_name` and `property_value` are strings. Other parts of the command have the same functionality as they do in SQL.

The above description can be pronounced using:

```
DESCRIBE [EXTENDED] DATABASE database_name;
```

Similar to a standard SQL engine, Hive also has `SHOW`, `USE` and `DROP DATABASE` commands with obvious meanings.

```
SHOW DATABASES;
USE database_name;
DROP (DATABASE|SCHEMA) [IF EXISTS] database_name
[RESTRICT|CASCADE];
```

2.2.2 Tables

In the hierarchy of the Hive data model, there exists the database concept at the top. Right after that, we have the notion of tables. Table creation with HiveQL is more extended and elaborate compared to SQL, which will be explained as we proceed. Hive provides two types of tables: managed (or internal) tables and external tables. Managed tables are created by the `CREATE TABLE` command and the data for each table is stored under a subdirectory called */user/hive/warehouse*. Dropping a managed table causes its data to be deleted from the mentioned directory. External tables are created by `CREATE EXTERNAL TABLE…LOCATION`. Unlike in a managed table, dropping an external table does not affect the data. Additionally, external tables do not copy the data; instead they access the data from the location mentioned in the `LOCATION` part, which makes loading the data to a table a fast process.

Here is the Hive table declaration, followed by a detailed description of its parameters and options:

```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.]table_name
[(col_name data_type [COMMENT col_comment], ...)]
[COMMENT table_comment]
[PARTITIONED BY (col_name data_type [COMMENT col_comment], ...)]
[CLUSTERED BY (col_name, col_name, ...) [SORTED BY (col_name
   [ASC|DESC], ...)] INTO num_buckets BUCKETS]
[SKEWED BY (col_name, col_name, ...) ON ([(col_value, col_value,
   ...), ...|col_value, col_value, ...])]
[
[ROW FORMAT row_format] [STORED AS file_format]
 | STORED BY 'storage.handler.class.name' [WITH SERDEPROPERTIES
```

```
      (...)]
   ]
   [LOCATION hdfs_path]
   [TBLPROPERTIES (property_name=property_value, ...)]
   [AS select_statement]
```

The `IF NOT EXISTS` and `COMMENT` have the same functionality as they do in SQL. `LOCATION` and `TBLPROPERTIES` clauses have the same functionality of `LOCATION` and `DBPROPERTIES` described in the previous section, with the possibility of using `COMMENT` at several places in table declaration.

Keyword `AS` at the end of the command is analogous to its SQL counterpart, which is used to mention aliases.

Other points about this command:

- The `EXTERNAL` keyword marks the table as external.
- `PARTITIONED BY` partitions the table based on the column names specified as its parameters. A separate directory is created for each distinct value combination in the partition columns.
- The `CLUSTER BY` along with the option `INTO num_buckets BUCKETS` bucketizes the data on the specified columns. The `SORTED BY` option sorts data in every bucket on the specified columns.
- `SKEWED BY` indicates that the data is skewed over which column and for what values of that column.
- The `ROW FORMAT` shows how the data is organized in the file, i.e., how tuples, attributes and elements of complex data types are separated.
- `STORED AS` is the file type choice to save the table data.
- `STORED BY` defines the serializer/deserializer to read from and write to the underlying

system.

We will provide representative examples, as we proceed, to illustrate applications of these options.

2.2.3 Partitions

The traditional method of distributing data horizontally is also supported in Hive. Partitions are means for re-organizing the data logically so that accessing the data is much faster. Each partition defined for a table is equivalent to a sub-directory under the directory where the table is located.

Both managed tables and external tables can be partitioned. In managed tables, partitions are defined at the table creation time and data is loaded afterwards to each partition using `SELECT` or `LOAD` commands. In external tables, to partition a table, the user has to omit the `LOCATION` in the `CREATE EXTERNAL TABLE` and each partition should be added to the table separately using:

```
ALTER table_name
ADD partition (partition parameters and values)
LOCATION location;
```

2.2.4 Buckets

Bucketing is another approach to segregate datasets into more manageable parts. Bucketing is useful to avoid having a lot of small partitions that may overwhelm the file system. In addition, unlike partitions, bucketizing provides means for data sampling and bucket map join. This is described in Chapter 4.

Buckets are defined at the table creation time, however it is the responsibility of the user to insert data correctly into each bucket.

Here is an example to define buckets and load data appropriately into them.

```
CREATE TABLE students (id INT, name STRING, course STRING)
CLUSTERED BY (id) INTO 12 BUCKETS;
SET hive.enforce.bucketing = true;
FROM raw_students
INSERT OVERWRITE TABLE students
SELECT id, name,course;
```

The parameter `HIVE.ENFORCE.BUCKETING` instructs the system to load the data into buckets. If it is not set, the data will not be distributed into the buckets.

2.3 Data in Hive

2.3.1 Hive Data Types

Hive defines the common relational databases data types as well as the three collection types: STRUCT, MAP, and ARRAY.

Collections are not supported in most relational databases because using them tends to break the normal forms. The consequences of breaking the normal forms would be data duplication that may lead to space waste and inconsistencies. In big data, however, sacrificing the normal forms allows efficient processing of complex data types within a record.

Hive gives the users full control over the data persistence and its life cycle. This allows management and processing of data with variety of tools, i.e., Hive can be bound with various data sources.

Since Hive itself is implemented in Java, the exact behaviour of each data type is the same as the corresponding data type in Java.

2.3.2 Hive File Formats

Hive is intended to work in batch mode. As a result, a statement like `INSERT INTO table_name VALUES(values)` does not make sense in Hive. Instead, big amount of data is read from a local or remote source and then loaded in one shot into Hive tables. Hive is able to process data coming from various sources using Extract, Transform and Load (ETL) tools by supporting reading data from different file formats and their customization. This section explains the `STORED AS` and `ROW FORMAT SERDE` parameters of Hive table declaration statement and gives a brief description of the type of files Hive can read from and how they should be managed.

2.3.2.1 Text file

The simplest file format supported in Hive is a text file. Hive uses control characters that are less likely to appear in a text to separate fields, lines and the components of a complex data type. The following example shows use of delimiters to distinguish between rows, fields of a struct, items of a collection, elements of a map and lines.

```
CREATE TABLE students (
      name STRING,
      id  INT,
      courses ARRAY<STRING>,
      grades MAP<STRING, FLOAT>,
      address STRUCT<street:STRING, city:STRING, state:STRING,
      zip:INT>
                      )
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\001'
COLLECTION ITEMS TERMINATED BY '\002'
MAP KEYS TERMINATED BY '\003'
LINES TERMINATED BY '\n'
STORED AS TEXTFILE;
```

2.3.2.2 Non-Text files

While text files are the most suitable choice for sharing data and viewing in an editor, binary files are more space-efficient which is critical for big data. Examples of such files are SequenceFile and RCFile (Record Columnar File).

Sequence file is the standard Hadoop file format which is a flat file consisting of binary key-value pairs.

RCFile stores columns of a table in a columnar way. It first partitions rows horizontally into row splits and then each row is vertically partitioned on columns way. RCFile first stores the meta data of a row split, as the key part of a record, and as the value part it stores the data of a row split.

Below is an example of how a column-oriented table is created and loaded using a row-oriented table.

```
CREATE TABLE columnTable (key int, value int)
ROW FORMAT SERDE
'org.apache.hadoop.hive.serde2.columnar.ColumnarSerDe'
STORED AS
  INPUTFORMAT 'org.apache.hadoop.hive.ql.io.RCFileInputFormat'
  OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.RCFileOutputFormat';
FROM anyTable
INSERT OVERWRITE TABLE columnTable
SELECT anyTable.col1, anyTable.col2;
```

SerDe (Serializer/Deserilizer) converts the unstructured bytes in a record into a record that Hive can understand or converts a Hive record to bytes suitable for writing to a table. Hive uses the `INPUTFORMAT` along with the de-serializer to read from a table and uses the `OUTPUTFORMAT` with the serializer to write to a table respectively.

2.3.3 Loading Data into Tables

As a requirement of big data, Hive has no row-level `INSERT`, `UPDATE`, or `DELETE` statement. Data is only inserted into tables through bulk load operations or simply batch-mode.

As mentioned before, for external tables the data is loaded using the `LOCATION` clause in `CREATE TABLE` statement. In managed tables, after a table is created, we can load data into it using the command `LOAD DATA [LOCAL] INPATH`, illustrated in the following example:

```
LOAD DATA LOCAL INPATH '${env:HOME}/CS-students'
OVERWRITE INTO TABLE students
PARTITION (dept = 'CS');
```

The above command, loads the data from the specified directory mentioned in the `INPATH` part to the specified partition. Obviously, if the table is not partitioned, the user is not supposed to declare it. It is important to note that once the data is loaded, any data already stored in the table is first deleted before the new data is loaded, unless the parameter `OVERWRITE` is omitted which leads to appending the data.

The keyword `LOCAL` instructs Hive system to copy the data from the directory `CS-students` in our example, to where the table is located in the distributed file system. If `LOCAL` is omitted, the data is just moved to the location of the table. In other words, `LOCAL` chooses between working with a copy of the data or the original one. This becomes crucial when the user decides to drop the table.

The result of a query can also be inserted into a table:

```
INSERT OVERWRITE TABLE students
PARTITION (dept = 'CS')
SELECT * FROM CS-students;
```

Here also if the user wishes to append the data the parameter `OVERWRITE` should be left out or replaced with the keyword `INTO`.

2.4 Index Related Commands in Hive

An index is an extra data structure associated with a table in the database that allows retrieving defined by a query the records more efficiently using only one or more fields of a table input.

Indexing is a technique that provides a faster means to access a portion of the data, rather than the whole data, based on the column values. This technique is effective when the selectivity of the query is “high enough.” The threshold for using the index purely depends on the data size, data distribution, index size, and its cost, including creation time and storage utilization.

Supporting indexing in Hive started in 2009 in order to speed up processing simple queries having a predicate expressing a condition over individual columns (no aggregation function on column values). Later on, bitmap indexes were considered for the same type of queries, which worked well when the columns had small number of distinct values.

Index handlers are JAVA classes that implement indexing. The index handlers implemented use a tabular format to store the index, i.e., they create a table even though other structures are claimed to be supported [21].

Hive requires the user to create indexes manually. Indexing, as a reasonably new mission in Hive provides few options. Alternatively, the design is intended to be customizable, meaning that a desired index code can be easily plugged to extend the functionality. Indexes help to prune the data especially when partitions are plentiful and tiny; indexes are a better substitution for table partitions.

Not all Hive statements can benefit from indexes. Indexes so far are used in queries with a `WHERE` clause or a `GROUP BY` clause. The `EXPLAIN` statement can be used to verify if a Hive command can potentially use the index.

2.4.1 Index Construction in Hive

Here is how we can create an index in Hive:

```
CREATE INDEX index_name
ON TABLE base_table_name (col_name, ...)
AS 'index.handler.class.name'
WITH DEFERRED REBUILD
[IDXPROPERTIES (property_name=property_value, ...)]
[IN TABLE index_table_name]
[PARTITIONED BY (col_name, ...)]
[[ROW FORMAT ...] STORED AS ...| STORED BY ...]
[LOCATION hdfs_path]
[TBLPROPERTIES (...)]
[COMMENT "index comment"]
```

There are some parameters and keywords used in the command listing:

1. `index_name` : The index name given by the user used to access the index by the user itself. This name is used to refer to the index in `ALTER INDEX` and `DROP INDEX` commands.
2. `base_table_name (col_name, ...)`: The table over which the index is to be created using the desired columns listed.
3. `'index.handler.class.name'`: this specifies the type of the index, which could be bitmap or compact values.
4. `index_table_name`: The index name given by Hive (the default name) or the user, used to access the index as a table by the user or Hive. For example the user can see the content of an index using this parameter (Figure 1). Any other behaviour of the index as a table can be addressed using this name.
5. `WITH DEFERRED REBUILD`: This means the index starts empty. Index will be populated using:

```
ALTER INDEX index_name ON table_name [PARTITION (...)] REBUILD
```

6. `IDXPROPERTIES` : This gives the properties of the index. For example: `'index_creator' = 'Mahsa'` can be considered a property for an index.
7. `IN TABLE index_table_name:` This is used when the user wishes to build an index in a different table from an already built index (if any), to keep them separate.

Other possible options are the same as the one used in Hive `CREATE TABLE`.

Indexes are built in two phases. In the first phase, index layout and variables are validated at the index creation time. Examples include checking if the column being indexed exists in the input table, or checking if the partitions are valid, etc. In the second phase, the index table is loaded with relevant data, which takes a major time of building an index.

If `PARTITIONED BY` clause is omitted from the `CREATE INDEX`, index spans all the table partitions. Index partitions and table partitions do not have necessarily the same level of granularity [10, chapter 8, Creating an Index]. The following examples show that table partitions and index partitions could be different:

```
CREATE TABLE students(Id int, Name string, Dept_id int, Major_id
int)
PARTITIONED BY (Dept_id int,Major_id int);

CREATE INDEX students_index ON TABLE students
AS 'compact' WITH DEFERRED REBUILD
PARTITIONED BY (Dept_id);
```

The data must be scanned thoroughly and sorted to be imported to the index. If data in the base table changes, then the `REBUILD` command is used to bring the index up to date. This is an atomic operation, so if the table was previously indexed, and a rebuild fails, then the stale index remains intact.

We have provided an example of creating an index and a table at the same time.

Here is the index creation command we have used in our experiments:

```
CREATE INDEX x
ON TABLE lineitem (L_ORDERKEY)
as 'compact' WITH DEFFERED REBUILD;


ALTER INDEX x ON lineitem REBUILD;
```

Note that, `L_ORDERKEY` in `lineitem` has a large number of values, the index type bitmap is not a suitable choice in our experiments, instead it is useful when there are few distinct values among the index key.

2.4.2 Showing Indexes

The following statement can reveal indexes over a table:

```
SHOW [FORMATTED] INDEX[EX] ON TABLE table_name;
```

Column titles of an index table do not appear in the output of the above command unless the parameter `FORMATTED` is mentioned.

2.4.3 Dropping an Index

An index can be dropped using the command:

```
DROP INDEX [IF EXISTS] index_name ON table_name;
```

Upon dropping a table or a table partition, the corresponding index is also dropped but the opposite does not happen.

2.4.4 Hive Index Structures

In the relational databases context, there are a number of structures that can hold the index. They include [16: chapter 13]:

1. Primary indexes on sorted file
2. Secondary indexes
3. B-trees

4. Hash tables

The primary indexes have key-pointer pairs in an index file. A search key is associated with a pointer to a record holding that search key. Primary indexes are either dense meaning there is one entry in the index for every record or sparse in the sense that there exist an entry in the index file for every block of records. The key point is that the data file is sorted on the search key and the index only determines the location of the desired records.

Secondary indexes find the location of the desired records in an un-sorted file and they are always dense with the same structure described for primary indexes. Compared to the primary index, secondary index requires more disk I/Os as is in general a result of the un-sorted data.

In commercial systems, B-trees and its most common variant B+ trees organize data in a balanced tree. B-tree has the advantage of searching a key at a time proportional to the height of the tree with minimum number of disk I/O operations.

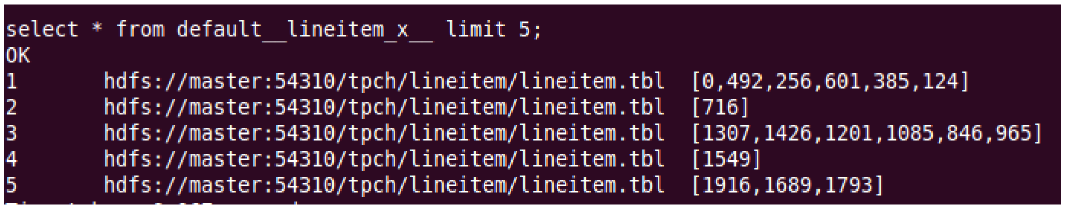
In a Hash table, a hash function takes a search key as an argument and computes an integer in the range 0 to B-1, where B is the number of buckets. This integer is the index of an array that holds the headers of B linked lists containing the records. Each search takes an ideal 1 disk I/O operation.

Current Hive index structure is pretty analogous to the secondary indexes or sometimes the primary index once the data is sorted on the search key/index key as will be explained shortly.

Presently, Hive supports two types of indexes, *compact* and *bitmap*. The corresponding handlers in both index types store the index in a tabular format, i.e., indexes are stored in tables.

The compact index, as the name suggests, builds a compressed index separate from the data. This means that rather than storing the HDFS location of each occurrence of a particular value, it only stores the addresses of HDFS blocks containing that value. This is optimized for point-lookups when a value typically occurs more than once in nearby rows; the index size is kept small since there are many fewer blocks than rows. The trade off is that extra work is required during query processing in order to filter out the other rows from the indexed blocks, but as we will see in Section 5.5, that extra search is not considerable.

