A MAPREDUCE RELATIONAL-DATABASE INDEX-SELECTION TOOL

by

Fatimah Alsayoud

Bachelor of Computer and Information Sciences in the field of Information technology,

King Saud University, 2008

A thesis

presented to Ryerson University

in partial fulfillment of the
requirements for the degree of

Master of Science

in the Program of

Computer Science

Toronto, Ontario, Canada, 2014

©Fatimah Alsayoud 2014

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my thesis may be made electronically available to the public.

A MapReduce Relational-Database Index-Selection Tool

Master of Science 2014

Fatimah Alsayoud

Computer Science

Ryerson University

Abstract

The physical design of data storage is a critical administrative task for optimizing system performance. Selecting indices properly is a fundamental aspect of the system design. Index selection optimization has been widely studied in DataBase Management Systems (DBMSs). However, current DBMS are not appropriate platforms for many data nowadays. As a result, several systems have been developed to deal with these data. An index-selection optimization approach is still needed in these systems. In fact, it is even more necessary since they process Big Data. Under these circumstances, developing an index-selection tool for large-scale systems is a vital requirement.

This thesis focuses on the index-selection process in HadoopDB. The main contribution of the thesis is to utilize data mining techniques to develop a tool for recommending an optimal index-set configuration. Evaluation shows significant performance improvement on the tasks running time with the tool index-set configuration.

Acknowledgements

First and foremost, I am sincerely thankful to my supervisor, Dr. Ali Miri for encouragement, support and guidance. I am indebted to his understanding of my personal circumstances and the respect that I always received from my supervisor. Finally, and most importantly, I must express my gratitude to the love, encouragements and continued support that I have received from my parents, husband and siblings and certainly to the inspiration that I have got from my little angel.

Contents

1	Introduction	1
1.1	Research Motivation	3
1.2	Contributions of the Research	4
1.3	Thesis Outline	5
2	Related Work	6
2.1	Big Data	6
2.2	Large Scale Data Processing Systems	9
2.2.1	Parallel Database Management Systems	9
2.2.2	MapReduce-Based Systems	11
2.2.3	MapReduce-Based vs. Parallel Database Management Systems	19
2.2.4	MapReduce and DBMS Hybrid Systems	24
2.2.5	HadoopDB system	25
2.3	Hadoop Indexing	31
2.4	DBMS Indexing	33
2.5	Index Selection	35
3	Our Index Selection Tool	39

3.1	Frequent Itemset Mining	39
3.1.1	Apriori Algorithm	41
3.2	Index-Selection Tool	45
3.2.1	Objectives	45
3.2.2	Design	46
4	Experimental Results	50
4.1	Experimental Environment	50
4.1.1	HadoopDB Settings and Configuration	52
4.1.2	DataSets	53
4.1.3	Analytical Tasks	54
5	Conclusions and Future Work	63
5.1	Conclusions	63
5.2	Future Work	64
	References	69
	Glossary	70
	Acronyms	71

List of Tables

2.1	Parallel Database Management Systems Vs. Hadoop.	24
3.1	Example Transactions Database	41
4.1	HadoopSetting	52
4.2	Datasets Details	54
4.3	Large Aggregation Task Response Time in seconds with and without index	58
4.4	Small Aggregation Task Response Time in seconds with and without index	59
4.5	Join Task Response Time in seconds with and without index	60
4.6	Selection Task Response Time in seconds with and without index	61

List of Figures

2.1	HDFS Architecture.	12
2.2	The Word Count MapReduce data process flow.	15
2.3	MapReduce Execution Overview.	16
2.4	Physical and Logical Division in MapReduce Framework.	18
2.5	HadoopDB Architecture.	26
2.6	Hive and HadoopDB SMS execution Plan.	30
3.1	Apriori Algorithm Example	43
3.2	Index Selection Flow Chart	44
4.1	Large Aggregation Task.	57
4.2	Small Aggregation Task.	58
4.3	Join Task.	59
4.4	Selection Task.	60
4.5	Tasks Response Time on Different Index Configurations.	61
4.6	Compare Tasks Response Time on Index-based and Index-less configuration.	62

Chapter 1

Introduction

Data volume increases every day since data generation has become a technology that is used daily, not only by companies and systems, but also by users. Data comes from machines, mobile devices, networks and social media. Data that have been generated in the last two years represent 90 percent of all data [1] and the volume is increasing rapidly every day, resulting in the term *Big Data* being introduced and used. More precisely, Big Data is data that is sizeable (*Volume*), contains different data types (*Variety*) and flows continuously (*Velocity*). These characteristics are known as the *3Vs* model.

To fully benefit from Big Data potentials, there is a need to develop efficient data analysis tools that can provide better search results and provide valuable decisions and design high-quality strategies. However, Big Data does not fit in traditional data process and storage systems such as a DBMS. This is because a DBMS is not designed to store huge amounts of data, manage different types of data or to scale up when the data grows rapidly. New data characteristics have inspired researchers and developers to build new systems or to modify existing systems to process and store Big Data such as Hadoop, HadoopDB, MongoDB and the Oracle NoSQL Database.

Managing Big Data is a fundamental task to be able to effectively benefit from collected information. There are several possible uses of extracted information such as the efficient design of system and

effective decision making. Managing huge and highly heterogeneous data and its correct interpretation is a challenging task. Therefore management tools become pivotal on Big Data systems.

Planning efficient system design from existing information could significantly improve system performance. One of the main aspects in the physical design of a system is indexing. In general, an index is created to speed the information retrieval process but not all index configurations are useful for the system; some indices just reserve space and do not add any benefit to the system. Building indices is an expensive and time consuming operation so building all candidate indices is a naïve method that may be useful in the case of small data but it could have a negative impact in the Big Data case. Index configuration design and selection are dependent on the system workload and dataset schema. Knowledge discovery techniques are widely used to extract information from workload queries for the purpose of index optimization. There are several methods for recommending index configurations. One of them is to analyse the workload to compute the attributes and attribute set frequencies which could be computed by frequent itemset mining. It would help the administrator to make efficient index configuration design decisions. While some attributes should have an index, since they appear in many queries, building indices for infrequent attributes may not benefit performance.