There are only a few columns in a compact index, which include the name of the column(s) on which the index is built, the block followed by a string column "_bucketname" (indicating the name of the file containing the indexed block) followed by a column "_offsets array<string>" (indicating the block offsets within the corresponding file). As an example of the compact index content, Figure 1 provides a sample of the data in the index we used in our experiments. The first column shows the values of the attribute `L_ORDERKEY`, the second column is the HDFS location, and the third column is an array of offsets from the file mentioned in the former column.



```
select * from default__lineitem_x__ limit 5;
OK
1       hdfs://master:54310/tpch/lineitem/lineitem.tbl  [0,492,256,601,385,124]
2       hdfs://master:54310/tpch/lineitem/lineitem.tbl  [716]
3       hdfs://master:54310/tpch/lineitem/lineitem.tbl  [1307,1426,1201,1085,846,965]
4       hdfs://master:54310/tpch/lineitem/lineitem.tbl  [1549]
5       hdfs://master:54310/tpch/lineitem/lineitem.tbl  [1916,1689,1793]
```

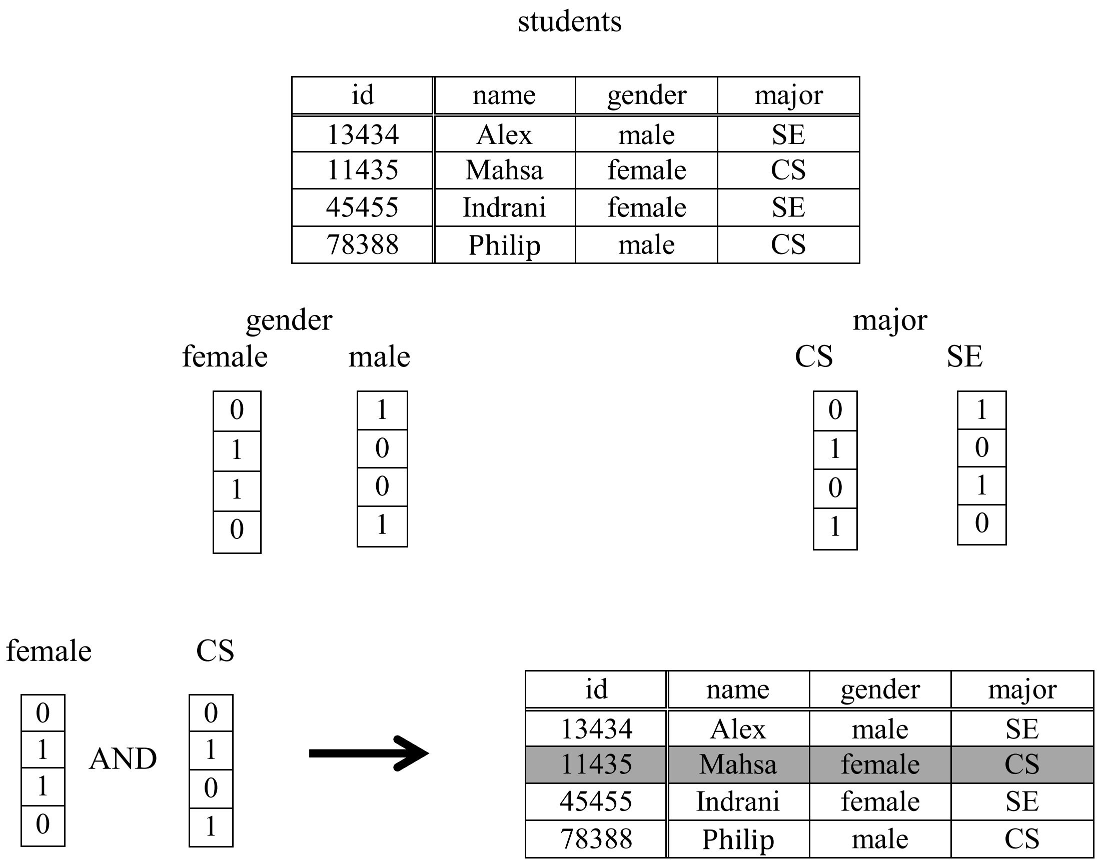
Figure 1 Compact index structure

Similar to compact indexes in coding technique, bitmap indexes are created for columns with few distinct values, such as gender. Bitmap operations are then used to quickly identify rows that satisfy a combination of conditions on such columns.

Bitmap index has the same columns as the compact index in addition to a number of binary bit vectors used to represent the value of the indexed column. The index uses one bit vector for each possible value of the column. Each value in the vector represents a row and is set to 1 if the value is present in the row, and set to 0 otherwise.

As an example of the bitmap indexes (Figure 2), consider the following query, and assume we have bitmap indexes on `gender` and `major`:

```
SELECT *
FROM CS_students
WHERE gender = 'female' AND major = 'CS';
```



id	name	gender	major
13434	Alex	male	SE
11435	Mahsa	female	CS
45455	Indrani	female	SE
78388	Philip	male	CS

female	male	CS	SE
0	1	0	1
1	0	1	0
1	0	0	1
0	1	1	0

Figure 2 Bitmap index example

The bitmap indexes over columns `gender` and `major`, which have few values, are fed to the logical AND to produce the results.

As most of the HiveQL syntax is similar to MySQL, we only focused on the portion of HiveQL that is very different from SQL and we used in our work. The reader is referred to [10: chapter 4-8] for a full description about HiveQL.

The indexing component was covered thoroughly as it is used in our work of providing further improvement in Hive indexing techniques.

The next section introduces Hadoop, which is the foundation of Hive.

2.5 Hadoop

Hadoop is a platform for storing big data economically and reliably. In addition, it can use its powerful mapreduce algorithms to process big data. We begin this section by an introduction to modern distributed file systems and proceed with Hadoop.

2.5.1 Distributed File systems (DFS)

In contrast to single processing computer systems, called "compute node," which consists of a sole processor with its main memory, cache, and a local disk, parallel processing is performed on special-purpose computers with specialized hardware. This pattern has changed to use more commodity machines as "compute nodes" that work relatively independently. The strategy used in case of crashes is replication of data and dividing a job to independent tasks.

In this new environment, files have also different characters and behaviours. DFS is typically used when the files are large (terabyte in size) and updates do not happen or append-only updates are done, since otherwise *if the data is small or it changes quickly, using a DFS is pointless*.

Each file is divided into chunks (commonly 64 MB) that are replicated on different compute nodes of different racks. The association between these chunks and their location is stored in a small file, called *master node* or name node. This file is also replicated. A directory for the entire file system as a whole keeps track of where these copies are saved. The directory itself can be replicated too.

Examples of popular file systems exploiting the described architecture including:

1. GFS (Google File System), the pioneer
2. HDFS (Hadoop Distributed File System)

2.5.2 Hadoop Distributed File System (HDFS)

HDFS forms the underlying structure of Hive that hosts Hive data and also executes the queries. The storage, computation capacity, and I/O bandwidth can be enhanced crudely by adding more commodity hardware. Here are the key characteristics of HDFS:

1. Quick failure detection and automatic recovery
2. High throughput of data access
3. Handling computation over large datasets
4. Ability to append files
5. Portability

In the next section, we describe the HDFS architecture in brief.

2.5.2.1 HDFS Architecture

2.5.2.1.1 NameNode and DataNodes

HDFS architecture conforms to the traditional master/slave style. In each cluster, the master sever, called NameNode, maintains the file system namespace and manages access to the data. Slaves or the DataNodes are the machines that store the data and execute operations like opening/closing a file.

The NameNode and DataNode are pieces of software designed to run on commodity machines, often running GNU/Linux operating system, rather than super computers.

2.5.2.1.2 Block Placement

HDFS uses data replication as the fault tolerance technique. The NameNode receives reports from DataNodes periodically or on-demand to determine if data is under-replicated to replicate it.

The default HDFS block placement targets at minimizing the write cost, maximizing data reliability and availability and economizing the bandwidth. For this, HDFS puts the first replica of a block on the node where the writer is located. The second and third replicas are placed on two different nodes in different racks and the remaining replicas are placed randomly in such a way that no DataNode holds more than one replica and when the number of replicas is less than twice the number of racks, at most two replicas are placed on the same rack [12]. In case of a read, HDFS chooses the closer replica to the reader.

2.5.2.1.3 HDFS Failures

There are two files residing on the NameNode local file system that hold critical meta data: a transaction log file and the entire namespace file. Each time the NameNode starts, it applies all the changes from the transaction log to the in-memory namespace file and writes it to the disk and then truncates the log file.

These two vital files are replicated on the NameNode and all replicas get updated synchronously when a replica is modified.

NameNode is the only single point of failure in a Hadoop cluster for which the solution requires manual intervention.

In management of big data, the environment in which huge datasets can be stored and retrieved is the main issue to address. The next step is to perform computation on multi-tera-bytes of data in parallel.

2.5.2.2 Mapreduce

Inspired by the map and reduce functions in commonly used functional programming languages, *mapreduce* framework is implemented to do huge calculations that is tolerant of hardware failures with a simple programming model. Only two functions *map* and *reduce* need to be written and thereafter the execution flow is as follows:

2. One or more chunks of data are given to some map tasks. Map tasks transform the input to a sequence of key-value pairs. The way these pairs are produced is code-dependent.
3. Key-value pairs are collected by master controller and sorted by the key. Pairs are distributed to reduce tasks in such a way that pairs holding the same key go to the same reduce task.
4. The reduce tasks combine all the values for that key. The way combine works is code-dependent too.

   In Figure 3, the mapreduce execution in Hadoop is depicted in more details. The user program forks a master controller process and some number of worker processes at different compute nodes. A worker is either a mapper or a reducer. Based on the parameters provided by the user program, master creates some number of map tasks and reduce tasks and assigns them to the workers. A chunk can be given to a single map task, but it can be given to one or more reduce tasks because the number of output files that is equal to the number of reducers cannot exceed the limited space capacity. The master also knows the status of each worker whether it is idle, executing, or completed. When a task is completed, the worker informs the master so that the master assigns it a new task.

Each map task executes the code (functionality provided) on its chunk(s) of data and creates a file for each reduce task on its local disk. The master knows the size of the file, its location, and the reducers to which they are destined. When a reducer begins execution, all these files become accessible to it. After the reduce task completes the execution of the code, it writes the output to a single file in the file system.

Depending on where a node failure happens, the recovery time varies. Master node failures are the most severe one for which the whole mapreduce job has to be restarted. Hadoop has a backup master node (secondary name node) to survive in case of a failure. If a mapper fails, all the tasks have to be redone, even those completed, since the output of the map task became unavailable to the destined reducers. When the tasks are restarted on another node, the location of the output is reported to the reducers. Failures in reducers only involve restarting the task on another reducer.

2.5.2.2.1 Map Task

A chunk consists of elements, which can be of any type (tuple, document, etc.). All inputs to mappers and outputs of reducers are key-value pairs. Normally the key part of the input is not relevant which we ignore. Assuming that every input/output is in the form of key-value pairs is necessary for the sequence of mapreduce operations to process.

Based on the number of reducers, say r, the master controller picks a hash function say *mod*, which when applied to keys, produces a bucket number from 0 to r-1. Each hash key, that is the key part of the map task output, is hashed to select specific reducers out of reducer_0 to redicer_{r-1} and then the key-value pairs are put on the local files. These local files are destined for the determeined reducers. This process is called “grouping.”

When all the map tasks are finished, the master controller merges the file from each map task that is destined for a particular reduce and feeds the merged files to that reducer as a sequence of key-list of values pairs, like (k, $[v_1,v_2,\ldots,v_n]$). This process is called “aggregation.”

2.5.2.2.2 Reduce Task

The reducer is responsible for combining the values associated with a key in some way. The combine operation is commonly commutative and associative, i.e., combining can be accomplished in any order without affecting the input data. When the order does not matter, some of the operations that the reducer is supposed to do can be pushed to the mapper. Nevertheless, the reducer cannot be eliminated because pairs coming from different mappers can also have the same key. The output of the reduce is the key along with the combined values. The outputs from all the reducers are merged into a single file.

2.5.2.3 Mapreduce and Relational Algebra

A relation or table, however large, can be stored as a file in a distributed system with its elements behaving as the tuples or rows.

Standard database query primitives such as selection, projection, set operations, natural joins, and grouping/aggregation operations can be implemented within the mapreduce framework. The next section explains how each operation is developed over mapreduce.

Before that, there are a few considerations about mapreduce to be always kept in mind, as

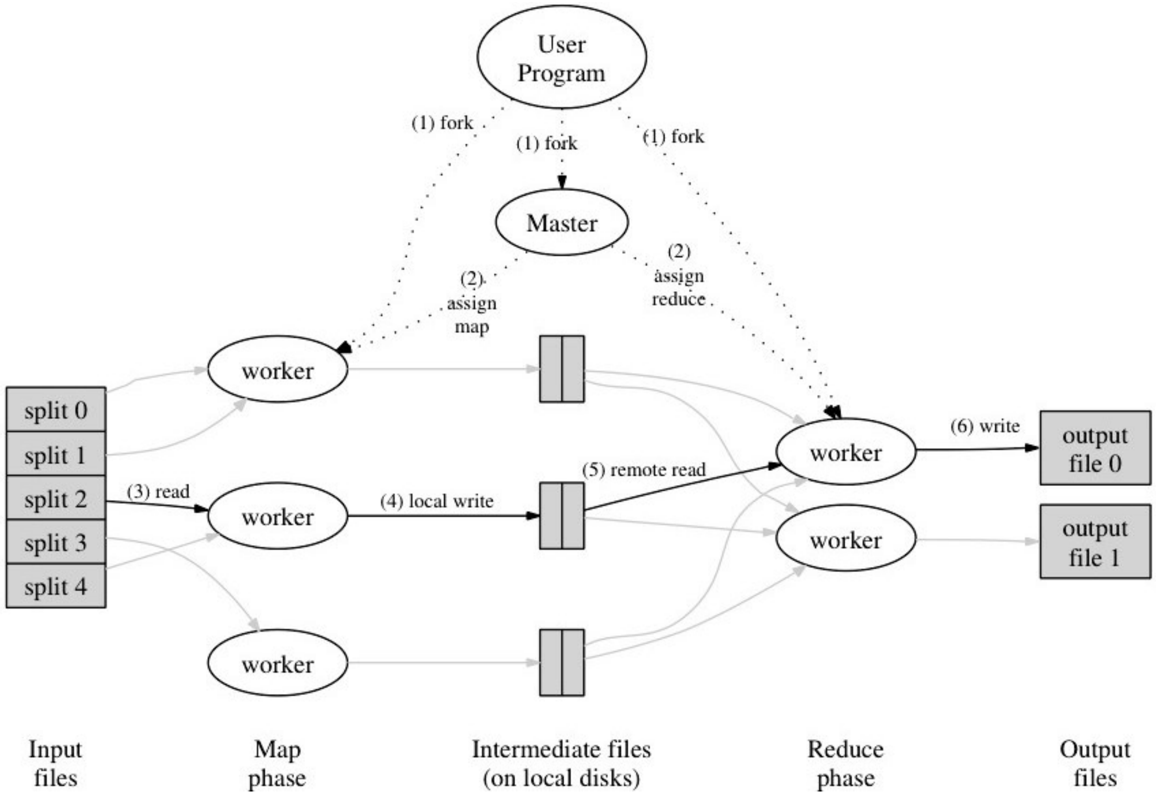


Figure 3 Mapreduce execution flow [14]

follows. At the outset, input and output of the map and reduce functions are in the form of key-value pairs except for the input of the map which may be simply ignored. Examples include the first time we feed the untouched data into map functions. Second, basically map function transforms the input (tuple) to a pair of tuples. Finally, the reduce function input is of the form (key, list of values) [32].

Table 1 describes the notations in relational algebra used in the pseudo-codes that follow.

Variable	Type
R and S	Relation
t and t'	Tuples in either S or R
c	Boolean condition
s	A list of desired attributes for projection
A, B, C	Columns/Attributes in R and S
(a,b), (a,b,c), (b,c)	Tuples containing values for attributes A, B, and C
θ	Aggregation function

Table 1 Relational algebra notation for mapreduce

2.5.2.3.1 Selection σ_c(R)

Selection can be described as a map-only or reduce-only job. The reduce function above is just an identity function, that is, f (x) = x. Note that the output of the selection is only the key part or the value part of the mapreduce output.

- Map function:

```
for each t in R
   if (c)
      emit (t,t)
```

- Reduce function:

```
for each (t,t)
   emit (t,t)
```

2.5.2.3.2 Projection π_s(R)

- Map function

```
for each tuple t in R
   emit($t'$, $t'$)
```

- Reduce Function:

```
    for each $t'$ key in ($t'$, [$t'$, $t'$,… $t'$])
   emit ($t'$, $t'$)
```

t' is a tuple obtained from t by taking only the values corresponding to the attributes listed in s. The elimination of the duplicates of tuples obtained by the reduce function can be done at the map phase too, but for excluding identical tuples coming from different mappers, the presence of a reduce function is essential. In other words, if duplicate elimination is performed at the mappers, in each mapper there is no duplicates, but the final result obtained by merging the outputs of all the mappers may contain duplicates. Therefore, excluding the duplicates at the reduce side is indispensable.

2.5.2.3.3 Union R∪S

- Map function

```
for each t in R
   emit (t,t)
for each t in S
   emit (t,t)
```

- Reduce Function

```
for each t in (t,[t,t]) or (t,[t])
   emit (t,t)
```

Note that here the relations R and S should be compatible. For the set union operation, the reduce removes the duplicates. If the input of the reduce is of the form (t,[t,t]), it means the tuple exists in both relations. On the other hand, (t,[t]) indicates that the tuple belongs only to one of the two relations.

2.5.2.3.4 Intersection R∩S

- Map function

```
for each t in R
   emit (t,t)
for each t in S
   emit (t,t)
```

- Reduce Function:

```
for each t in (t,[t,t])
   emit (t,t)
for each (t,[t])
   emit(t, NULL)
```

Since any input/output is of the key-value form, (t, NULL) is generated when no tuple is generated, that is, neither of the relations contains it.

2.5.2.3.5 Set Difference R-S

- Map function:

```
for each t in R
   emit (t,R)
for each t in S
   emit (t,S)
```

- Reduce Function:

```
for each t in (t,[R])
   emit (t,t)
for each t in (t,[(R,S), (S), (S,R)])
   emit(t, NULL)
```

For set difference, we distinguish between the tuples coming from R from the ones coming from S. (t,[R]) means the tuple *t* only belongs to R and is supposed to be in the output. In any other case, no tuple is generated in the final result.

2.5.2.3.6 Natural Join R⋈S

Considering relation schemas R(a,b) and S(b,c) with the common attribute *b*:

- Map function

```
for each (a,b) in R
   emit (b, (R,a))
for each (b,c) in S
   emit (b,(S,c))
```

- Reduce Function:

```
for each b in [(b, (R,a)), (b,(S,c))]
     emit all combinations of (a,b,c)
```

The map function re-organizes the tuples in a way that the key part is the common attribute *b* and the value part is the rest of the tuple. The reduce function receives all the tuples from both relations, sorted on the common attribute that allows performing the join efficiently. In the reduce function, for each value of *b* if the pairs come from different relations, this is determined using the value part of the pairs, the pairs are combined to be in the output.

2.5.2.3.7 Grouping and Aggregation $\gamma_{A,\theta(B)}$(R)

- Map function:

```
for each (a,b,c) in R
   emit (a,b)
```

- Reduce Function:

```
for each a in (a,b)
   emit (a ,$\theta$(b))
```

Notice that, each ‘a’ in pair (a,b) represents a group and each ‘b’ is a list of b values in the pair.

The standard relational operators can be extended in a straightforward way to support more number of relations and attributes. For generalization, each of these components mentioned in the implementation above should be replaced by a list of components of the same type.

In the subsequent section, we provide details of joins implemented over mapreduce, by considering different types of join algorithms and through examples.

2.5.2.4 Join Algorithms in mapreduce

It is essential to mention that join algorithms developed for RDBMs are not appropriate to execute over mapreduce. Mapreduce join algorithms have a different framework, as we will discuss later on in the current section.

The join algorithms suitable to run for mapreduce in general fall into the three categories [11]:

1. Reduce-side joins
2. Map-side joins
3. Broadcast joins

If the join is performed in the map phase, it is a map-side join, and if done in the reduce phase, it is called the reduce-side join. Broad-cast join is a more efficient map-side join.To explain these two algorithms, we start with the reduce-side join since it makes use of both the map and the reduce phase (contrary to map-side join).

We preferred to keep this classification as simple and as generic as possible. At the end of this section, we will consider its variations.

2.5.2.4.1 Reduce-side joins

Since the actual join happens in the reduce phase, the ‘map’ phase only pre-processes the tuples of the input tables in order to organize them in terms of the join key.

2.5.2.4.1.1 Map Phase

The map function reads one tuple at a time from both of the input tables via a stream from HDFS. The key part of the key-value pairs is the join attribute, and the value part consists of a tag that identifies the input table along with the rest of the tuples being fetched.

The output tuples of the map phase are partitioned based on their keys. It is useful to send all the tuples with the same key to the same reducer. Partitioning and sorting are done simultaneously.

2.5.2.4.1.2 Reduce Phase

The result of the map phase is already sorted (primary sort) on the join key. However, a secondary sort on the tag is done before the reduce function is called.

The reason for having a secondary sort is that, after the data is read through an HDFS stream, the connection is closed and therefore there is no access to the values. This then requires buffering the values. When the data is sorted on tag, which is associated with the base table, it is possible to read all the tuples in the first dataset in the HDFS stream but read the second table one tuple at a time.

Figure 4 shows an example of reduce-side join for the following query in mapreduce:

```
SELECT Employees.Name, Employees.Age, Department.Name
FROM Employees INNER JOIN Department ON
Employees.Dept_Id = Department.Dept_Id;
```

The tables `Employees(Name,Age,Dept_Id)` and `Department(Dept_Id,Dept_Name)` are joined on the attribute `Dept_Id`. The map phase and the reduce phase are separated by a horizontal dotted line. In the map phase initially the tuples are transformed into the key-value pairs. Then pairs sharing the same join key form a partition. In the reduce stage, each reducer receives only the pairs having equal join key values in both tables. The pairs are sorted on the table tags and then the pairs coming from different tables are joined and the result is produced. Needless to say that, in each reducer there is a one-to-one (reducer3), one-to-many or many-to-many or many-to-one join (reducer 1), no tuple is produced or there is only one or more pairs originating from one of the tables for which nothing is emitted.

As evident from the description above, the costs or drawbacks involved including sorting, tagging tables, and probably skewed data that causes workload imbalance among the reducers [8].

This join strategy is close to the partitioned sort-merge algorithm in the parallel RDBMS context and is called Repartition Join in [9].

2.5.2.4.2 Map-Side Join

The sorting step described above in the reduce-side join is a time-consuming operation that can be skipped if the data is already sorted.

The reduce-side join seems like the natural way to join tables using Map/Reduce. Hadoop offers another way of joining tables without using reducers. This allows a faster join, however requires that (1) all the input tables be sorted in the same order on the join key. This is simply for performing the least number of comparisons on the join key. (2) All the input tables use the same partitioner module with the same algorithm and parameters. (3) The number of partitions of the input tables must all be the same. A given key must be in the same partition in both tables so that all partitions having the same key are joined together. This is why the partitions in both tables have to be identical in terms of the number of partitions and partitioning algorithm. Hadoop default practitioner is a hash-based one that builds the partition on the join key or set of join keys. The number of partitions is equal to the number of reduce tasks for the job and can be set in the Hadoop job configuration file [28].

What if the data is not sorted? The conditions can be simply satisfied with passing all the input data through a basic Hadoop job. The Hadoop job partitions, groups, and sorts the data without doing any heavy processing. The rest of the join process is the same as the one mentioned earlier.

The map-side join eliminates the sorting and shuffling (distributing map outputs to reducers) and performs better than the reduce-side join in terms of response time; though we should consider the cost of running additional mapreduce jobs to prepare the data if the requirements are not met.

Map-side join is quite popular in DFS. The reason is that, unlike in a parallel RDBMS, where data is located near to the computation module, there is no guarantee that the tables we wish to join are located on the same node. In other words, the NameNode makes independent decisions over where to put data blocks. As a result, at the query processing time if the data is not on the local machine, it is first loaded into a hash table for faster access time.

Map-side join is referred to as Directed Join in [9].

2.5.2.4.3 Broadcast Join

Broadcast is a map-only algorithm. If one table is small enough to fit in the memory, it is loaded into the memory. The map function is then called for each tuple in the bigger table, one at a time.

The map function probes the in-memory table and finds the matching tuples. Loading the small table into a hash table can further speed up this process. This approach is called memory-backed join in [8].

Hive leverages all these implementations within its own constraints and introduces its supported joins that we will discuss shortly.

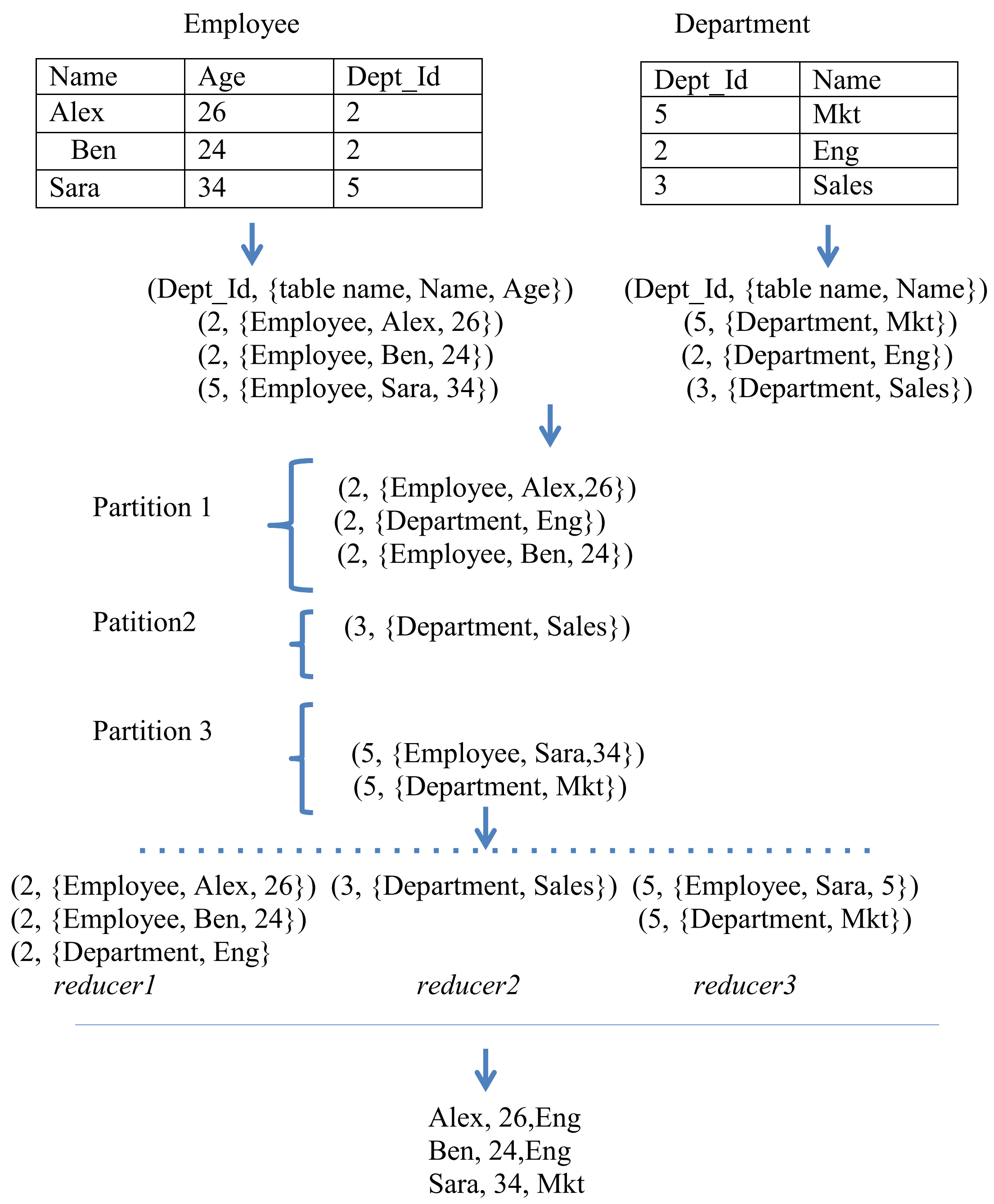
Our index-based join implementation targets the Hive compiler and optimizer, thus it leaves the underlying mapreduce framework intact.

Blanas et al. added another join strategy to the classification above [9]. It introduces semi-join that tries to prepare the distinct values of the join key of one table and extract values from the other table if there is a match in the distinct value list. It organizes three consequent mapreduce jobs (one reduce-side and two broadcasts) to perform the join. This design avoids the bandwidth needed to produce un-matched pairs at the cost of extra scanning.

Another work [8] introduces a hybrid approach for join that accomplishes the partitioning of the mappers' outputs partly and avoids tagging the tables to improve the performance.

Understanding the details of mapreduce along with its trade offs helps creating appropriate solutions and clarifies why traditional approaches, which may seem more intuitive, do not fit well in big data context.

Now that Hadoop, as the basic layer of Hive, and HiveQL are covered sufficiently, we can look Hive from a higher-level view. Next chapter introduces Hive architecture.



Employee

Name	Age	Dept_Id
Alex	26	2
Ben	24	2
Sara	34	5

Department

Dept_Id	Name
5	Mkt
2	Eng
3	Sales

(Dept_Id, {table name, Name, Age})
(2, {Employee, Alex, 26})
(2, {Employee, Ben, 24})
(5, {Employee, Sara, 34})

(Dept_Id, {table name, Name})
(5, {Department, Mkt})
(2, {Department, Eng})
(3, {Department, Sales})

Partition 1
(2, {Employee, Alex,26})
(2, {Department, Eng})
(2, {Employee, Ben, 24})

Patition2
(3, {Department, Sales})

Partition 3
(5, {Employee, Sara,34})
(5, {Department, Mkt})

(2, {Employee, Alex, 26})
(2, {Employee, Ben, 24})
(2, {Department, Eng}
reducer1

(3, {Department, Sales})
reducer2

(5, {Employee, Sara, 5})
(5, {Department, Mkt})
reducer3

Alex, 26,Eng
Ben, 24,Eng
Sara, 34, Mkt

Figure 4 Reduce-side join example

Chapter 3

Hive Architecture

In order to make changes to any system, the foundation and the interactions among its components have to be recognized. In Chapter 2, we studied Hadoop, the underlying layer of Hive. This chapter goes one level higher and describes the Hive architecture and justifies what, where and how changes have to be applied to implement our proposed technique.

3.1 High-level View

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 5.

At the bottom of Figure 5, we can see the Hadoop system. At the top of Figure 5, 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. Because Java works with objects and an RDBMS uses the relational model, an Object Relational Mapping (ORM), called DataNucleus [13], is accompanied by the RDBMS to translate between objects and relational schema.

With respect to meta data availability, meta-store is backed up regularly. On the subject of scalability, the meta-store can become overloaded by the calls from mappers or reducers of a job.

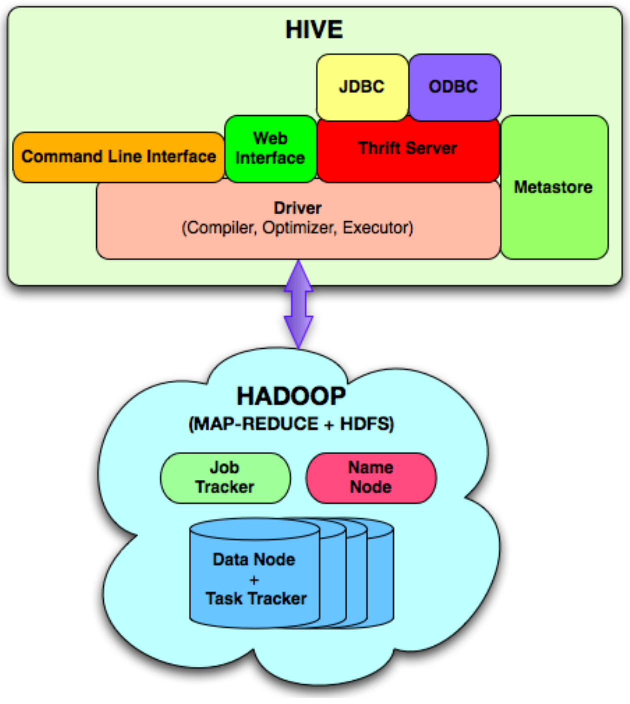


Figure 5 Hive System Architecture [5]

Therefore, the query compiler generates the desired meta data and passes them to mappers and reducers through query plan files so that mappers/reducers do not need to ping meta-store.

For example, in our index-based join implementation we need to know whether a table mentioned in the query is indexed or not. Or we need to know if the index covers all the partitions of a table. Such information is stored in the meta-store and is requested by the query compiler only once, but the vital part of this information is sent to several task trackers.

- 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. Session has the same meaning as it does in traditional databases: "The SQL operations that are performed while a connection is active form a session." [16: chapter8, sessions] In Hive, we start a session as soon as we start Hive and close it by the `quit` or `exit` command. Below is a sample of a Hive session:

```
$ cd $HIVE_HOME $ bin/hive Hive history
file=/tmp/myname/hive_job_log_mahsa_201201271126_1992326118.
txt hive> SHOW TABLES;
OK
orders
lineitem
Time taken: 0.543 seconds
hive> exit; $
```

In Figure 5, the Driver consists of three main components, namely, Compiler, Optimizer, and Executor. The compiler translates HiveQL into a DAG (Directed Acyclic Graph) of mapreduce tasks that are executed by the executor or execute engine in the order of their dependencies. The optimizer resides at some point between the compiler and executor to improve the performance. Our proposed index-based join algorithm interacts with the compiler and optimizer modules, which will be discussed later in more detail.

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. Thrift is a scalable cross-language service development framework; or simply, a binary communication protocol [7]. Clients in different programming languages can communicate seamlessly with Hive using the “thrift interface”. Here, by client we mean any source that issues a query.

- 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. These Application Programming Interfaces (API) provide 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. Figure 6 shows the steps of this process.

Since our work is focused on the Driver component, we further elaborate on this in the following section.

3.2 Query Processor

The query processor is a group of components that transforms DML and DDL commands to a sequence of executable database operations. The components in a conventional RDBMS query processor and the ones in Hive along with the operations that the query goes through in query processor are almost the same. Next section reviews the Hive query processor elements.

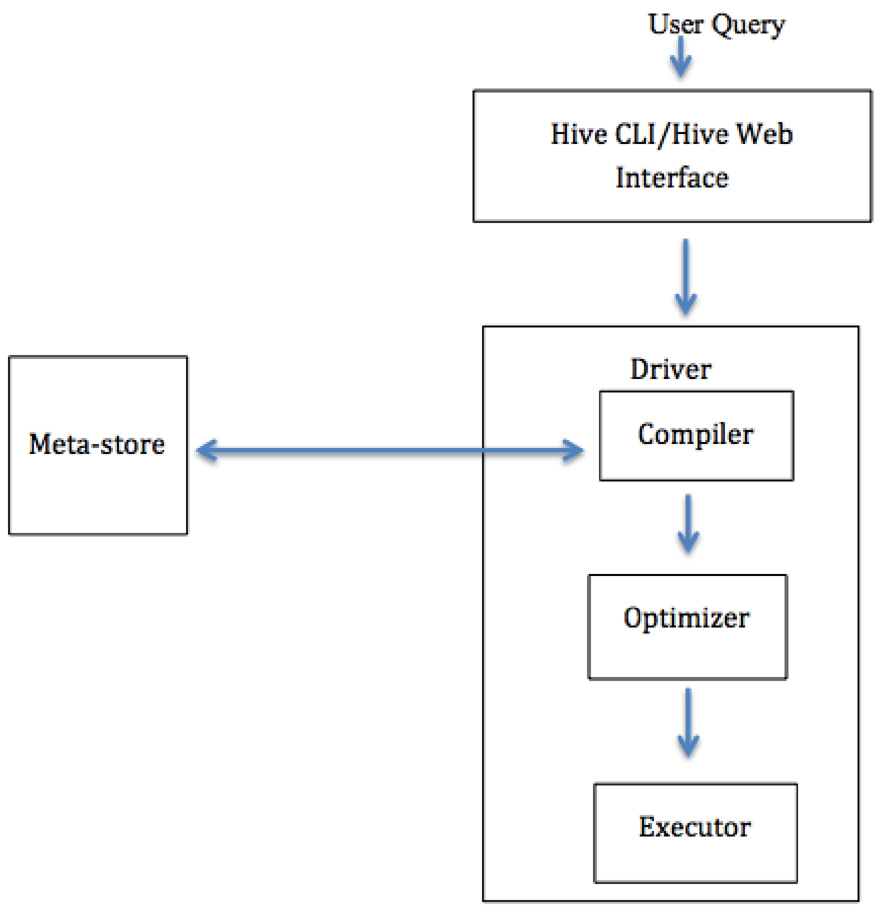


Figure 6 Hive components interaction

3.2.1 Compiler

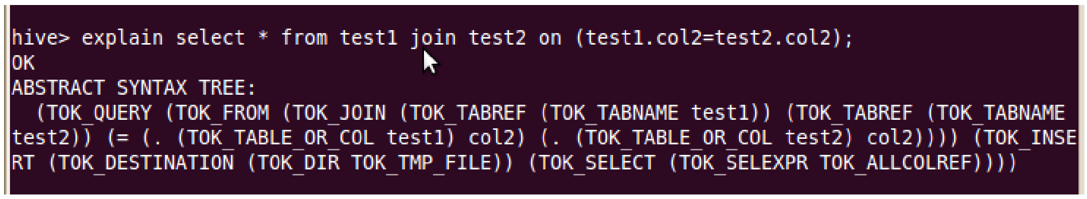
Needless to say, query compiler translates the query into an internal form that uses relational algebra notation and file system operations to be applied on the data. Hive compiler includes these steps: parsing, type checking, semantic analysis, plan generation, and task generation.