In the last two decades, research into optimizing the physical design of indices mainly have relied on traditional database systems for storage engines and on SQL for the query language. Thus these techniques have a rich history, not only in this area, but also in many other data management areas from research to commercial companies. Although Big Data techniques outperform DBMS on some data management features such as scalability and fault tolerance, they are still not comparable to DBMS's historical gains. Consequently, new systems could learn a lot from DBMS. However, adapting methods from one system to another is not a straightforward process due to system model variation.

The goal in this thesis is to implement an index-selection tool on HadoopDB[2]. HadoopDB is a platform combining a relational database and a MapReduce model (a Hadoop communication layer). The proposed tool analyses HadoopDB query workloads and applies Frequent Itemset mining methods

to recommend an optimal index-set configuration.

1.1 Research Motivation

Automatically selecting an appropriate index-set configuration received a great deal of attention in DBMS physical design optimization research. When a query is sent to the database, the DBMS optimizer chooses the optimal plan for executing the query and that plan is usually faster and cheaper than others, depending on the quality of the optimizer. The execution plan is the logical arrangement of operations. For example, the optimizer decides if the join operation in a query should execute before or after the selection operation. In addition, it decides if the operation run using a brute-force scan method and read the whole database or if it should access one of the existing indices or materialized views first then read only the relevant records in the database. In general, indices speed up the execution of commands by avoiding accessing records that are not relevant to the query. Clearly, logical execution performance is dependent on the quality of the physical design.

Query optimization is an essential aspect of system performance regardless of the system model. Executing all queries with the same plan is a naïve approach. Although MapReduce-based systems outperform DBMS on queries requiring scanning the whole datasets, brute-force scanning is not an effective approach for selective queries. Several methods and systems were developed to resolve this issue by adding indexing feature to the MapReduce-based systems. However, these indices still need to be selected accurately.

HadoopDB systems benefit from both the MapReduce distribution model and DBMS functionalities. The system queries utilize indexing features in the database layer. MapReduce acts as a communication layer to distribute and assign queries to DB nodes in a cluster, while DB nodes run an instance of a database server as a storage layer. Each DB node executes queries separately on the local storage then sends back the result to the MapReduce framework. Each Map gathers the results from the local DB

and sends it to the reduce phase to finalize the operation.

Carefully selecting the index-set configuration in large systems is more urgent than on smaller systems. This is because the index size will be large on these system, and will require more storage space. Moreover, the quality of the index configuration does not only affect one node in the cluster, it affects the performance of the whole cluster.

Queries are executed in two layers in the MapReduce database which requires optimization techniques to interact with both of them. In contrast, in DBMS the whole execution process works on the same engine. To select an index in MapReduce database systems, query workloads are collected from MapReduce frameworks for analysis and recommendation for index-set configuration for the DB cluster are made. Query workload contains information about all queries executed on the system, processed over the MapReduce and database layers.

This thesis mainly aims at improving performance in MapReduce database systems like HadoopDB by applying data mining techniques on the query workload to find the optimal index-set configuration the physical design of the DB layer. In other words, the proposed tool utilizes frequent itemset mining methods and extracts useful knowledge from the system workload to automatically recommend an index-set configuration that reduces task execution time.

1.2 Contributions of the Research

The goal of this research is designing an index-selection tool for HadoopDB, large scale hybrid MapReduce and DBMS systems to improve system query execution performance. The key contributions of this research are summarized as follows:

- A detailed investigation of HadoopDB open-source infrastructure [3] to identify the system query execution model that can be used to design a selection tool. HadoopDB's main objective is to leverage Hadoop distribution and parallelization capabilities to manage multiple single-node Post-

greSQL database layers instead of using Hadoop Distributed File System (HDFS). The MapReduce communication layer coordinates and schedules queries in parallel, while the PostgreSQL storage layer stores, indexes and manages the data. Queries generated in MapReduce layer, push to the database layer, execute separately on single-nodes, then all results are sent to the MapReduce layer to finalize the query computation.

- Implement Apriori algorithm in the MapReduce framework: In general the algorithm requires huge computation and consumes memory even in non-large systems and it is more expensive on a large dataset. Executing the algorithm in the MapReduce model improves the algorithm's performance by reducing the complexity of operations. The candidate indices are counted on multiple processes in parallel instead of on one processor in order to divide the computing burden and improve the algorithm efficiency significantly.
- The main contribution of this work is to build an index-selection tool for the HadoopDB system to recommend an optimal index-set configuration. The tool analyses the system query workload by applying the Apriori algorithm for frequent itemset mining to extract information about query attributes, and then recommends a database index-set configuration. Experimental results show performance improvement in query execution time using the proposed index-set configuration tool over the index-less model.

1.3 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 provides background information and a brief survey of related work. Chapter 3 includes details of the proposed index-selection tool design and its implementation. Tool evaluation and experimental results are presented in Chapter 4. Finally, the thesis is concluded and future work is presented in Chapter 5.

Chapter 2

Related Work

This chapter covers background information and related work to this thesis. It includes information about Big Data, large scale data processing systems, indexing in Hadoop and DBMS, as well as DBMS and MapReduce research related to our work.

2.1 Big Data

Data increase very rapidly in this age of internet, due to peoples' daily activities as well as companies' operations. According to a report by International Data Corporation (IDC) in 2011[4], data had grown nine-fold in a span of five years, and this volume is expected to increase even more every year. *Big Data* is the term that is used to describe the explosion of data. Generally, Big Data is a very large dataset with different types. Usually it is differentiated from other data by three keys attributes: *Volume*, *Velocity* and *Variety*, commonly referred to as the 3Vs model, which are briefly described below.