3.2.2 Parsing

A SQL query parser builds a tree structure out of the textual form of the query. Similarly, Hive parser examines the user query for syntactic errors. The parser used in Hive is an automatic LL parser, called ANTLR [4], which generates the Abstract Syntax Tree (AST).

The main difference between a parse tree and an AST is that the former looks very much like the original query except that it is in a tree form, whereas AST is more abstract, in which tokens such as braces, parentheses, etc., do not exist in the output.

A user can print the non-tree format of AST for any query using the `EXPLAIN` command.



```
hive> explain select * from test1 join test2 on (test1.col2=test2.col2);
OK
ABSTRACT SYNTAX TREE:
  (TOK_QUERY (TOK_FROM (TOK_JOIN (TOK_TABREF (TOK_TABNAME test1)) (TOK_TABREF (TOK_TABNAME
test2)) (= (. (TOK_TABLE_OR_COL test1) col2) (. (TOK_TABLE_OR_COL test2) col2)))) (TOK_INSE
RT (TOK_DESTINATION (TOK_DIR TOK_TMP_FILE)) (TOK_SELECT (TOK_SELEXPR TOK_ALLCOLREF))))
```

Figure 7 Hive EXPLAIN command

Figure 7 shows the AST of a sample query printed on the Hive Command Line. Figure 8 represents the same query in a tree structure.

AST consists of Hive tokens and literals (column names, table names, etc.). AST is quite useful for understanding what Hive does to a user's query.

The TOK_QUERY is the label of the root used in all ASTs generated for all queries, and represents the query. (TOK_INSERT (TOK_DESTINATION (TOK_DIR TOK_TMP_FILE)) is shown up in the AST whenever there is an output to be displayed on the screen; though the output is printed on the screen, Hive first writes the output of a query to a temporary file. This is why this instruction is present in the AST.

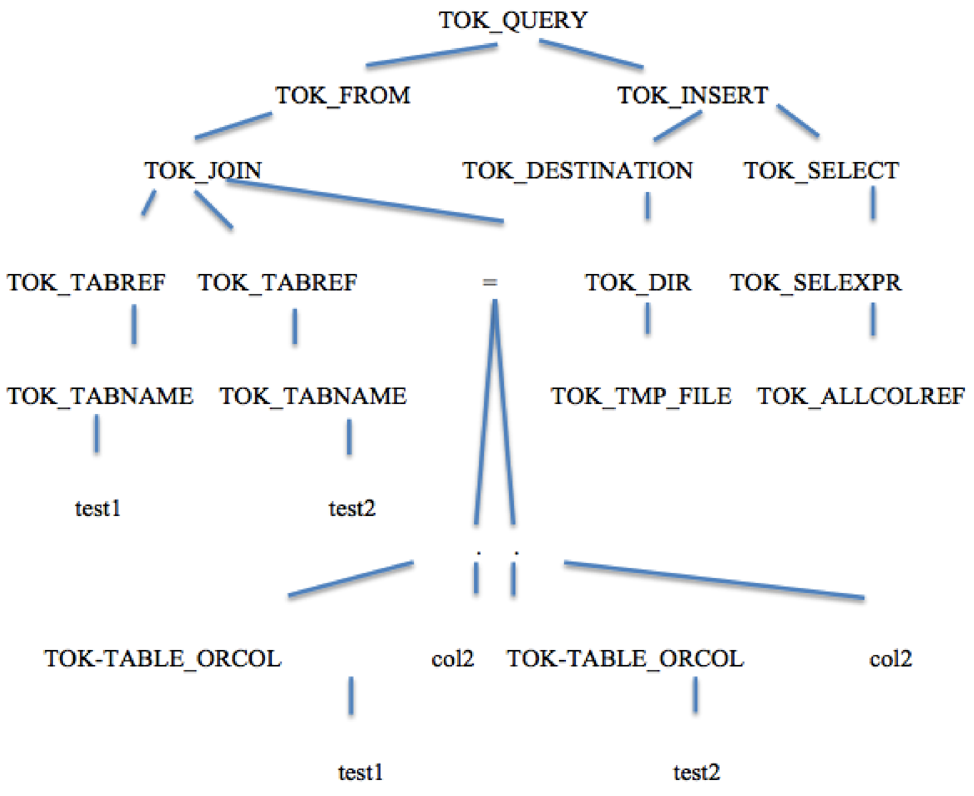


Figure 8 Abstract Syntax tree

Since AST becomes an input for the Semantic Analyzer and the query is transformed from one internal representation to another in the subsequent phases, changing AST requires changing all other data structures that will be used to maintain the query representation in next steps. What is more, the AST only holds the query text and divides the query elements to Hive tokens and literals. The constituents of the AST do not carry other meaningful information and the AST is not changed unless there is a change in the query syntax. Accordingly, our index-based join implementation leaves the AST as is.

3.2.3 Type Checking and Semantic Analysis

Checking type compatibilities and flagging out any semantic error is the core responsibility of the Semantic Analyzer, e.g., verification of the column or table names, performing *, type checking, implicit type conversion, collecting information about sampling (if the table under consideration is sampled) and partitions (if the table under consideration is partitioned; such information could be used for pruning the partitions).

3.2.4 Logical Query Plan Generation

A conventional RDBMS performs such semantic checks via the "query pre-processor." It also does some transformations to turn the parse tree to query plan tree whose nodes are relational algebra operators. In Hive, this transition happens more smoothly by using another internal representation of the query in between the AST and the operator tree.

Hive Semantic Analyzer transforms the AST to an internal representation of the query, called Query Block (QB), which is still block based and not an operator tree. The compiler converts nested queries into parent-child relationships in a QB tree and relevant parts of the AST are reorganized. This makes it easier to transform the AST to Directed Acyclic Graph (DAG) of operators.

The operators consist of (1) relational algebra operators and (2) Hive specific operators. Table 2 lists all operators followed by functionalities of some hive-specific operators.

3.2.5 Query Optimization

The DAG of operators is passed to the optimizer to choose the best possible sequence of operations on the actual data in the query plan. Most RDBMSs today benefit from a cost-based query optimizer. Hive offers a simple yet rule-based optimizer in which the operator tree is recursively traversed and broken up into a series of mapreduce serializable tasks, each encapsulating a part of the query plan, suitable to be executed on HDFS. The plan also carries the required samples/partitions if specified as such by the query itself. Hive optimization includes a chain of transformations in which the operator DAG output of one transformation step is fed as an input to the next. The starting point to change the optimizer or add new optimization algorithm is the Transform interface. To do so, one should implement the Transform interface using their custom logic to add it to the chain of optimizations in Hive Optimizer. Hive optimizer does nothing but invoking all the transformation, one after another, to alter the query plan.

Figure 9 is a generic optimizer showing the optimization steps along with its components. In the source code, each element is an interface (an abstract type) so that each optimization can use its own implementation. The roles of these modules are as follows:

1. Node: This interface implements the nodes in the operator DAG. In the Hive architecture, a Node could be an AST node, an Operator (Table 2), a Task (Table 3), or an ExprNodeDesc (Table 4). ASTs are the tokens generated during the parsing phase mentioned earlier. Tasks are serializable HDFS tasks that will be

discussed in Section 3.2.7.

ExprNodeDesc is the node used to represent compartments of an expression in a query such as columns, constant values, null values, fields of a struct and a generic function.

2. GraphWalker: This interface facilitates traversing the operator DAG. In Figure 9 we see the graph walker fetching each Node for visiting and keeps track of the ones already visited.

3. Dispatcher: This component is basically in charge of rule matching and, in case a certain rule is matched with a Node, it calls the corresponding processor. In Figure 9, "rule = dispatcher.getMatchingRule()" checks whether there is a rule matched for the node being visited. If it returns null, it means either there is no rule associated with the current Node or the condition in the rule is not satisfied.

4. Rule: "Rule.java" is an interface implemented by a single class to specify a rule using the aid of regular expressions notation. Since the elements in the DAG are operators, the basic tokens used in such regular expressions are also of the same type. For example, the rule "TS.*RS" denotes TableScanOperator followed by any operator for any number of times, followed by "ReduceSinkOperator." Our index-based optimization applies one rule specified by "TS%" that simply tries to find a match for "TableScanOperator." The reason is that "TableScanOperator" points to the base table, which is the primary entity that has to be located in the query.

5. Processor: This interface describes the computation required for a specific rule, for example, to use index for the query mentioned in the Node section. In a

nutshell, the processor includes the optimization logic.

To illustrate the optimization components with representative examples in Chapter 4 we will consider a few optimizations with index in Section 4.2.1 for accelerating a query with a WHERE clause and in Section 4.2.2 for accelerating a query with a GROUP BY clause.

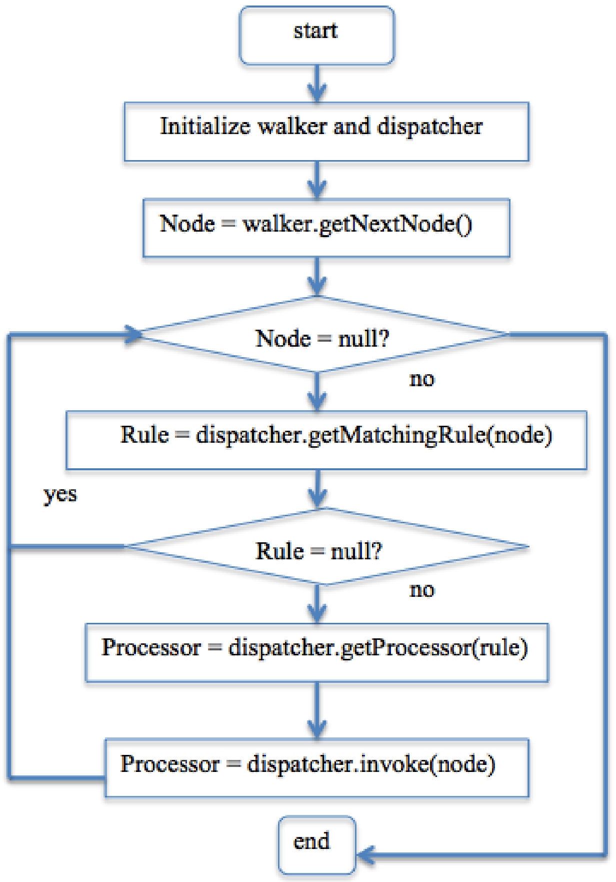

start
Initialize walker and dispatcher
Node = walker.getNextNode()
Node = null?
no
Rule = dispatcher.getMatchingRule(node)
yes
Rule = null?
no
Processor = dispatcher.getProcessor(rule)
Processor = dispatcher.invoke(node)
end

Figure 9 Hive optimization flow [5]

Table 2 consolidates the Hive physical operators that extend the abstract Operator.java class. The functionality of JoinOperator, GroubByOperator, LimitOperator, SelectOperator, and UnionOperator can be intuitively understood. CommonJoinOperator consists of various join implementations including MapJoinOperator. FilterOperator is the implementation of the so-called WHERE clause. UDFOperator implements User Defined

Functions to be applied on the table columns. Hive specific operators like Terminal Operator contains implementations for operators such as ReduceSinkOperator, which is practically the reduce operator. Other operators in Hive are more or less used internally.

Hive physical operator
Collect operator
CommonJoin operator
ExtarctOperator
FilterOperator
ForwardOperator
GroubByOperator
HashTableDummyOperator
LateralViewForwardOperator
LateralViewJoinOperator
LimitOperator
ScriptOperator
Terminal Operator
SelectOperator
TableScanOperator
UDFOperator
UnionOperator

Table 2 Hive Operators

Table 3 shows the serializable Hive tasks or jobs directly executed on HDFS. In Table 3 CopyTask and MoveTask are file system tasks. BlockedMergeTask is used in merging RCFiles. DDLTask contains all the DDL commands including create, alter, drop, alter, add and rename of a table/partition/database/view and more.

ExplainTask implements the `EXPLAIN` facility. FunctionTask is used for creating various functions including User-Defined Functions, Generic User-Defined Functions, and User-Defined Aggregate Functions. MapRedLocalTask and ConditionalTask are used in mapjoins discussed in Section 4.3.1.2.

In Table 4, the expression nodes from top to bottom represent a column, a constant value, a field of a struct, a function defined on the column values (such as an aggregate function), and a null value, respectively.

These nodes represent the `SELECT` list of columns or the conditions mentioned in the `WHERE` clause.

ExprNodeColumnDesc
ExprNodeConstanctDesc
ExprNodeFieldDesc
ExprNodeGenericFuncDesc
ExprNodeNodeNullDesc

Table 3 Hive expression nodes

At the end of the optimization phase, the DAG of operators is converted to a DAG of tasks, each of which is either a map/reduce task or an HDFS task encapsulating a part of the query plan.

BlockMergeTask
ConditionalTask
CopyTask
DDLTask
ExecDriver
ExplainTask
FunctionTask
IndexMetadataChangeTask
MapRedLocalTask
MoveTask
StatsTask

Table 4 Hive serialization tasks

3.2.6 Physical Query Plan Generation

The logical query plan is split into several map/reduce and HDFS tasks. At the end of this stage the physical plan looks like a DAG of tasks with each task encapsulating a part of the plan.

3.2.7 Query Execution

When all the prerequisites of a task have been executed, it can be executed. A mapreduce task first serializes its part of the query plan into the query plan file called plan.xml. This file is added to the HDFS job cache and instances of ExecMapper and ExecReducers are spawned using Hadoop. ExecMapper and ExecReducer are classes which de-serialize plan.xml and then execute the relevant part of the operator DAG. The output is written to a temporary file. If the query is DML, this temporary file is then moved to the desired location [5].

In order to clarify the steps a query goes through in Hive and the Hive architecture modules, we next present the lifecycle of a sample query.

3.2.8 A Sample Query Life Cycle

The query under investigation is a variation of the standard SQL syntax. Hive offers a multi-table `INSERT` in which data is scanned only once (this is why the `FROM` clause below comes first) yet the result can be split into different tables.

```
FROM (SELECT a.status, b.school, b.gender FROM status_updates a
JOIN profiles b
ON (a.userid = b.userid AND a.ds='2009-03-20' )) subq1

INSERT OVERWRITE TABLE gender_summary PARTITION(ds='2009-03-20')
SELECT subq1.gender, COUNT(1) GROUP BY subq1.gender

INSERT OVERWRITE TABLE school_summary PARTITION(ds='2009-03-20')
SELECT subq1.school, COUNT(1) GROUP BY subq1.school
```

The query plan is exhibited in Figure 10. There are 3 map/reduce jobs generated to execute the query. Nodes are physical operators and the arrows represent the flow. The last line in each node shows the output schema of the operator.

In the first map job on the left first the table `status_updates` with the columns `userid`, `status` and `ds` (the partition) becomes available for reading in the TableScan Operator. Through the FilterOperator the data in the desired partition is fetched which is then sent to the reducer by ReduceSinkOperator. The partition `cols` mentioned in the ReduceSinkOperator specifies the column on which the map outputs are classified. Note that this is different from the logical partitions defined over the table. The partition column here is the join key. Map outputs are split into classes in a way that each class holds the key-value pairs having the same key. The same processing happens for the table profiles except for the filtering.

The JoinOperator receives data from both tables and performs the join using the predicate col[0.0] = col[1.0]. Practically it ensures that the tuples come from different sources. Tables are tagged by 0 and 1 before they are passed to the JoinOperator. The component SelectOperator fetches `status`, `school` and `gender` fields accordingly from all the columns. The component GroupByOperator on the left hashes the output of the join on `school` (key: school-value: 1) and the one on the right does the same thing on `gender` field. Afterwards the FileSinkOperator writes these into two temporary files tmp1 and tmp2.

The TableScanOperator in Map2 and Map3 reads the entire sequence of data from the file. The ReduceSink sends the pairs with the same `school` to the reducers on the left GroupByOperator and the pairs with the same `gender` value to the right one. The GroupByOperator collects the pairs with the same key in each group and sends them to SelectOperator which fetches the field and the number of tuples for each value of that filed. Finally the component

FileSinkOperator writes the outputs to the tables `school_summary` and `gender_summary`.

Realizing the different modules and their interactions in Hive is the fundamental step in building insightful solutions that will not only break the architecture but can also fit well. Moreover, such solution can be easily analyzed to recognize potential improvements. We achieved a clear picture of Hive architecture out of the few established publications and reverse engineer of Hive code. Our proposed index-based join relies on this architecture and complies with its conceptual standards, as we will see in the next chapter.

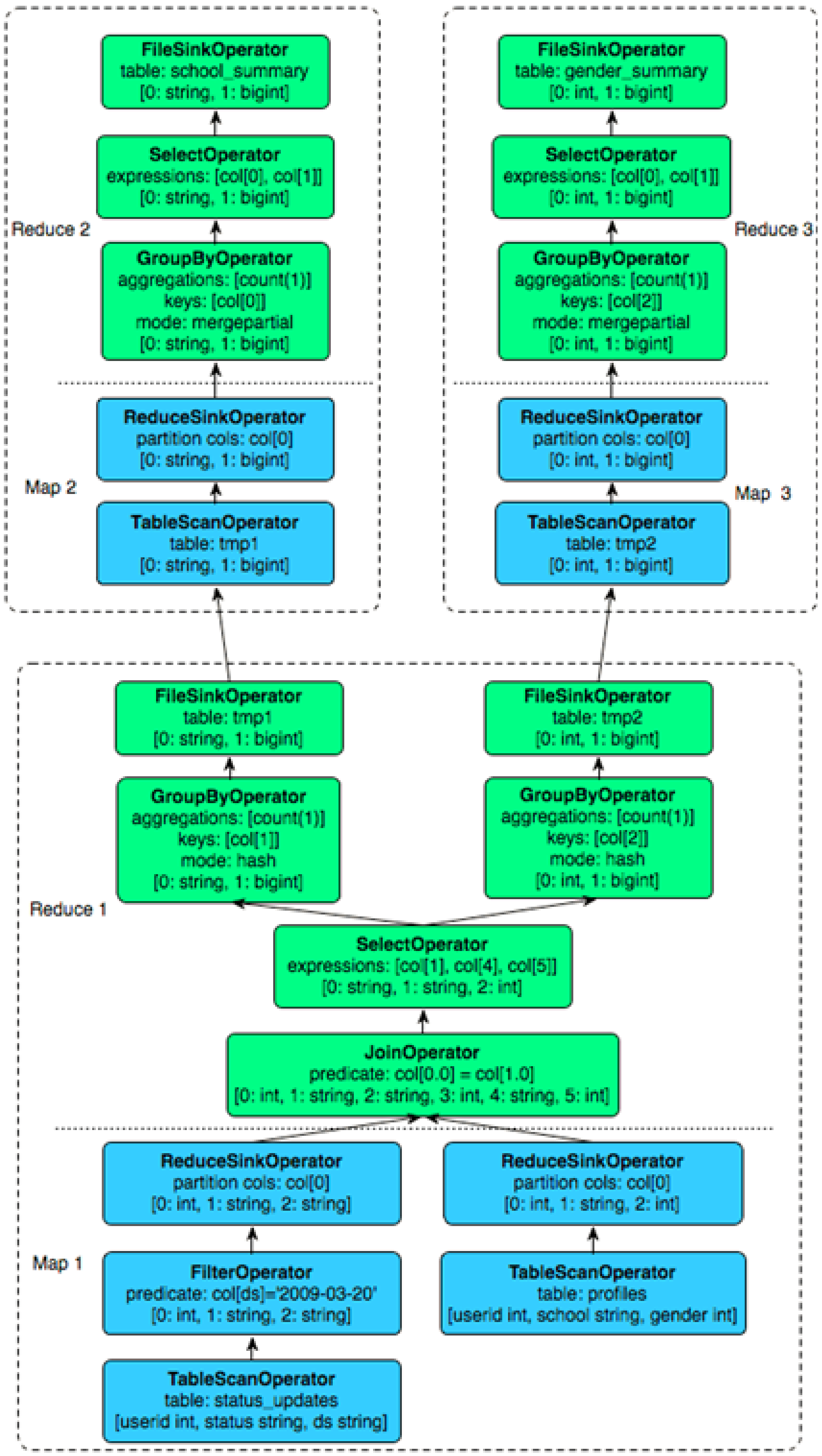


Figure 10 Query plan for a multi-table insert transaction [5]

Chapter 4

Hive Index-based Join

4.1 Hive Optimization

Query optimization aims at selecting the best plan for executing the query. In relational databases, an SQL query is translated into relational algebra followed by compile-time and run-time optimizations before it is executed.

Hive query optimization is a part of its query processing module, which instead of relational algebra uses mapreduce implementation of relational operators. It is worth mentioning that Hive only supports compile-time optimization for Hive. Even though some optimizations like the index-based operations seem intuitively run-time optimizations, they are indeed compile time techniques.

An operator tree is broken into several tasks to be executed on mapreduce. A physical optimization takes a task and modifies and/or optimizes it. Physical optimizations in Hive implement the interface PhysicalPlanResolver.java and non-physical optimizations implement the interface Transform.java. The optimization flow in both interfaces is the same, as elaborated with examples in Section 4.2.

Before we get into technical details of the optimizations, we introduce some of the Hive optimizations using the operator tree transformation notion as follows:

- Column pruning: Only the required attributes are projected out of a row. Required columns are mentioned in various clauses of a query, e.g., `SELECT`, `WHERE`, `GROUP BY`, etc.

- Predicate pushdown: pushes a predicate down to the TableScanOperator so the rows get pre-filtered.
- Partition pruning: the same as column pruning, but for partitions.
- Join re-ordering: Keeps the smaller table in memory and streams out the larger one.
- Handling skews in `GROUP BY`: randomly distributes the data, performs partial aggregations and then re-distributes them based on the `GROUP BY` columns.
- Hash-based partial aggregations: performs local hash-based `GROUP BY` in the mapper. Reducer has to only merge these partial aggregations.

The central idea in the implementation of the above optimization is to reduce read/write or data transfer.

4.2 Related Work

Hive project uses JIRA [22] as its issue-tracking software and issues are addresses in the form of “HIVE-issued code.” There are two main features regarding index-based optimization which are described first. We will then review other works related to Hive.

4.2.1 Accelerating a query with a WHERE clause with index

HIVE-1644 [23] is the implementation of a query containing a `WHERE` clause that leverages the index to fetch the tuples. The main 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.

As all previously discussed optimizations implement the Transform interface, physical optimizers implement the PhysicalPlanResolver interface. For HIVE-1644, IndexWhereResolver.java implements PhysicalOptimizer.java as the start point. Afterwards, the appropriate dispatcher, IndexWhereTaskDispatcher.java, is passed to the graph walker to traverse the tree. The dispatcher looks for the rule "TS%," but only those TableScanOperators that point to tables being indexed. This is simply done by querying the meta-store for the indexes on all the TableScanOperators in the query plan. Upon a positive response from the meta-store the corresponding processor IndexWhereProcessor is called. It worth is mentioning that the TableScanOperator used for obtaining the indexes, is also sufficient to recognize the `WHERE` clause. In the operator tree, `WHERE` clause, expressed by ExprNodeDesc node, is a child of the TableScanOperator, which is represented by Operator node. The "IndexWhereProcessor" is used to extract the predicate out of TableScanOperator, check the coverage of the indexes over the partitions (if any), check if the table size is greater than 5GB (configurable), and if all these conditions hold, it calls CompactIndexHandler because the query can benefit from the compact index type. CompactIndexHandler first decomposes the predicate to the parts that can be processed by the index and the part that cannot benefit form the index. It then re-writes a query upon the index table using the relevant parts of the predicate. The re-written query is compiled and the produced root tasks replace the main query root tasks. As an example, let us consider the query below:

```
INSERT INTO intermediate_file_name
SELECT name
FROM students
WHERE age > 22 AND major = 'CS';
```

If a compact index is already built on attribute `major` and the optimization is set to true, this query is internally re-written to:

```
SELECT _BUCKETNAME, _OFFSETS
FROM students_index
WHERE age > 22 AND major = 'CS';
```

As this is not the original query, the intermediate results should be kept somewhere. The `INSERT` part takes this responsibility. `_bucketname` and `_offsets` are columns of the index table and are used to fetch the corresponding values from the `SELECT` clause on table `students`.

This optimization should be enabled by the user through setting the configuration variable `HIVE.OPTIMIZE.INDEX.FILTER` to true.

HIVE-1644 only allows the part of the `WHERE` clause that contains pure conjunctions over binary expressions, i.e., comparing a column reference with a constant value. Another limitation is that all columns must refer to the same table (no joins or sub-query). If the suitable part of the predicate contains a single condition, such as the query above, Hive searches the index using a binary search.

HIVE-1644 is not applicable for joins. The basic idea it uses is to look for a single constant value or a set of such values all at the same time. The join is supposed to compare all relevant partitions of both tables (in case we have two tables) rather than comparing the data with a single constant value.

In Hive 0.8.0 there are four physical optimizations in addition to HIVE-1644 including: mapjoin and automatic conversion of the common join to mapjoin, skew join, and meta data only optimization which optimizes queries that reference only partition columns in the `WHERE` clause. More precisely, it decides which TableScanOperator points to only partition columns and uses metadata to execute the query. An example of a query that can benefit from metadata optimization is as follows:

```
CREATE TABLE employee (empNo int, empName string)
PARTITIONED BY (deptNo);
SELECT COUNT (DISTINCT deptNo) FROM employee;
```

4.2.2 Accelerating 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. Here is an example of a query and its rewritten form using HIVE-1694.

```
SELECT COUNT (KEY)
FROM TABLE
GROUP BY KEY

SELECT SUM (_COUNT_OF_KEY)
FROM IDX_TABLE
GROUP BY KEY
```

Though this optimization 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:

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

Clearly, the optimization is applied on the operator DAG. Unlike HIVE-1644 that adds up to the operator tree, HIVE-1694 only manipulates the elements inside the operator tree in the sense that no additional minor query and its produced task tree is plugged into the current operator tree. Instead, all the data structures containing the base table are modified so that they point to the index table. This apparent inconsistency between the DAG operator and other data structures created before the operator tree is built, is not causing a problem since there is no dependency on those previous data structures once the query is executed. In addition, the `SELECT` clause and `GROUP BY` clause are re-written completely to fetch the data from the new index table.

HIVE-1694 queries can also have a `WHERE` clause or a sub-query, but again all the six constraints should be satisfied to benefit from the index.

Table 5 exhibits the results of this optimization on a cluster of 2 server class machines each of them having the specifications: CentOS 5.x Linux, 5 SAS disks in RAID5 and16GB RAM and using TPC-H Data[34] lineitem table as benchmark. A query used to test this approach is:

```
SELECT YEAR(L_SHIPDATE), MONTH(L_SHIPDATE) AS MONTH_BKT, COUNT(1)
FROM lineitem
GROUP BY YEAR(L_SHIPDATE), MONTH(L_SHIPDATE);
```

	1M	1G	10G	30G
Index-less	24.161	76.790	506.005	1551.555
Index-based	21.268	27.292	35.502	86.133

Table 1 Query execution times (seconds) for HIVE-1694 [20]

HIVE-1694 imposes a plenty of constraints and works for a limited number of queries; nevertheless, they have added a number of new rewrite assist methods and a skeleton generic rewrite engine that helps being inspired for implementing further optimizations.

Neither HIVE-1644 nor HIVE-1694 aim at accelarating joins; both can only support one single table in the query `FROM` clause.

4.2.3 Using Indexing over mapreduce

Hive consolidates all necessary facilities required 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 [3], 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 pairs form the leaf nodes of the index tree. In the next step, all the leaf nodes are scanned and the intermediate 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.

In this work, given a query, the index is accessed twice to locate the start point and the end point in the leaves. The nodes between these two positions satisfy the query. Map jobs are generated and attached to blocks of data covered between the start point and the end point. Each map first scans the index and then retrieves the records using the offset.

In a conventional B+ tree, since leaf nodes point to each other there is no need to use the index to locate the end nodes. Simply the block in the sequence is scanned until a record with a key bigger than the value of the end point is found.

Hive index structure is slightly different from the one described in [3]; in the sense that the index creation in Hive ends by writing the pairs (index_key, offset) to the index file. This makes the index creation more efficient with respect to time and space, even though there is no formal evaluation on the index creation time or the space requirement in this work. In case there is a query like:

```
SELECT *
FROM table_name
WHERE column_name = column_value;
```

Hive simply performs a binary search on the sorted index keys rather than traversing a tree that is built over the sorted keys.

A predominant alteration this approach brings to Hive index is the random access it uses. Although they have applied paging techniques instead of a less efficient method of reading one record at a time, there is no guarantee that the consecutive offsets in the index drop into the range of the page and finally this number of I/O disks dominates the response time. It is worth mentioning that the cost-model in [3] considers all these factors completely.

Not astonishingly, the proposed method outperforms when the selectivity of the predicate is low. Their best result over a 20GB dataset and a cluster of 8 nodes is more than two times better than the brute-force full scan assuming the selectivity is less than 40% and the I/O volume ratio is less than 80%. This is totally in contrast with the Hive index functioning in which the performance dramatically increases as the selectivity ratio reduces and the data grows.

4.2.4 Query optimization using statistics

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

A recent work proposed storing column-level meta data in Hive tables to benefit from during query execution [19]. Column-level statistics or more specifically, histograms that exhibit value distribution within a table provide more accurate information than just the table size to estimate the output size. A new table is added to Hive meta-store that holds the number of distinct values, number of null values, min and max values and most frequent values as its fields.

In their experiments [19], Gruenheid et al. perform join re-ordering, with consideration of a rough estimation over either the final output size or intermediate tables, as the case study.

There are few weaknesses about their work as follows:

1. Collection of meta data at any levels like partition, table or column imposes extra

overhead in terms of time and space for the database management system though it is not frequently updated.

2. The implemented component is rather a separate component than an elaborately embedded constituent in Hive. Other optimization techniques require more sophisticated implementation that requires additional detailed knowledge over the architecture and dependencies.
3. The time taken to extract the statistics, done by issuing direct queries to Hive, is totally neglected.

This work provides enough functionality and satisfactory results to determine whether or not to use the approach. More importantly, their mapreduce-specific cost formula is very precise. When a computation can not be accomplished in a single mapreduce job, a sequence of several mapreduce tasks have to be carried on and the intermediate results of a reduce is written to the disk to be read for the next map or reduce operation. Their proposed cost formula takes the extra I/O to write to and read from intermediate files into consideration.

Column level stats can provide a rough estimation of the query selectivity to decide between the index-based plan or the regular one. The cost-based query optimization area in Hive has a lot of opportunities to work on.

4.3 Hive Joins

Hive joins syntactically conform to the classic ANSI join, but support only equi-joins. In equi-join, the predicates only check for equality of one or more pairs of attributes from different tables. These equality expressions can be combined by the logical AND.

Join key is the attribute on which two or more tables are being joined in the join predicate. The predicate is recognized by the ON keyword.

The reason for no provision of theta-join in Hive is that it is a difficult operation to be implemented over mapreduce. An algorithm called “1-bucket” theta is recently proposed to process arbitrary joins over mapreduce [31], which uses statistics (input cardinality), though it is not incorporated in Hive yet.

Hive predominantly uses one mapreduce job for each pair of attributes to join, working from left to right. However, if the join key is common between subsequent joins, for each pair of those joins that share the common join key, it uses only one mapreduce job.

In a sequence of joins, Hive buffers tables from left to right except for the last one, which is streamed. Therefore, it is economical to consider the largest table at last. If not, the user can give a hint to the compiler as to which table to stream. For example, in the following query, table “a” will be streamed.

```
SELECT /*+ STREAMTABLE(a) */ a.val, b.val, c.val
FROM a JOIN b ON (a.key = b.key1) JOIN c ON (c.key = b.key1);
```

4.3.1 Implementations of Hive Joins

In practice, a join query can be executed using diverse implementations. Either the user, with prior knowledge about the data, or the Hive compiler, chooses the implementation to use. Different data distributions, tables’ sizes, and tuples’ order in tables can create different use cases where a particular join implementation happens to be more efficient. This section describes Hive join implementations and the use case in which each shines.

A review on the Hive join implementation in the next section is useful to see where our implementation fits in and how it is related to other joins.

4.3.1.1 Common Join

Common join, illustrated in Figure 11, is the basic join implementation on mapreduce framework and works for most of the use cases. In Figure 11, tables X and Y are read through some mappers and the key-values are extracted and passed to the shuffle stage in which they are merged, sorted, and finally sent to some reducers to produce the results. CommonJoin is the default join implementation in Hive.

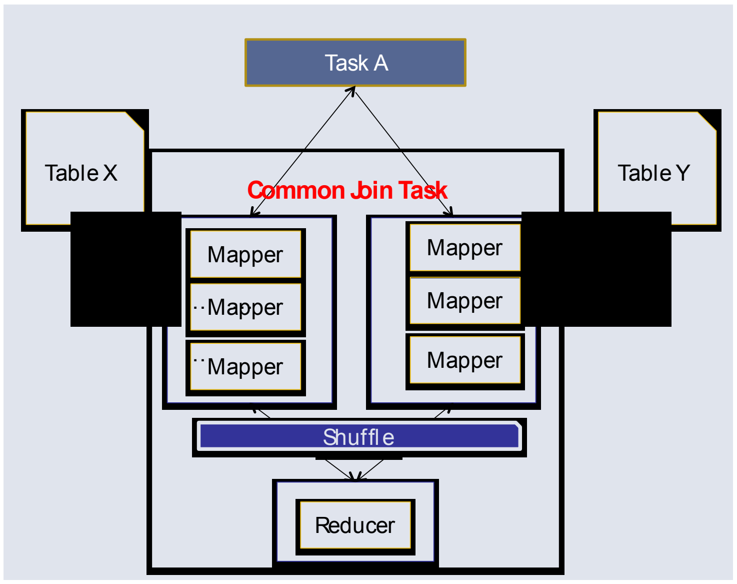


Figure 11 Common Join [24]

4.3.1.2 Map Join

The shuffle stage in Figure 11 is a quite expensive phase and can be disregarded when one of the tables is small enough to fit into the mapper memory. As it can be seen in Figure 12, map join is a map-only job in which the small table is copied into all mappers and some portions of the big table is loaded into the corresponding mappers. Being accessed from numerous mappers, the small table turns out to be a bottleneck for map joins. To overcome this, the small table is converted to a hash table locally, compressed and archived in the Hadoop distributed local cache. The hash table can be duplicated as many times as needed by changing the replication parameter to ensure all mappers can read the small table data from their local disks.