Volume: In general, end-users normally generate, store and access huge amounts of data every second by exchanging images, uploading and downloading videos, click streams and email. Data does not only come from internet surfing but also from daily practices such as shopping, banking and

e-health operations. Indeed, the amount of data generated increases in parallel with the increase in technologies and applications and it is no longer in the range of gigabytes or terabytes, but rather more than petabytes and exabytes. More data means more opportunities for analyses and better decision making. However, large data requires special tools and technologies for storage and analyses instead of traditional database techniques.

Velocity: Data is created, moved and shared very quickly. Social media data is a clear example of how messages spread in seconds. Real-time data is very useful for many companies and applications when data must be analyzed at the time it is generated, for example in traffic information for Global Positioning Systems (GPSs).

Variety: The enormous streams of data come from everywhere: smart phones, sensors, machines, and social networks and form structured, unstructured and semi-structured data models. Different data types are generated, such as text, image, video, sensor data and voice. Big data technologies should be able to store and process all data types effectively.

Big Data provides a great opportunity for quality improvement if it is effectively utilized. However, it also introduces new data management challenges as traditional database management systems cannot effectively handle scalability and fault-tolerance of data system, and different data types in them. Several solutions have been proposed to deal with various Big Data management issues.

Storage solutions: Big Data storage mechanisms should be able to store large-scale of datasets with diverse data structures and achieve data access reliability and availability. Many storage mechanisms have emerged to meet Big Data requirements. In general, the mechanisms could be classified into two classes, file systems and databases.

- File systems control storage and retrieval of distributed files on the system. The Distributed File System (DFS) was first introduced on Windows servers as a client/sever-based application to allow sharing of files stored on a global server. To support large-scale data on inexpensive servers

and to maintain fault-tolerance and high-performance services, Google expanded DFS into Google File Systems (GFSs)[5]. Later, HDFS and Kosmosfs file systems [6] were derived from GFS open source code. More Big Data storage solutions were developed after that for different purposes. For example, Cosmos by Microsoft was founded for business advertisement and Haystack by Facebook was created for small-sized photos[7].

-A database is a set of data organized within certain data model restrictions. Traditional relational databases do not support all Big Data types and cannot handle data scalability. NoSQL databases were motivated by that perspective. NoSQL DB is a popular Big Data storage solution and usually it is faster than relational databases since operations in its environment do not have to follow a specific schema so it can run without dealing with many restrictions. NoSQL DB includes three main storage types: document-oriented, graph and Key-value storage. On the negative side, most NoSQL DBs do not meet Atomicity, Consistency, Isolation, Durability (ACID) properties. In addition, there is no standard low-level query language for NoSQL DBs, like SQL in DBMS [8].

Computing solutions: Parallel and distributed processing is a successful and dominant Big Data computing solution. MapReduce, the powerful Big Data computing solution, utilizes parallelism for large-scale computing. The MapReduce model splits problems into several smaller problems and distributes them over a cluster in parallel for processing. Jobs in this model simply create map and reduce functions without specifying networking information that is embedded in the model. The model controls data distribution, job scheduling, node resource monitoring, and task coordination.

Infrastructure solutions: Cloud Computing guarantees smooth operations of the Big Data. It provides scalable clusters where nodes can be added or removed easily depending on need. In addition, Cloud Computing is designed to serve inexpensive virtual low-end machines that are appropriate for multi-node parallel computing. Big Data technologies usually demand a very large cluster

of nodes up to thousand which are very expensive to set up physically, whereas utilizing virtual clusters in the Cloud is a flexible and cheaper alternative.

Technology solutions: The research and business communities have developed different Big Data technologies. Each has different defining characteristics, depending on need of the the application. Some of them depend on file system storage like Hadoop, while others rely on NoSQL DB such as (I) Dynamo and Voldemort key-value databases, (II) BigTable, Cassandra and HBase, column-oriented databases (III) MongoDB, SimpleDB, and CouchDB, document databases.

2.2 Large Scale Data Processing Systems

Data increases rapidly every day and finding an appropriate system to process that amount of data efficiently and in an acceptable period of time is an issue. The volume, velocity and variety of data that comes from applications today challenges traditional database systems. Data processing and storage methods in database systems are not designed to handle such large amount of data, which is known as Big Data. Since the last decade, many new approaches, such as MapReduce-based systems, have been developed to deal with Big Data processing and storage issues. However, there is an argument that traditional database systems perform better on querying tasks while MapReduce-based systems work best with data loading [9]. This argument leads to the development of hybrid systems that have the advantages of MapReduce-based systems and traditional database systems to provide superior data processing results in the Big Data environment.

2.2.1 Parallel Database Management Systems

Parallel Database Management System (PDBMS) architecture designed by Gamma [10] and Grace [11] to store databases on a cluster and run queries on parallel began to flourish in mid-80s. Despite the rich history and many performance improvements that PDBMS have gained from academia and industry,

they are still ineffective for analysis of large-scale data.

The main rational behind PDBMS architecture is to facilitate executing a single query in cluster nodes simultaneously to achieve outstanding data-processing performance results. Adding nodes to PDBMS clusters supplies more data storage capacity than one server can afford, since single servers, even those with large memory and disk space, are inadequate to store and process data in terabytes.

PDBMS can be built on shared-memory, shared-disk or shared-nothing architectures. Shared-memory architecture means that the nodes in the cluster can share memory space. Shared-disk architecture means that nodes share storage but they have private memory and shared-nothing architecture means each node in the cluster has private storage and memory. Shared-nothing architecture is the one that is used in the Cloud environment since nodes are not usually physically close to each other. The cloud environment is mostly used for massively parallel database systems. In this work, we will refer to shared-nothing PDBMS architectures as PDBMS.

Data can be partitioned and distributed across PDBMS clusters for parallel computing following various partitioning designs. Each node works independently on its own split and keeps some information about the other splits' locations, replications and the applied partitioning function. This information is important for query execution and managing nodes in the cluster. After executing queries, each node sends the intermediate result to the responsible node to be merged with other results and to finalize the query execution. To economize storage capacity and accelerate execution, the intermediate results are not stored locally in the executing node before being sent to the responsible node which speeds up the process but reduces fault-tolerance. For the query writing simplicity, PDBMSs employ a declarative language layer on top of the structured storage database. Indeed, in most cases, all PDBMS queries are expressed through SQL language or through SQL connector when the queries run over other environments. For instance, Java Database Connectivity (JDBC) is required to run database SQL in Java environments.

Data content characteristics in DBMS are managed through schema. Schemas work as data cat-

alogues, containing information about all tables and attributes, domains and relationships. Schemas protect databases from any additions or modifications that violate integrity. In other words, any transactions not matching the schema are rejected. These restrictions guarantee data consistency in the database.

Although PDBMS have been successfully processing and analyzing data over the past few decades in various industries and have produced good results with efficient querying tasks, they cannot manage rapid increase in data diversification coming from clickstream, sensors, server logs, machines and other applications due to the length of time that parallel database systems take to load large amounts of data as a result of PDBMSs' loading process. Furthermore, parallel database clusters cannot scale to thousands of nodes. The largest known database cluster has just around a hundred nodes[12].

2.2.2 MapReduce-Based Systems

Due to PDBMS limitations, many systems are implemented on a MapReduce model to provide simple shared-nothing cluster communication layers, avoid bottlenecks, support fault tolerance and to be able to scale cluster to more than a few thousands nodes. The MapReduce computing platform follows the same parallel computing concept as PDBMS which runs on distributed database. However, MapReduce is superior on parallel computing in large-scale data. In a like manner, PDBMS runs on structured data while MapReduce is compatible with any other data storage type including structured, unstructured and semi-structured.

The best known MapReduce-based open source system is Hadoop. Hadoop entices a large user community in both academia and industry due to, not only its availability and low-cost installation compared to other data communication and storage systems, but also its strong management features in large clusters. Hadoop consists of two core layers: the HDFS highly-scalable data file storage system and the MapReduce data processing layer. Both layers have master/slave architectures. There is just one master node in MapReduce and HDFS layers are called *JobTracker* and *NameNode*, respectively.

In some cases, the same node can perform as a JobTracker and a NameNode. MapReduce and HDFS layers can have one or more slave nodes called *TaskTrackers* and *DataNodes*, respectively. It is worth mentioning that the master node can also work as a slave at the same time.

Figure 2.1: HDFS Architecture[13].

Hadoop Distributed File System (HDFS) Layer

The Hadoop Distributed File System is a chunk-based distributed storage layer in Hadoop where the file is divided into chunks and those chunks are distributed across the cluster. HDFS differs from other distributed file systems in many ways. Firstly, HDFS data replication procedures and distribution processes over the cluster nodes to provide high fault tolerance features in the system. In addition, HDFS does not require high-end costly machines; it is designed to be established on low-end machines

and it can handle very large data that other systems cannot. Data is stored in HDFS as blocks where the default block size is 64MB. Files are stored in one block unless the file size is greater than the block size, in which case the file is divided into several blocks. For example, if a file size is 140MB and the block size is just 64MB, the file resides in three blocks. Every block is stored in the HDFS is replicated three times by default. The number of replications across the cluster can be specified by the user to determine fault tolerance levels, depending on the cluster size and setting. Data block splitting and replication, cluster resource monitoring and managing, and tracking blocks location on DataNodes are the responsibilities of the NameNode. In detail, slave DataNodes store blocks and their replications and master NameNodes manage them through metadata. DataNodes send heartbeats and report about the blocks existing in the DataNode to the NameNode regularly. The HDFS design assumes that the data is written once but is read several times to provide fast processing in large-scale data. HDFS architecture is illustrated in Figure 2.1.

Hadoop MapReduce Layer

Hadoop distributed computation uses MapReduce model as a communication layer. MapReduce is a functional programming model that splits big computing problems into a set of independent small problems and processes them separately across a cluster. It was popularized by Google [14] and it contains two main functions: the Mapper and the Reducer which were designed to process unstructured large data sets and executes jobs in parallel manner. The MapReduce framework is responsible for the job execution plan, distributing work, launching tasks and relaunching failed tasks. It controls cluster networking operations. The model's general process is in four steps: (I) Pre-Load data: The InputFormat reader reads datasets from the storage system e.g.DFS and iterates over the dataset to split it. Splits are then sent to the map phase. There is one map for each split. (II) Map phase: The *Map()* function process the split to generate a list of Key Value Pairs (KVPs). (III) Shuffle and Sort: These shuffle and sort fetch and group the intermediate values by key for all mappers. (IV) Reduce Phase: The *Reduce()*

function generates output after gathering each group by key. The Word count MapReduce task is usually used to explain the model job processing procedure [15]. It counts words frequencies in a dataset. The pseudo-code could be written as follows:

```
map(String key, String value)[15]

// key: document name

// value: document contents

for each word w in value

    EmitIntermediate(w,1);
```

EmitIntermediate method in map function emits each word in the input data with its number of occurrences. (for simplicity, this pseudo-code assume that each word occurred just one time).

```
reduce(String key, Iterator values)[15]

// key: a word

// values: a list of counts

int result = 0;

for each v in values:

    result += ParseInt(v);

    Emit(ToString(result));
```

Emit method in Reduce function, grouped and summed all frequency values for a certain key(word).

Figure 2.2 presents word count task data flow in MapReduce.

When JobTracker receives a client-submitted job, it contacts a DataNode to fetch data from HDFS in KVPs and assign each of them to one TaskTracker map phase. Distribution and assignment of tasks are performed under the cluster scheduling algorithm rules and node resource availability that are known

Figure 2.2: The Word Count MapReduce data process flow.

by the TaskTracker heartbeat. TaskTrackers slave nodes take their assigned job from the JobTracker and process map phase independently and in parallel and store the intermediate results locally. Whenever all map tasks for a certain job are complete, the OutputCollector is partitioned, shuffled and sorted depending on the intermediate key and distributed to the reduce phase. The reduce phase cannot be performed until all map tasks are completed thus each TaskTracker notifies the JobTracker about task completion. Outputs with same key are processed by the same reducer and all reducers work in parallel and store results in HDFS. JobTracker's responsibilities do not finish after distributing and assigning jobs to Tasktrackers, it continues to monitor tasks running to re-execute failed jobs during execution on another node. Figure 2.3 illustrates MapReduce execution steps.