As mentioned, map join can be accomplished if a user gives a hint to the compiler poiting out table the small one. If hint was not given, Hive tries to convert the common join into the map join automatically. In this case, a conditional task is performed between Tasks A and C in Figure 12. In the compile time, there is no precise information on the size of the result, however all possible solutions to join the tables can be generated. At run-time, the conditional task performs the optimal execution based on the size of the result table using the already generated solutions at the compile time.

It is not always the table size that leads to a common join execution. If there is plenty of join keys, we possibly run out of memory, in which case, the task is automatically aborted to perform the original common join.

This strategy, though showed to be the fastest in [24], does not yield the best performance for all cases. Even though the hash table is compressed and archived, large hash tables can become potential bottleneck. Moreover, until the hash table is not copied to a mapper, it cannot start processing.

4.3.1.3 Bucket Map Join

In data warehousing framework, there are arguably large tables for which map join cannot be a suitable solution. In Section 2.2.4, we introduced the notion of buckets. A bucket is a fairly smaller piece of data compared to tables and table partitions and is a candidate for map joins.

Hive requires the user to manually enable bucket map join using the following instruction:

```
        SET HIVE.OPTIMIZE.BUCKETMAPJOIN = TRUE;
```

This is because bucketing itself is not done automatically and the user has to ensure a table is bucketized and the buckets are properly populated.

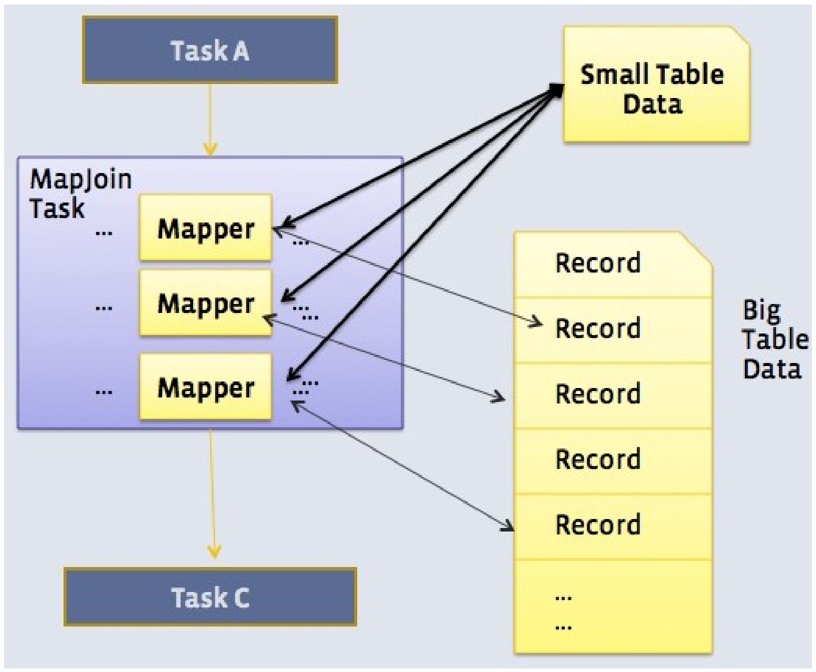


Figure 12 Map Join [24]

There are certain conditions to be satisfied before performing bucket map joins. First, tables must be bucketized on the join keys. This is because the whole idea of bucket map join is to avoid reading the entire tables, but only the relevant buckets. Second, the number of buckets of a table must be a multiple of that of the other table. Note that every table involved in the join must be bucketed. Figure 13 illustrates a bucket map join process.

4.3.1.4 Merge sort bucket map join

Bucket map join limits the size of a bucket to the available main memory capacity. The number of mappers cannot exceed the practical number of mappers per job determined for each cluster.

If the tables have the same number of buckets and they are sorted, a sequential scan of the tables is sufficient to accomplish the join. This is the idea in the merge sort bucket map join that outperforms bucket map join by avoiding the costly shuffle and reduce tasks. The only requirement here is that the join key columns, sorting columns, and bucketing column must all be the same.

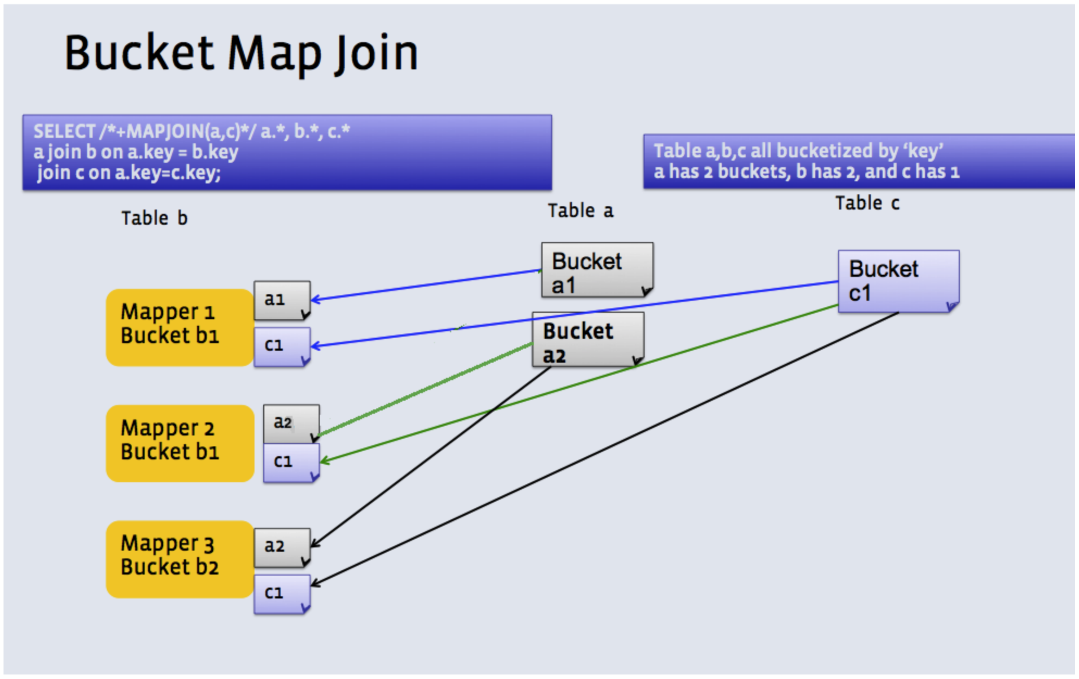


Figure 13 Bucket Map Join [24]

Like Bucket map join, this technique is not triggered automatically. The following commands are used to trigger it:

```
SET HIVE.INPUT.FORMAT =
ORG.APACHE.HADOOP.HIVE.QL.IO.BUCKETIZEDHIVEINPUTFORMAT;
SET HIVE.OPTIMIZE.BUCKETMAPJOIN=TRUE;
SET HIVE.OPTIMIZE.BUCKETMAPJOIN.SORTEDMERGE=TRUE;
```

A use case of this implementation by Facebook is rolling aggregates, which used to be done all at once, but now it is computed incrementally on a daily basis and then merged using the merge sort bucket map join.

4.3.1.5 Skew Join

The first step to do a join is to read the data form tables. This ends up in extracting the key-value pairs form the data in a sorted format (if the data is not already sorted) such that all pairs sharing the same key will be sent to the same reducers. Since the original data may not be distributed uniformly, it is possible that a table is highly skewed, i.e., a specific key corresponds to a large number of values. The skewed key is basically a table column or the join key. When the data is skewed, all other reducers finish quickly except for the one that receives the skewed key, which has become the bottleneck.

The basic idea of skew join is, if a table or a portion of a table fits into the memory, this is b-K1 in Figure 14, we build an in-memory hash table to perform a map join with the table or the portaion of a table that is highly skewed (a-K1 in Figure 14). For non-skewed values, nothing changes. The results of these two phases (the one produced out of the map join and the ones produced without the map join) can be merged to make the final result. Skewed join helps if a small number of skewed keys covers a major percentage of the whole data. Figure 14 illustrates this case.

As it can be seen in the figure, unfortunately tables A and B have to be read twice. The first time they are read to perform the join and the second time they are read partially to do the map-join. Furthermore, the results of the map-join and the ones coming from the reducers have to be read and written twice to merge for producing the final result. A solution is to first read B and build the hash-table, and then read A. For the skewed key, take the hashed-B and execute a map-join. For the non-skewed keys, send them to the

set hive.optimize.skewjoin = true;
set hive.skewjoin.key = Skew_Key_Threshold;
Skew Join
Job 1
Job 2
Final results

Figure 14 Skewed Join [24]

reducer to do a normal join. This strategy reads only B twice that is the non-skewed table.

Skew join has to be prompted manually by the user using the statements:

```
SET HIVE.OPTIMIZE.SKEWJOIN = TRUE;
SET HIVE.SKEWJOIN.KEY = SKEW_KEY_THRESHOLD;
```

A problem here is the Hive user should have prior information about the data distribution and skewed keys in order to take advantage of the approach.

4.4. Proposed Index-based Join

To provide a background for our proposed technique, we begin by defining join in the context of this project with the aid of basic relational algebra. Afterwards we cast this definition onto the query we consider to optimize.

4.4.1 Two-way joins

Given two relations A and B, a two-way join is a dataset obtained by combining tuples `a ∈ A` and `b ∈ B`, such that `A.c = B.d`, where `c` and `d` are column values in A and B respectively on which the join is to be performed. This is called equi-join in database terminology and is denoted by:

$$A \bowtie_{c=d} B$$

The two-way join can be extended in a straightforward way to join more than two relations, called "multi-join:" given n relations $R_1, R_2, ..., R_n$, a multi-way join produces combination of tuples $r_1 \in R_1, r_2 \in R_2, ..., r_n \in R_n$, such that $r_1.a_1 = r_2.a_2 = ... = r_n.a_n$. The join attributes are $a_1, a_2, ..., a_n$ which are column values in $R_1, R_2, ..., R_n$ respectively. This join is denoted by:

$$R_1 \bowtie_{a1=a2} R_2 \bowtie_{a2=a3} ... \bowtie_{an-1=an} R_n$$

4.4.2 Objective

The aim of our work is to speed 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 in the queries we consider. All our changes are internal and the syntax of the query remains intact. Though, our implementation can be easily modified to work for mutiple tables.

4.4.3 Index-based Joins vs. Index Joins

The existing indexes in Hive are built only over single tables. There is an interesting idea of "join indexes" for materialized views once Hive could support such structures [21]. An implication of this definition, we note, would be that an assembly of index built over more than one table could be used at the time of a join. In RDBMS, a join index is a pre-computed access structure that maintains pairs of identifiers of tuples from two or more relations that would match in case of a join. This assembly is used for the tables that are updated infrequently and thus would be a suitable optimization approach in Hive. A sample design for an index join is to keep unique identifiers of the matched tuples in the same structure and cluster them on either of the unique identifiers of one or both tables. Though, one should keep in mind that current implementation of Hive does not support the concept of primary keys [10, chapter 9, Unique Keys and Normalization], which are considered the unique identifiers of tuples in RDBMSs. For further details on index join please refer to [25][36].

4.4.4 Design

This optimization flow conforms to the regular optimization flow we already described. The optimizer receives an operator tree and invokes all the possible or enabled optimizations one by one. Each optimization class implements the Transform interface and transforms the operator tree into an optimized one. In our implementation, "RewriteJoinUsingIndex.java" implements the Transform and it is inside its transform() method that the query is examined carefully to ensure it meets the requirements. Figure 14 is an abstract representation of what happens inside the transform() method. In Figure 14, the query plan is given to the optimizer in the form of a DAG of operators. First, the optimizer searches for a JoinOperator. If this step is omitted the optimization is enabled for any query, which may fail at last. Depending on the different operators, different decisions have to be made. As mentioned earlier in Related work in Section 4.2, a query containing a `WHERE` clause uses a distinguishably different design to benefit from the index from the one a query containing a `GROUP BY` does. Second, the optimizer examines the query for a two-way join. Our technique can be easily extended to support multi-way joins, by leaving this check out, but since we have limitations over the `SELECT` column list, the `SELECT` columns turns out to be a small subset of all provided columns. Depending on the use case (the desired columns to be projected) our implementation can work for a two-way or multi-way join. In the next step we get the operator TableScanOperator which points to the table, it should manipulate. We have to check that the table has an index and the index is valid. The optimizer iterates over the indexes and checks if 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 bitmap index obviously does not fall into this kind of optimization. In practice, normally the join key is a column with considerable number of values, which makes the bitmap index an improper choice. About the partitions, the query is inspected for having any references to partitions. Partitions are those previously known to user, distinct valued and meaningless columns, in the sense that they do not hold real data. Thus the partitions are not referred to in `SELECT` clause. They can appear in the `WHERE` or `GROUP BY` clauses. In our implementation, the optimizer checks whether the `WHERE` clause contains any partitions and returns both the confirmed and the unknown partitions. Confirmed partitions are the ones that satisfy the condition in the `WHERE` clause obviously the unknown partitions are the ones that their usability becomes clear only at run-time. The index validity check returns true if a table is not partitioned, or if it has partitions, 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.

In the Hive architecture, introduced in Chapter 3, we studied the normal flow of the query optimization. We described the “rules” and “rule matching” which are used to invoke the relevant optimization logic, called processor. If the query goes through all the steps in Figure 15 to ensure that the optimization can be applied, what is the role of a rule? In other words, between the rule and the examination process in Figure 15 which one decides to apply the optimization? Figure 15 ensures the query meets all the requirements of the specific optimization. Rules, expressed by regular expressions, are unable or sometimes too complex to decide suitability for the optimization. On the other hand, they are fast in recognizing the nodes. Once the query is proved to be able to benefit from the optimization by going through the steps in Figure 15, the rules are used to point to the target pieces in the operator tree that have to be manipulated. In our case the rule “TS%” seeks for the TabeScanOperator that has indexes.

After the existence and suitability of the index is confirmed, query is re-written to use the index:

```
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 operator DAG and the new table must be added instead. Other data structures such as QB or AST created previously do not match with the new operator DAG. However since there is no dependency on them, this is not of an issue when the optimization process starts. Since the table is changed, the schema is also changed. This requires adjusting the de-serializer.

If any of the conditions is not met in the flow described in Figure 15, the cycle ends in "Exit," which means the execution does not use the index and produces the result as usual without considering indexes.

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. We will elaborate on this point when we present our experiments. Index-based join can be set through a run-time parameter as follows:

```
    SET HIVE.OPTIMIZE.INDEX.JOIN = TRUE;
```

By analyzing the works on Hive query optimization we inspired how to accelerate a query joining two tables using indexes. We first inspect the query representation elements for one join operator, two tables and indexes built over at least one of the tables. Then we replace a table with its effective index. This flow conforms to the normal Hive query optimization flow and causes our optimization to easily integrate with Hive. We will proceed with evaluating our solution performance in the next chapter.

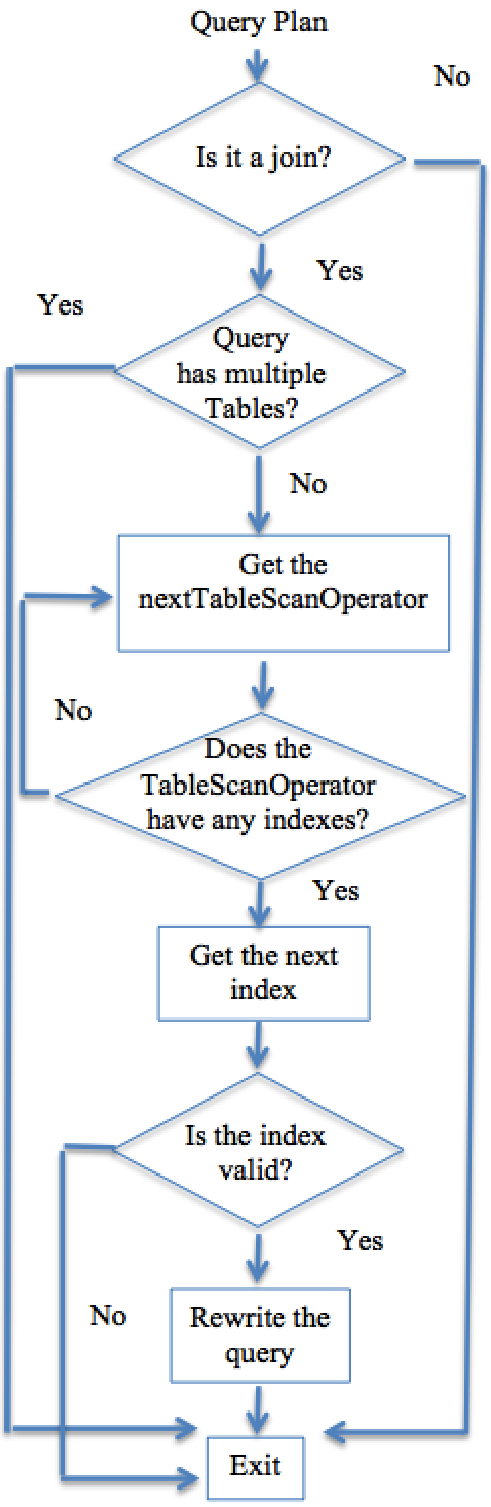


Figure 15 Optimization flow for index-based join

Chapter 5

Experiments and Results

This chapter describes our experiments to evaluate the performance of the proposed index-based optimization in Hive.

The testing 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.

To generate the data used in our experiments, we considered the standard benchmark TPC-H version 2.14.4. Among the eight tables defined in the benchmark, we used `lineitem` and `orders` for containing the largest number of tuples. We created datasets of various sizes ranging from 1GB to 20GB, distributed between these two tables with `lineitem` being relatively 5 times larger than `orders`. Tables 6 and 7 depict the precise data distribution used in Experiment 1in Section 5.5.

Data distribution (GB)	1	5	10	15	20
Size (GB)	0.00015	0.81	1.6	2.5	3.3
No. of tuples	1,500,000	7,500,000	15,000,000	22,500,000	30,000,000

Table 2 Data distribution in `orders` in Experiment 1

Data distribution (GB)	1	5	10	15	20
Size (GB)	0.71	3.6	7.2	10.9	14.6
No. of tuples	6,001,215	29,999,795	59,986,052	89,987,373	119,994,608

Table 3 Data distribution in `lineitem` in Experiment 1

5.1 Test Datasets

During testing, first, each query is executed using the already existing technique. The second time, query is executed using the proposed index approach. Experiments are repeated 5 times for each query and the average time is reported as the response time.

Performance is measured with respect to two different criteria. First, performance is evaluated over different dataset sizes. The volume of data being tested ranges from 1GB to 20 GB and the number of tuples ranging from 7×10^6 to 150×10^6. The queries used are presented in Section 5.2. The results are organized and presented in Tables 8 to 11. The experiments are conducted on both single-node setup and multi-node setup.

The second evaluation criterion is measured performance with regard to query selectivity ratio. The sizes of datasets considered were in the 1 GB to 90 GB range; in each step the data is double that of the previous step (Table 14). Consequently, though the output of the query has a fixed size of 1,500,000 tuples, when the number of tuples is doubled in each step, the selectivity ratio gets doubled too.

5.2 Test Query sets

In order to evaluate the performance, a two-way join is executed with and without the presence of a `WHERE` clause and/or `GROUP BY`. This leads to the following 4 different combinations:

1. `INSERT OVERWRITE TABLE result_q1 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. `INSERT OVERWRITE TABLE result_q1 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. `INSERT OVERWRITE TABLE result_q1 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. `INSERT OVERWRITE TABLE result_q1 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;`

In the query plan, if there is `WHERE` or `GROUP BY` clause, it is the child of the TableScanOperator. In other words, `WHERE` are `GROUP BY` clauses are dependants of the TableScanOperator. Since we locate and manipulate the TableScanOperator in our technique, we considered queries 2-4 in order to ensure our approach does not affect any of the potential dependants of the TableScanOperator. The result of all queries is written to a table to check for consistency with those of Hive approach, and hence correctness of our implementation and also to avoid lengthy display time.

The `DISTINCT` keyword appears in all types of the queries. The reason is that the index file stores the unique values of the join attributes and when the base table is replaced with the index table, only the distinct values are accessible. Elimination of the `DISTINCT` keyword is possible when the query contains a `GROUP BY`, as it eliminates the duplicates at the final stage. `GROUP BY` and `DISTINCT` arrange for the same functionality in our work but in different stages.

Another consideration is that not all the columns are accessible with the developed approach. As soon as an index table is placed in the query plan, access to the previous table would be limited only to the ones stated in the index file.

5.3 Run-time Parameters

Between the two sets of experiments, all parameters have equal values. There are two mapreduce parameters, which are set specifically for these tests. The parameter `mapred.map.tasks` controls the number of map tasks and `mapred.reduce.tasks` holds the number of reduce tasks. As a rule of thumb, number of map tasks is 10 times the number of tasktrackers and reduce tasks are twice the number of tasktrackers. In our experiment, these parameters were set to 20 and 4, respectively.

5.4 Evaluation Metrics

In all of our experiments, we measure performance using the query response time in seconds(s). In Experiment 2, we measure performance by also considering query selectivity since it becomes important in the presence of indexes. [9] took a glance at the number of map/reduce tasks in their experiments; however, this was not considered as an evaluation metrics in their work.

5.5 Experiment 1

Experiment 1 includes execution of the 4 query types on a multi-node and a single-node Hadoop cluster using the data size ranging from 1GB to 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}.

5.5.1 Query1

Query1 is the simplest query to test our approach. It joins two tables on a single join attribute:

```
INSERT OVERWRITE TABLE result_q1
SELECT DISTINCT o.O_ORDERKEY, o.O_TOTALPRICE, o.O_ORDERDATE
FROM orders o JOIN lineitem l
ON o.O_ORDERKEY = l.L_ORDERKEY;
```

We can see the response time (s) in Table 8 for using existing implementation of Hive (no index) and index-based (our approach) for multi-node and single-node setup. The average response times are depicted in Figure 16.

	Index-less approach response time (s)					**Index-based approach response time (s)**				
	1GB	5GB	10GB	15GB	20GB	1GB	5GB	10GB	15GB	20GB
Multi-node setup	73.8	191.58	294.29	436.15	585.85	72.32	147.82	198.56	230.40	342.05
	68.88	187.06	297.13	426.89	519.54	70.89	139.22	211.65	212.16	321.87
	70.76	193.57	299.06	418.89	560.63	66.94	146.03	219.45	219.37	317.09
	69.34	187.02	291.36	387.16	522.61	65.76	141.56	201.14	220.07	320.40
	69.50	186.57	317.04	387.16	574.43	66.77	137.83	208.45	214.07	317.37
	70.46	**189.16**	**299.78**	**411.25**	**552.6**	**68.54**	**142.49**	**207.85**	**219.21**	**323.76**
Single-node setup	54.14	274.61	545.91	807.93	1104.47	36.18	145.50	284.88	338.30	636.92
	55.15	257.49	551.75	811.14	1136.64	36.40	141.31	296.48	344.92	626.15
	55.78	255.15	546.37	804.14	1134.2	35.36	144.87	284.81	334.98	624.23
	55.14	269.85	544.83	825.83	1132.37	36.63	145.64	288.62	345.95	621.92
	54.40	273.78	546.87	806.35	1134.4	36.40	160.45	287.56	341.49	614.19
	54.93	**266.18**	**547.15**	**811.08**	**1128.42**	**36.20**	**147.55**	**288.47**	**341.13**	**624.68**

Table 4 Query1 Response time with/out index on multi-node and single-node setups

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 is, the bigger the gap between the index-less and index-based approaches becomes. The largest response time gap is at 15GB in Figure 16 and our index-based method is almost twice as fast as the index-less approach.

In the single-node setup, we see the same behaviour; for each data size, our proposed method outperforms the normal one and the larger the data is, the bigger the gap between the index-less and index-based approaches becomes. The index-based method is almost always more than two times faster than the index-less approach.

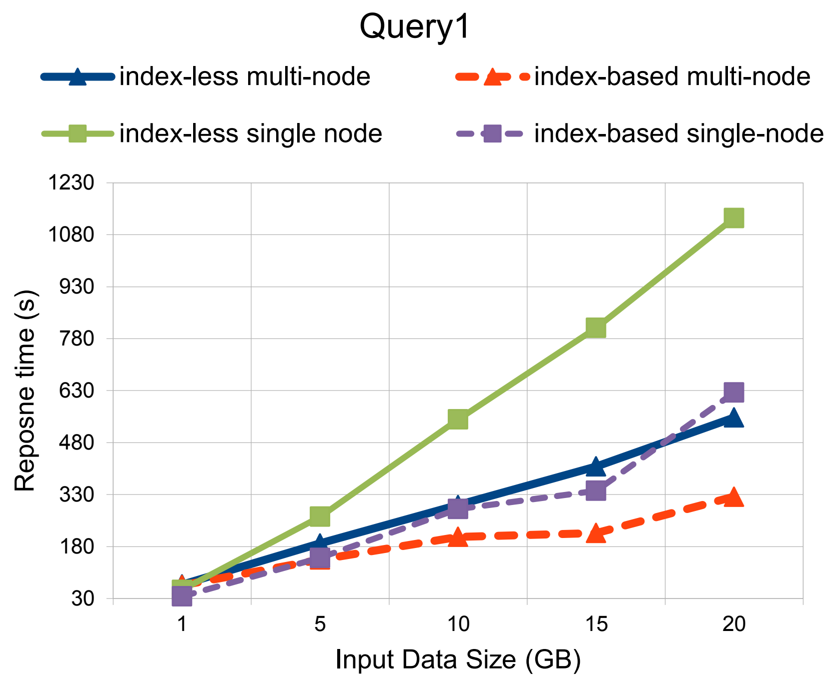


Figure 16 Comparison of executing join Query 1 with/out index on multi-node and single-node setup

Comparing the results from both setups, the single-node setup works faster than the multi-node setup for 1GB in both approaches. For 5GB, the multi-node setup is slightly faster than the single-node one. 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.

As can be seen in Table 8 that repeating the same query over the same dataset does not lead to significantly different response times. If we had run the same query with the same dataset on a traditional RDBMS like MySQL, the first response time would have been the largest one. The reason is, unlike in traditional RDBMSs, 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 (The first response time for 10 GB in Table 8 is the smallest 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 Experiment 1, we conduct the same test with different queries, which are extensions of query 1.

5.5.2 Query2

```
INSERT OVERWRITE TABLE result_q1
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;
```

	Index-less approach response time(s)					Index-based approach response time(s)				
	1GB	5GB	10GB	15GB	20GB	1GB	5GB	10GB	15GB	20GB
Multi-node setup	69.90	187.68	293.26	436.16	517.01	70.10	140.66	202.12	219.83	318.13
	72.81	194.72	285.84	387.01	529.52	68.11	138.92	202.63	218.24	315.70
	70.00	185.14	287.73	392.63	508.36	68.68	139.51	195.59	212.21	311.33
	71.66	195.89	308.49	430.85	562.9	69.63	140.24	212.38	218.02	313.03
	71.81	186.08	290.68	388.88	503.01	67.77	138.82	207.27	219.12	315.99
	71.23	**189.90**	**293.2**	**407.11**	**524.16**	**68.86**	**139.63**	**204.00**	**217.48**	**314.84**
Single-node setup	56.18	268.63	531.2	802.41	1120.21	35.60	150.70	298.55	348.24	636.2
	54.26	272.28	596.26	801.76	1127.75	36.63	148.04	300.51	340.93	631.27
	56.00	269.62	532.24	802.9	1131.64	36.69	152.49	301.00	342.71	646.23
	56.01	273.50	540.28	803.63	1090.02	36.67	148.78	301.04	344.44	605.03
	56.29	271.49	606.83	791.4	1207.17	36.32	148.83	305.17	337.5	632.71
	55.75	**271.10**	**561.36**	**800.42**	**1135.36**	**36.38**	**149.77**	**301.25**	**342.76**	**630.29**

Table 5 Query2 Response time with/out index on multi-node and single-node setups

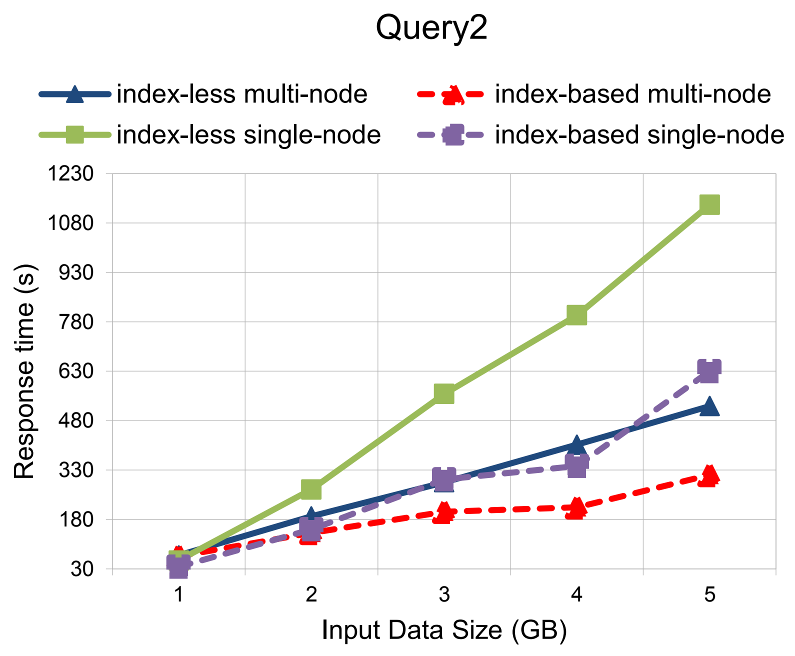


Figure 17 Comparison of executing join Query 2 with/out index on multi-node and single-node setup

5.5.3 Query3

```
INSERT OVERWRITE TABLE result_q1
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;
```

	Index-less approach response time(s)					Index-based approach response time(s)				
	1GB	5GB	10GB	15GB	20GB	1GB	5GB	10GB	15GB	20GB
Multi-node setup	71.64	180.63	286.13	437.8	528.85	69.28	139.67	206.38	212.67	310.01
	70.56	190.65	309	405.8	521.32	68.68	147.87	203.01	215.93	327.87
	70.57	187.71	287.22	430.4	513.65	65.85	142.24	206.15	218.18	312.33
	70.93	184.35	298.02	435.35	527.58	66.83	143.71	209.80	225.87	308.46
	70.49	186.60	316.28	426.2	553.39	68.51	138.44	209.86	221.26	361.97
	70.84	**185.99**	**299.33**	**427.11**	**528.96**	**67.83**	**142.39**	**207.04**	**218.78**	**324.13**
Single-node setup	55.12	270.29	536.28	807.36	1123.52	35.41	145.53	300.55	349.04	627.6
	55.42	273.59	541.68	815.46	1127.75	35.36	147.57	301.17	339.63	611.81
	54.60	270.04	531.66	811.67	1126.56	35.65	146.64	300.39	334.84	656.55
	53.47	268.18	549.4	820.93	1131.92	35.33	145.90	302.84	344.70	661.83
	54.94	269.11	620.72	810.71	1135.86	36.44	152.49	301.26	345.65	643.96
	54.71	**270.24**	**555.95**	**813.23**	**1129.12**	**35.64**	**147.63**	**301.24**	**342.77**	**640.35**

Table 6 Query3 Response time with/out index on multi-node and single-node setups

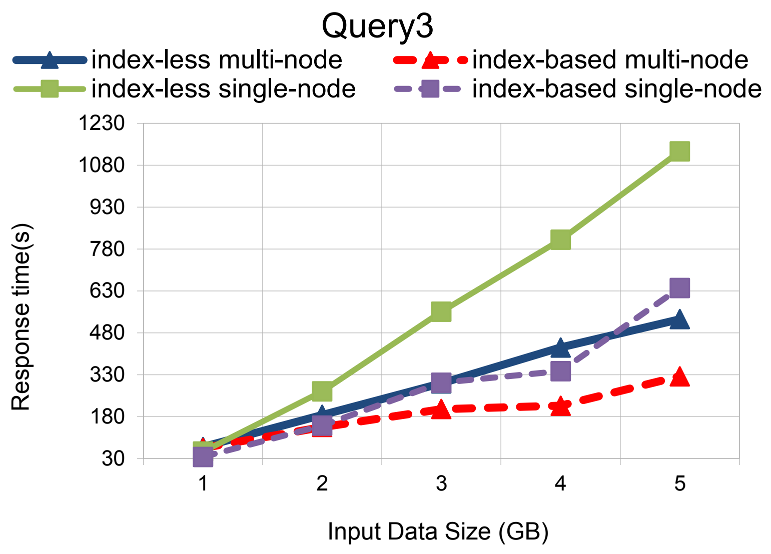


Figure 18 Comparison of executing join Query 3 with/out index on multi-node and single-node setup

5.5.4 Query4