MapReduce Input Format Classes

HDFS layer manage and store the data physically and MapReduce layer process the data logically. When a MapReduce job is submitted in Hadoop, a few preprocessing steps are run before executing data in the map phase. First, the readers class checks if the input data meets the submitted job data specification.

Figure 2.3: MapReduce Execution Overview[15].

Then the input data is logically divided into splits where each split is processed by one Mapper. Finally, the Mapper maps the split records to KVP. In practice, the input data is checked by an `InputFormat` interface, split by the `InputSplit` method and the process is splitted by the `RecordReader` method. Split does not contain any data; it just pointer to the data location. The splitting characteristics can be modified to meet the application requirements either at the job configuration level through `InputSplit` or at the cluster level, where the change affects all jobs running in the cluster, through changing the splitting property in the Hadoop configuration file. Some applications force the system not to split a file at all. For the purpose of checking if a file is sorted, the map function is required to run on an entire file at the same time, instead of processing the file record by record, to compare the values with others [16]. In this case, the application just runs one mapper while the number of mappers is equal to number of the splits. Split size may be equal to or greater than HDFS block size, so each refers to one or more blocks. See Figure 2.4.

Storing and processing a variety of data types is one of the fundamental features in the MapReduce

framework. Different data types can be loaded and stored in HDFS and one reader could not read all of them since each data type demands particular processing procedures. To support several data types, the MapReduce framework provides `InputFormat` interface which contains different classes for individual input data types like `FileInputFormat`, `TextInputFormat` and `DBInputFormat`. Each class reads and splits data differently to convert it into a list of KVPs. For instance, `TextInputFormat` reads split content and generates KVP1 lists where the lines represent values and a line's offset represents keys.

In general, the MapReduce model process the data in a KVP representation in three stages. First, data is entered into the map phase as KVP1 and is processed to generate a KVP2 list. After that, K2 and V2 list are paired to generate a KVP3 list. KVP1 and KVP3 can be processed within different data type classes, but the KVP2 list and the K2 and v2 list pairs should be processed within the same data type class. In other words, map phase data input type can be different from the reduce phase data output type, but the map phase intermediate KVP output should match the reduce phase KVP input data format.

Hadoop Execution Plan

Hadoop works, in general, in five main phases (*Load*, *Pre-load data*, *Map*, *Shuffle* and *Reduce*). The execution plan can be customized by modifying phase functions for each job separately. The data first **loads** for storing in HDFS and is divided horizontally into blocks which can be specified by the partitioning operator. Each block is replicated either by the default replication parameter which is three or by a user defined parameter. Each replica is fetched and stored in a separate node. Next, the data **Pre-loads** and splits logically for processing following the `InputFormat` splitting procedure. The programmer could use certain `InputFormat` class to read the file in a specific way or can customize one of the build in user-defined functions (UDFs) to modify how the data is read and processed to meet the application requirements. Each split is managed by a separate **Map**. The programmer's could modify the split UDFs to change the way the data is processed. Map output is stored in the disk as intermediate

Figure 2.4: Physical and Logical Division in MapReduce framework.

data. The **Shuffle** redistributes this data with sort and merge UDFs to the final phase. **Reduce** groups the redistributed data with the user grouping UDF.

Hadoop Related Projects

Many projects have been designed on top of Hadoop either for adding missing aspects like HBase for supporting database or simplifying the system coding that is mainly written in the Java programming language, like Hive. Below are some of the Hadoop projects related to work in this thesis:

Apache Hive[17] is a data warehouse execution and managing interface, originally developed by Facebook. It provides HiveQL for ad-hoc queries on top of large-scale distributed data storage to simplify writing a query on a MapReduce programming model. Hive turns queries written in

HiveQL into the map and reduce model before submitting to a large-scale cluster. The software allows writing queries, not only on HiveQL, but also in the map and reduce model. It is important to realize that Hive does not have all SQL characteristics. For example, indexing techniques are not fully supported. Hive does not yet provide joint indices.

Apache Pig[18] provides a simple scripting language called Pig Latin to support data flow in Hadoop for analyzing data in large-scale clusters. In fact, the results generated by a Pig Latin could act as an input for another Pig Latin construct which could work as a query execution plan. Like Hive, Pig converts script written in Pig Latin into the MapReduce model before executing tasks.

Apache HBase[19] is a non-relational distributed database for Big Data in a scalable cluster. HBase is categorized as a NoSQL data storage system without many of RDBMS' core features such support for a secondary index. Hbase build large tables on top of HDFS to provide real time read and write.

2.2.3 MapReduce-Based vs. Parallel Database Management Systems

MapReduce-based systems have emerged to provide features that are missing on DBMS and required by Big Data applications such as scalability and fault-tolerance. Conversely, the new systems relax several DBMS core features that are still needed for data analysis queries like indexing and query optimization. Each system outperforms the others in certain aspects. Therefore, administrators must choose a platform after analyzing the needs of the the application. Table 2.1 on page 24 compares the main features of Hadoop -MapReduce-based Systems- and PDBMS.

Data Model: Data is stored and manipulated in the relational model in PDBMS where data is organized in tables. By contrast, the MapReduce programming model stores and retrieves data in a KVP fixable structure to allow managing verity of data types.

Data Type: The system data model determines which data types are supported. The PDBMS rela-

tional model limits support to structured data which does not serve the needs of qualitative data that comes from application with unstructured and semi-structured data such as social network data, click-stream data, machine data, sensor data and images. For this reason, the MapReduce model was developed on KVP fixable data, representative of all data types.

Query Operations: Relational algebra operations such as selection, projection, joining and aggregation are the main operations used by database queries and are included on DBMS framework for direct use. These operations are not built-in to the MapReduce model, but can be implemented and added manually in the Map and the Reduce functions. For example, an aggregation task to compute the total adRevenue generated per sourceIP from UserVisits table [9] could be written in SQL as follows:

```
SELECT sourceIP, SUM(adRevenue)

FROM UserVisits

GROUP BY sourceIP;
```

To execute the same aggregation task in the MapReduce model; the Mapper works over the records to extract adRevenue and sourceIP then *Emit* adRevenue are keyed by sourceIP where the values (adRevenue) are automatically grouped by key(sourceIP) and, finally, the Reducer computes the total of all adRevenue emitted for certain sourceIPs. The pseudo-code could be written as follows:

```
map(LongWritable key, Text value)

// key: Record offset

// value: Record content

Extract adRevenue AND sourceIP from each value

//newKey = sourceIP

//newValue = adRevenue
```

```

Emit(newKey,newValue);

reduce(Text key, Iterator values)

// key: sourceIP

// values: a list of adRevenue

int sum = 0;

for each adRevenue in values:

    sum += ParseInt(adRevenue);

Emit(ToString(sum));

```

Data Processing: Hadoop is superior to DBMS in data loading time [9] because the data in Hadoop is batched to the storage engine without indexing the data or checking data consistency. Data preprocessing, like creating an index in DBMS, cause longer loading times. In contrast, Hadoop has worse performance than DBMS, in transaction tasks. This is because Hadoop must scan all data, even for tasks that required retrieving just a few records, since Hadoop does not benefit from built-in indices and views like DBMS [9].

Query optimization: DBMS optimizers rewrite queries depending on an optimal execution plan in order to speed up execution. MapReduce models execute queries as coded in the map and the reduce functions and leave the optimization task to the programmer. However, writing optimal code for each individual query is quite complex and that optimal code cannot be re-used on queries with different operations and attributes. Optimal execution plans reduce the running time by reducing the number of operations in the query. Optimizers generally follow the same algebraic optimization rules for operation reduction. To demonstrate query optimization, an algebraic optimization example is described for $X = (((y * 5) \div 1) + (y * 2)) + 0$, using within algebraic laws:

1. (+) Identity law for addition $y + 0 = y$
2. ($\div$) Identity law for division $y \div 1 = y$
3. (*) Distributive law $n * (y + z) = (n * y + n * z)$
4. (*) Commutative law for multiplication $y * z = z * y$

The result is just two operations instead of five: $X = y * (5 + 2)$.

Support Schema: Schema in the data management layer are accessed by all data applications to verify that insert, update or delete transactions do not violate the data integrity constraints. DBMS supports schema to simplify multiple-application data sharing where data properties such as type, size and domain are available. Providing schema for various data types is a complex task, thus Hadoop does not support schema on its environment. Schema can be simulated on Hadoop for each application individually by writing custom data parsers and creating metadata. In this case, schema can present only within an application and are not accessed by other applications that may share data and can easily corrupt it. To make schema available to other applications, metadata must first be built for all datasets one by one, then a specific reader must be developed to interpret the metadata before executing any queries.

Querying Model: Query languages simplify writing transactions to store, manage and retrieve data from the data storage engine. Commercial DBMS provides a primary distinguishing operator, the SQL query interface, and the storage engine within the same box processing. Hadoop was written for general-purpose complex statistical queries and provides map and reduce functions as the querying model. It does not contain any built-in query languages. Complex queries cannot be expressed in relational algebra, like manipulation of images and videos can be hard coded. It is important to realize that several Apache Hadoop foundation have been created to add declarative query language features to Hadoop systems such as Hive and Pig.

Support Indexing Indices are an essential approach for dramatically reducing the scope of searches . DBMS provides various index structures like hashing , B-trees, single and multiple indexing to avoid processing records irrelevant to a query. Hadoop does not have the same built-in indexing structure, however, indexing techniques could be simulated for Hadoop by customizing the system's function or utilizing other connected system's indexing technique as detailed in Hadoop indexing section 2.3.

Scalability: DBMS cluster architecture was designed to contain dozens of nodes and, in the best case up to a few hundreds of nodes. In addition, scaling DBMS clusters is costly since it usually requires high-end machine and fee-based DBMS. Open source Hadoop is mainly designed to support scaling of clusters up to thousands of shared-nothing low cost machines.

Fault-tolerance: The cluster storing intermediate results, checkpoint creation and relaunching of failed nodes during query execution determines the fault-tolerance support level of the system. Nodes in the MapReduce cluster store intermediate results locally and in the case of node failure, only failed node jobs are re-executed on other nodes while other results are fetched from working nodes. In DBMS, nodes executed queries and pushed the intermediate results generated across the cluster. This requires relaunching of all nodes if one node has failed [9]. In addition, the MapReduce model frequently generates checkpoints for any committed sub-task in order to recover from any failure within a short time. However, the number of checkpoints is inversely proportional to execution performance. For this reason, DBMS does not provide as frequent checkpoints as MapReduce to increase system performance [2]. In other words, the checkpoints keep information about all committed transactions for recovery purposes and must be generated carefully since too frequent checkpoints have a negative impact on performance. While the main assumption in a DBMS is that failure is uncommon, the MapReduce model is designed to run on low-end nodes in large-scale clusters where nodes could be down at any time.

Feature	Parallel DBMSs	Hadoop
<i>Data Model</i>	Relations	KVP
<i>Data Type</i>	Structure	Structure, Unstructured and Semistructured
<i>Query Operations</i>	Select, Project, Join , Aggregate,...	Map and Reduce
<i>Data Processing</i>	Online Transactions	Offline Batch
<i>Query optimization</i>	Build in within the system	Implement by programmers
<i>Support Schema</i>	Schema	Schema-less
<i>Querying Model</i>	Declarative Queries Languages(SQL)	Functional Programming
<i>Support indexing</i>	Native index support	Customized by programmers
<i>Scalability</i>	Max 100 nodes per cluster	Up to thousands nodes per cluster
<i>Fault-tolerance</i>	Unsupported	Supported

Table 2.1: Parallel Database Management system Vs. Hadoop.