```
INSERT OVERWRITE TABLE result_q1
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;
```

	Index-less approach response time(s)					Index-based approach response time(s)				
	1GB	5GB	10GB	15GB	20GB	1GB	5GB	10GB	15GB	20GB
Multi-node setup	69.53	185.09	287.22	429.12	565.72	68.47	143.50	202.88	222.54	321.12
	72.90	191.07	300.48	402.4	574.45	71.03	141.57	207.91	211.39	307.60
	70.67	187.98	308.98	427.11	495.04	68.86	139.94	193.11	212.01	321.03
	73.84	188.14	300.25	392.3	547.99	69.55	136.82	197.81	248.85	318.07
	70.45	189.37	309.15	405.78	516.28	66.77	143.98	195.86	223.28	317.21
	71.48	**188.33**	**301.22**	**411.34**	**539.90**	68.94	141.16	199.51	223.62	317
Single-node setup	55.12	271.49	541.67	806.68	1089.36	37.55	147.43	302.71	343.55	636.25
	54.54	272.21	538.28	809.75	1127.72	36.29	145.58	300.43	346.32	630.82
	54.15	264.64	576.79	807.53	1122.83	36.30	148.90	283.82	345.89	655.01
	56.61	269.03	523.18	810.46	1135.85	36.44	147.59	290.94	346.32	650.01
	56.44	271.85	545.09	815.59	1184.92	36.27	144.92	296.00	345.01	667.44
	55.37	**269.84**	**545.00**	**810.00**	**1132.14**	36.57	146.88	294.78	345.42	647.90

Table 7 Query4 Response time with/out index multi-node and single-node setups

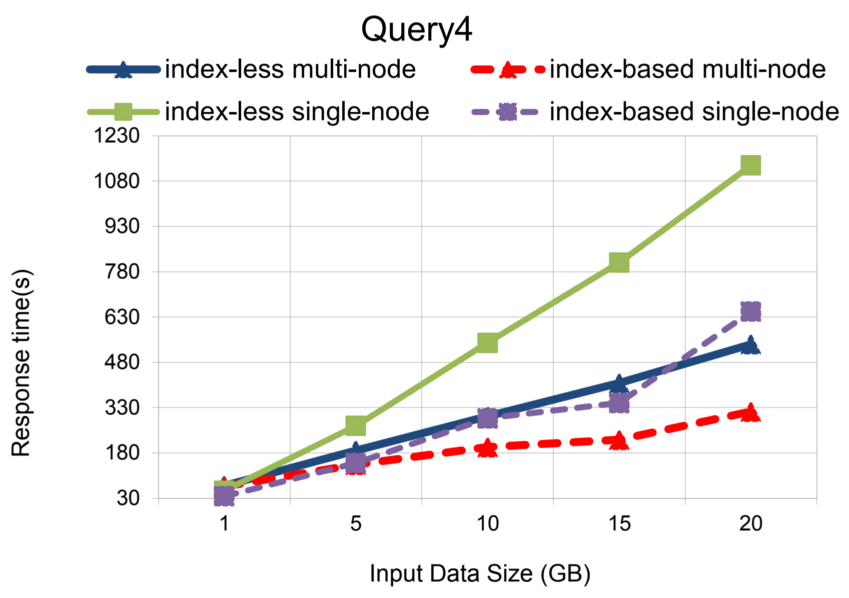


Figure 19 Comparison of executing join Query 4 with/out index on multi-node and single-node setup

Similar to Figure 16, the index-based approach showed in Figures 17 to 19, is faster than the index-less one as the data grows and the overhead to choose the index alternative only pays off when the data is huge enough.

Looking at Figures 16 to 19, the graphs show similar curves, using which we concluded that the 4 types of query have almost the same behaviour and they did not lead to significantly different response times in neither approaches (Figure 20 to 21). 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 2, 1 for the `JOIN` part and 1 for moving the output to table `result_q1`.

As a result, in the rest of the experiments we only use Query 1.

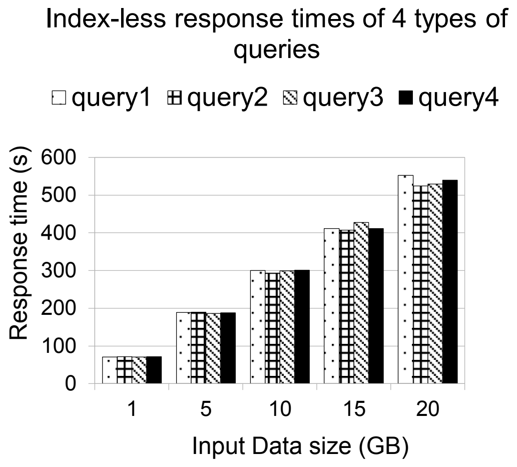


Figure 20 Response times of 4 different queries without index on multi-node setup

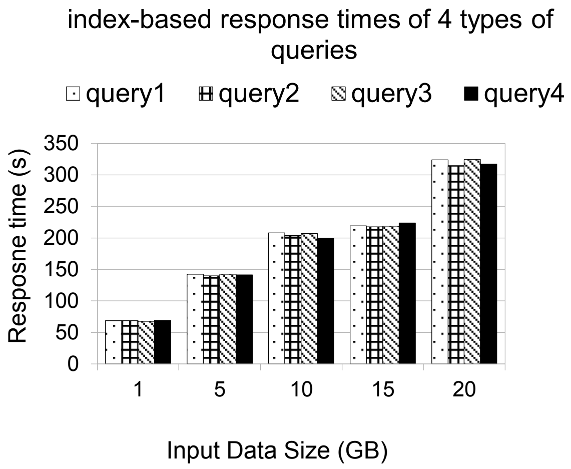


Figure 21 Response times of 4 different queries with index on multi-node setup

Data distribution	1GB	5GB	10GB	15GB	20GB
Size (GB)	0.131	0.69	1.38	2.1	2.82
Time (s)	43.38	109.37	209.35	302.75	411.57

Table 8 Index size and index creation time for Experiment 1

We also studied the cost of index creation in terms of time and space to decide whether or not to use index. Table 12 presents the information on the space and time taken for creating indexes in Experiment 1. Figures 22 and 23 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 multi-node setup respectively, since Figures 20 and 21 show almost constant response times for all the queries and the indexes were all built in the multi-node setup.

As shown in Figure 22, 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.

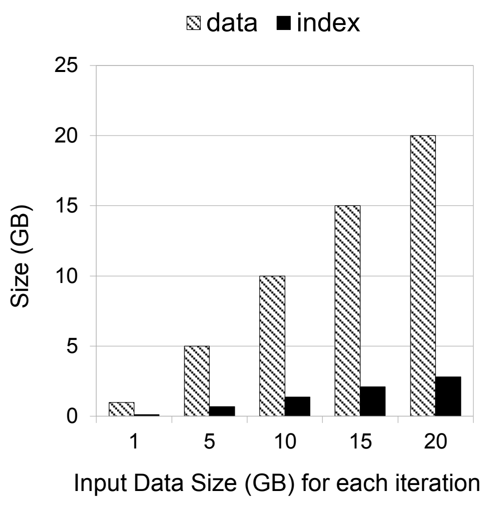


Figure 22 Index size vs. data size

However, since indexes are built manually, 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 Figure 23, 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.

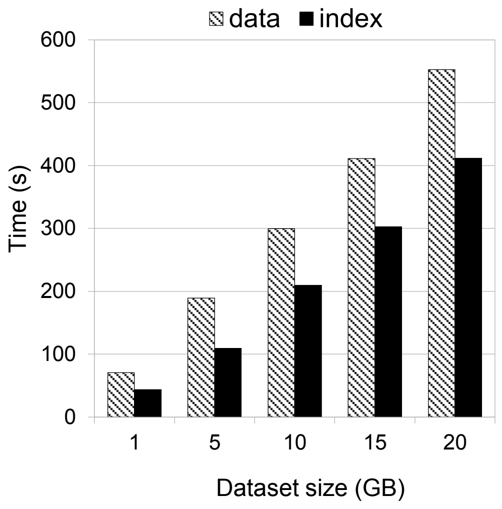


Figure 23 Index creation time vs. query response time

Table 13 looks at the execution of Query 1 from mapreduce perspective. Index-less approach results in dramatically less number of map tasks as the data grows. Also, we can see that the number of map tasks goes beyond 20, which we previously configured in Section 5.3. This means, `mapred.map.tasks` is just a hint to Hadoop and in practice the number of map tasks is determined by the size of the input data.

Another point about mapreduce implementation of joins in Table 13 is that the lengthiest part is the shuffle phase described in Section 4.3.1.1. To avoid this, one can execute mapjoin (see Section 4.3.1.2) instead of the standard join.

	Index-less approach					Index-based approach				
Data Size	1GB	5GB	10GB	15GB	20GB	1GB	5GB	10GB	15GB	20GB
No of maps	4	19	20	55	74	2	7	7	20	27
No of Reduce	4	4	4	4	4	4	4	4	4	4
Average map time (s)	8	9	9	11	10	6	9	10	12	12
Average shuffle time (s)	7	48	52	170	219	7	20	20	68	87
Average reduce time (s)	5	33	32	83	101	3	14	16	49	65

Table 9 Mapreduce metrics for query 1 executed on multi-node setup

5.6 Experiment 2

The second set of experiments we conducted for performance measurement considered different query selectivity ratios. For this we used Query1 over the tables `orders` having a fixed size of 164 MB with 15×10^{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 6×10^{6} to 7×10^{8}. Table

	#1	#2	#3	#4	#5	#6	#7	#8

14 provides details of data distribution in table `lineitem` and our measure of selectivity (number of output tuples/number of input tuples) for Experiment 2. 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.

No of tuples	6,001,215	12,002,430	24,004,860	48,009,720	96,019,440	192,038,880	384,077,760	768,155,520
Size (GB)	0.71	1.4	2.8	5.7	11.3	22.6	45.3	90.6
output/input	0.24	0.12	0.06	0.03	0.01500	0.00781	0.00390	0.00195

The response times are provided in Table 15. Figure 24 shows the graphs for average response times measured.

In Figure 24, as we move from case 1 to 8, the index-less approach grows non-linearly, while the index-based approach remains more or less constant at an average of about 87

Table 10 Data distribution in `lineitem` in Experiment 2

seconds. In case 7, 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 behaviour of the index-less graph in Figure 24, 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 0.0007% query selectivity.

	#1	#2	#3	#4	#5	#6	#7	#8
Index-less approach response time (s)	88.36	96	117.37	164.51	249.56	428.28	824.39	1840.69
	86.77	96.81	118.06	160.7	252.61	424.27	858.29	1918.13
	90.31	101.32	117.16	162.3	251.86	421.78	846.63	1826.77
	84.95	96	113.75	159.63	247.67	423.42	847.55	1901.92
	87.1	96.82	114.04	159.27	250.31	422.94	850.34	1835.74
	87.50	**97.39**	**116.08**	**161.28**	**250.40**	**424.14**	**845.44**	**1864.65**
Index-based approach response time (s)	82.25	75.86	80.06	81.81	96.49	96.12	98.99	146.41
	83.84	79.42	84.95	80.70	89.96	90.88	94.01	115.24
	82.61	79.82	85.16	82.15	91.85	91.92	90.57	121.85
	79.61	79.62	82.73	81.79	87.71	94.24	89.89	121.60
	77.07	79.71	82.08	80.96	85.90	93.89	88.81	120.52
	81.08	**78.89**	**82.99**	**81.48**	**90.38**	**93.41**	**92.45**	**95.84**

Table 11 Query1 Response time with/out index on multi-node setup

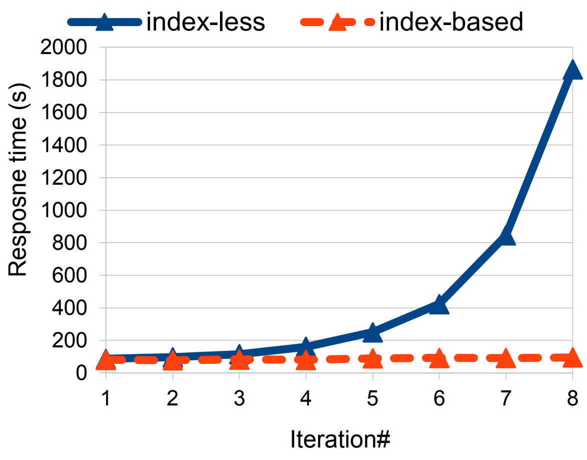


Figure 24 Comparison of joins with/out index in Experiment 2

As indicated in Table 16 and Figures 25 and 26, 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 Experiment 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, 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.

	#1	#2	#3	#4	#5	#6	#7	#8
Size (GB)	0.13	0.19	0.31	0.56	1.07	2.14	4.28	8.58
Time (s)	40.72	56.31	79.23	135.41	257.36	529.33	1307.23	3629.91

Table 12 Index size and index creation time for Experiment 2

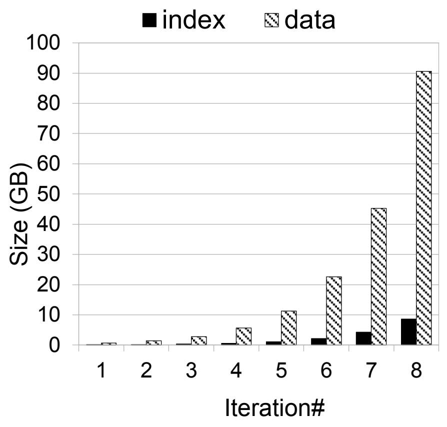


Figure 25 Index size vs. data size

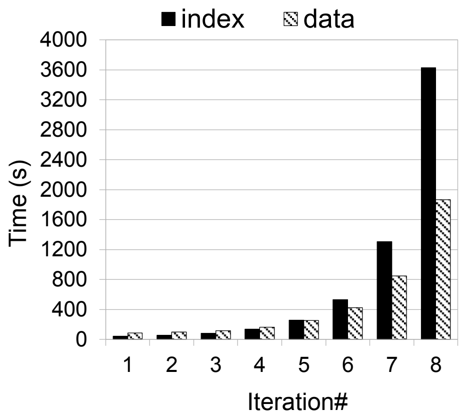


Figure 26 Index creation time vs. query response time

We conducted two sets of experiments using TPCH data and 4 custom queries on both a multi-node and a single-node Hadoop cluster. The dataset size, response time, selectivity ratio, and related costs were considered to evaluate the performance. Overall, our index-based join was faster than the existing Hive approach and the performance gap between the two approaches was wider when query selectivity was considered, assuming the index construction time as the only cost.

Chapter 6

Conclusion and Future Work

Current Hive joins work in a scan-centric manner, meaning that a large number of map tasks get initiated to read the whole dataset. When a small fraction of the whole dataset satisfies the query, this solution becomes inefficient. In such cases, helper structures like indexes can be used to locate records faster.

Indexes have been around for long time and the benefit of using them is obvious. However, deciding when to use indexes in a situation requires extensive evaluation of its cost and performance.

In this work, we used the current Hive indexing structure to speed up join queries. In the first set of experiments in Experiment 1, we observed that generally the larger the data is the larger the performance gain becomes. Our approach grew linearly in all curves shown in Figures 16 to 19. In the second set of experiments in Experiment 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 Figure 22, that the index size was almost fixed at only 15% of the data size in Experiment 1 and in Figure 25, 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 23 and 26 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 Figure 26, 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 m entries, accessing a specific value in the index is reduced from O(n) to O(m) with m much smaller than n.

Hive index maintenance cost is noticeably low, considering the infrequent updates and batch-mode data insertion as the characteristics of big data. If new data is loaded into a new partition of a base table, indexes can be created dynamically for that partition and kept separately without any need to perform expensive update operations.

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.
- Auto-indexing or the ability for the compiler to create indexes internally if proved to be more efficient than the brute-force scanning of the data.
- Index selection in which the best index out of all of the available ones is chosen to be used. The best index could be the smallest or the one with the optimum set of attributes. Current Hive naively picks the first applicable index to execute a query plan.
- Avoiding index creation time by building the index when loading the data into a table. Obviously, in Hive managed tables data is read twice. Once for copying it to the base table and once for creating the index. The former can be eliminated if the index can be created in the background while loading data into a table.
- Implementation of a hash-based index at the bucket level. Buckets, as the smallest data model units in Hive, are potential candidate for the fast hash-based index structure.

- Design of block-scope B+ trees or R-trees or integration of other powerful indexing tools in Hive that could help improve the index performance as in standard database management systems.

Bibliography

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

[2] Amazon DynamoDB [Online]. Available: http://aws.amazon.com/dynamodb/

[3] 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

[4] ANTLR [Online]. Available: http://www.antlr.org/

[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] Apache Hbase [Online]. Available: http://hbase.apache.org/

[7] Apache Thrift [Online]. Available: http://thrift.apache.org/

[8] Atta, F. “Implementation and Analysis of Join Algorithms to handle skew for the Hadoop Map/Reduce Framework,” M.S. thesis, Informatics School of Informatics, U. of Edinburgh, Edinburgh, 2010

[9] Blanas, S., Ercegovac, V., Patel, J. M., Rao, J., Shekita, E. J., Tian, Y., "A Comparison of Join Algorithms for Log Processing in MapReduce," in Proc. ACM SIGMOD Intl. Conf. Management of data, 2010, Indianapolis, Indiana, pp. 975-986

[10] Capriolo, E., Rutherglen, J., Wampler, D. Programming Hive: Data Warehouse and Query Language for Hadoop, 1st ed, O'Reilly Media, 2012

[11] Chandar, J. "Join Algorithms using Map/Reduce," M.S. thesis, CS School of Informatics, U. of Edinburgh, Edinburgh, 2010

[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] DataNucleus [Online]. Available: http://www.datanucleus.org/

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

[15] Eaton, C., Deroos, D., Deutsch, T., Lapis, G., Zikopoulos, P. Understanding Big data: Analytics for Enterprise Class Hadoop and Streaming Data, 1st ed, McGraw, 2011

[16] Garcia-Molina, H., Ullman, J., Widom, J. Database Systems: The Complete Book, 1st ed, Upper Saddle River, NJ, Prentice Hall Inc., 2002

[17] Gilbert, S., Lynch, N. A. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services", in Newslett. ACM SIGACT, vol. 33, issue 2, pp. 51-59, June 2002

[18] Grance, T., Mell, P. "The NIST Definition of Cloud Computing," NIST., Gaithersburg, MD, Rep. Recommendations of the National Institute of Standards and

Technology, 2011

[19] 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

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

[21] Hive index design doc [Online]. Available: https://cwiki.apache.org/confluence/display/Hive/IndexDev

[22] Hive JIRA [Online]. Available: https://issues.apache.org/jira/browse/HIVE

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

[24] N. Jain, L. Tang, "Join strategies in Hive", Facebook, Rep. Hadoop summit 2011, 2011 [Online]. Available: https://cwiki.apache.org/Hive/presentations.data/Hive%20Summit%202011-join.pdf

[25] 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

[26] Lou, W., Ren, K., Wang, C., Wang, Q. Privacy-Preserving Public Auditing for Storage Security in Cloud Computing, Proc. 30th IEEE Int'l Conf. Computer Communications (INFOCOM 10), IEEE Press, San Diego, CA, 2010, pp. 525–533.

[27] S. Madden: "From Databases to Big Data," IEEE Internet Comput., vol.16, issue 3, pp. 4-6, May-June, 2012

[28] MapReduce Tutorial [Online]. Available (date): http://hadoop.apache.org/docs/mapreduce/r0.22.0/mapred_tutorial.html

[29] mongoDB[Online]. Available: http://www.mongodb.org/

[30] Neo4j[Online]. Available(write date): http://www.neo4j.org/

[31] Okcan, A., Riedewald, M. "Processing theta-joins using mapreduce", in Proc. ACM SIGMOD Intl. Conf. Management of Data, Athens, Greece, 2011, pp. 949-960

[32] Rajamaran, A., Ullman, J. D. Mining of Massive Data Sets, Cambridge University Press, 2011

[33] Tiwari, Sh. Professional NoSQL, 1st ed, Wrox Press, 2011

[34] http://www.tpc.org/tpch/

[35] Tudorica, B. G. "A comparison between several NoSQL databases with comments and notes," in Roedunet International Conf. (RoEduNet 10), Iasi, Romania, 2011, pp.1-5

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