2.2.4 MapReduce and DBMS Hybrid Systems

MapReduce-based systems and DBMS both have weak and strong features with respect to processing and analysis of large-scale data as described in the previous section. Thus making a decision about which system should be used is mainly dependent on the needs of the application. While storing social media type data is not best suited to DBMS like systems due to lack of high scalability and variety applications with sensitive information like financial, governmental and health care information cannot neglect data integrity requirements that is not totally supported by MapReduce-based systems. However, they are all looking for high performance queries that are provided by DBMS and fault-tolerance provided by the MapReduce model. From this perspective, the research and business communities have tried to develop several solutions to integrate MapReduce and DBMS and bridge the gap between them to effectively process large-scale structured data in parallel.

In general, MapReduce-based systems could obtain DBMS features at two levels, the *interface-level* or the *system-level*. In the interface-level approach, the solution simply adds a layer to simulate the

DBMS features without modifying or adding to the core system components such as Hive[17], Pig[18], Cloudera Impala[20] and Sqoop[21]. For example, Apache Hive was created to add an SQL-like language interface to data stored on HDFS as files, not as a relational table. Consequently, underlying database characteristics like schema and indexing are not fully supported. In the system-level approach, the solution was a hybrid of MapReduce-based systems and DBMS core components. Hybrid systems apply MapReduce at the communication layer and database for storage layer. The MapReduce communication layer provides scalability and fault-tolerance, while the database storage layer provides the system high performance. HadoopDB[2] and more recently BigSQL[22] are two examples of open source Hybrid systems. Hadept, Greenplum and Aster Data are examples of commercialized systems.

Combining layers from different systems brings some challenges like memory use, layer connection and execution optimization plans running on two layers. In DBMS, both the query engine and the storage engine run in the same database, thus they can share the same memory. However, this is not the case in hybrid systems where the MapReduce model is designed to be independent in the storage layer. In addition, hybrid systems require specific readers to load data from particular database storage instead of HDFS. Queries execute over the MapReduce layer first, then the database layer, and, finally back to the MapReduce layer where part of the query is executed by the MapReduce and by the rest of database. Thus, a query execution plan must be designed for both layers. The MapReduce did not originally contain a built-in optimizer and optimization design was a developer's task. By contrast, an optimizer is one of the main components in DBMS. Adding optimization feature into hybrid systems is quite a complex task, and to the best of our knowledge, there is no work on hybrid system optimizer to date.

2.2.5 HadoopDB system

Nowadays, structured data and other data typed has exploded due to everyday activities. The collected and generated data must be stored and processed effectively and analyzed properly, otherwise it has

Figure 2.5: HadoopDB Architecture [2].

no value. PDBMS was the earliest solution for processing and storing massive structured data. The main issues with PDBMS, as previously discussed are the lack of scalability and fault-tolerance support. Furthermore, PDBMS is costly compared with the free, open-source MapReduce framework. MapReduce is a recent solution for processing massive data and it provides a strong and low-cost distribution framework. However, PDBMS still performs better in executing analytical selective queries [9].

By observing and studying PDBMS and MapReduce model features, the HadoopDB [2] team was motivated to implement a hybrid system of MapReduce technology and PDBMS to support analyzing structured Big Data on large-scale and shared-noting clusters. The HadoopDB system was designed to combine MapReduce and PDBMS technologies and benefits from the features of both techniques. From

the MapReduce model, HadoopDB gets scalability to go with thousands of nodes and fault-tolerance for node failures, and from PDBMS, HadoopDB gets high performance on structure selective queries.

HadoopDB Architecture

HadoopDB's basic design idea is to run multiple, separate, single databases in every storage node and coordinated them in a Hadoop communication layer. Shared-nothing PostgreSQL is used as an additional database storage layer and Hadoop as an coordination and communication layer. The system also provides an SQL-like interface translation layer utilizing Hive. The communication layer assigns jobs to particular nodes in the cluster after analyzing and monitoring node resource availability. Moreover, it controls cluster parallel distribution computing. The communication layer links with the additional storage database layer for executing queries and fetching results from a connector. The storage layer contains multiple nodes running independently on single-node database instances to provide systems with database properties like indexing and views.

HadoopDB modified the original Hadoop architecture [13] to combine PDBMS and MapReduce and added four main components: HadoopDB Catalog, Database Connector, Data Loader and SQL-MapReduce-SQL (SMS) Planner. The HadoopDB architecture is presented in Figure 2.5.

Database Connector Hadoop typically works with data stored in HDFS. To allow Hadoop to communicate with the new DB storage layer, *Database Connector* was developed to provide a HadoopDB with connection parameters. The Hadoop `InputFormat` library was modified to execute jobs in independent database instances instead of executing them on HDFS files. The results are retrieved and sent to MapReduce framework as KVP.

In general, connecting to the database required specifying the connection details such as JDBC driver and user authority information; consequently, any job running on a HadoopDB cluster must provide the *Database Connector* with those details through the *HadoopDB Catalog*.

HadoopDB Catalog contains database connection information such as JDBC driver, connection port,

username, and password and also includes details about number of data chunks and replications and their locations. This information is executed and stored on an XML file which is located in HDFS for the purpose of providing the MapReduce layer flexibility in accessing, managing and tracking data chunks across DB clusters.

Data Loader controls data partitioning and parallel loading across HadoopDB clusters. HadoopDB provides two levels of data partitioning: global and local. They were developed in MapReduce jobs with different hash functions that are not the same as the original hash-partitioning function in Hadoop. First, the Global Hasher partitions the data stored in HDFS by user defined attributes within the number of chunks equal to the cluster size in order to store them separately on nodes. Each local chunk is re-partitioned into smaller chunks by the Local Hasher. When the partitioning process is complete, each small chunk is bulk-loaded into an independent single-node database.

SMS Planner The HadoopDB SQL interface that converts SQL queries into MapReduce code then converts Mapreduce query outputs to SQL utilizing Hive[17]. This HadoopDB job could be written for either the MapReduce framework or the SMS planer that extends Hive and uses HiveQL for job creation simplicity. HadoopDB only uses HiveQL as a SQL-like language without running the Hive query execution plane. The system uses database logical query execution plane.

To further explain the execution plan, differences between a HadoopDB and Hive from the original HadoopDB are presented in Figure 2.6, we consider the example below:

```
SELECT YEAR(saleDate),SUM(revenue)

From sales

GROUP BY YEAR(saleDate);
```

Hive executes the query as follows: (I) The query is written in HiveQL. (II) It is represented as an abstract tree by the Parser. (III) The Semantic Analyser reads file meta information from the *MetaStore* in the Hive's internal catalogue and tells MapReduce framework file scanning Procedure,

InputFormat class, the required field extraction details, and the field's positions. (IV) The query plan generates a relational operator Directed Acyclic Graphs (DAG). (V) The logical plan converts to a physical plan as illustrated in Figure 2.6(a). (VI) The plan creates a map or reduce phases, as determined by the Hive optimizer. The optimizer involves the Reduce Sink operator that work as a repartition operator before executing join or GroupBY operators. In fact, in contrast to DBMS cost-based and adaptive optimizers, Hive optimizers are rule-based optimizers.

HadoopDB SMS planner executes the query as follows: (I) The query is written in HiveQL. (II) Hive *MetaStore* is updated by database table information that is stored in the HadoopDB catalog. (III) After planing the system's physical design and before running the query on MapReduce, SMS sets the partitioning keys depending on the required fields in the query. Then, instead of following the DAG relational operator, SMS applies cost-based SQL to modify the relational operators and assigns the logical process to the database layer. (IV) Each database instance processes the query independently and sends the results to the map phase. (V) When the table is split by the GroupBY attribute, YEAR(saleDate), the database instances process the whole query and send the results to the map phase as illustrate in Figure 2.6(b). There is no need for a reduce phase. When there is no partitioning specified during the loading of the table, the database layer executes partial aggregation with selected removal, the map phase operates the group-by and the reduce phase gathers all the partial aggregation for the final result.

In the join operation case, Hive's main task is to store files in HDFS separately where attributes in one file have no relation to attributes in other files. This is called *foreign key* in the database context. So, Hive execution plan is designed to scan each file individually and repartition the data according to the required field in the join clause to compute the join operation. In SMS, the join operation fully executed in the database layer when tables are partitioned by the join attribute. Otherwise, the join operation would likely fail [2].

Although SMS was tested and worked well for selection and aggregation queries on VLDB bench-

marks, running other queries correctly in SMS is not yet guaranteed [3]

In addition to HDFS, HadoopDB added PostgreSQL database storage layer to benefit to its properties and to speed-up the processing of complex tasks over structured data. It benefits from Hadoop fault-tolerance and scalability support. In general, HadoopDB is an analytical system not a transaction system that support single inserts, updates and deletes. Data is inserted to the system only through batch loading.

Figure 2.6: Hive and HadoopDB SMS execution Plan [2].

(a)Hive MapReduce job. (b)SMS MapReduce job without a partitioning assumption. (c)SMS MapReduce job with YEAR(saleDate) partition assumption.

PostgreSQL

PostgreSQL is an open source object-relational database system that was initially developed by the PostgreSQL Global Development Group in 1995. It is compatible with most operating systems like Linux, Microsoft Windows, Solaris, and Mac OS. To meet ANSI-SQL standards, PostgreSQL supports ACID-compliant, complex queries, migration, subqueries, functions, indexing and materialized views.

PostgreSQL is widely supported for indexing more than other databases. In addition to supporting the typical compound, unique and partial indices of B-trees, R-trees and hashes, it also provides Generalized Search Trees (GiST), generalized inverted indices (GIN), k-nearest neighbour (k-NN) indices, functional indices and user-defined index methods. Indexing methods can be written in many languages, such as Java, C/C++ , perl, and python or over PostgreSQL interfaces like JDBC for Java [23].

For a fully open-source system, HadoopDB utilizes single-node PostgreSQL as database instances on the storage layer. Furthermore, HadoopDB authors observed that PostgreSQL out-perform MySQL on the complex join queries since MySQL does not support clustered indices and its join algorithms are limited.

2.3 Hadoop Indexing

Hadoop is not a database and not all database features are appropriate to be added to the Hadoop framework; still Hadoop can learn from databases to improve its system performance. Hadoop is mainly designed for analytical tasks that require scanning all data and the system works efficiently with these types of queries but, at the same time, it can run on-line queries. Sequence scan is not the ideal model for low selectivity queries and may cause extra I/O overhead. In fact, several applications still demand both analytical and on-line queries. DBMS-built indices provide fast record lookups where the processor first accesses the indices that contain pointers to the requested records then fetches the referenced records from the data. The original design of Hadoop did not include a built in indexing technique,

thus many research are working on adding indexing support to the system to improve performance and avoid accessing data unrelated to the query. In general, there are four methods for simulating indexing technique in Hadoop: *customized data partitioning and replication*, *create and store index file in HDFS*, *utilizing a database index* and *utilizing high-level language indexing technique*.

customized data partitioning and replication The default Hadoop plan partitions data horizontally into 64MB blocks and replicates each block three times. Clustered primary indexing can be simulated by sorting and partitioning the data on the primary key then customizing the *InputFormat* class to prune the data blocks. Another way is to partition data by a particular attribute, store each part in a file and name it with a certain rule. These techniques would only support indexing by one attribute. However, one clustered index is not the ideal option for all queries; secondary indices are needed in many cases. HAIL (Hadoop Aggressive Indexing Library)[24] was developed to support more than one cluster index. Each block replica is sorted to support one cluster index. Specifically, the data first is divided into blocks, then each block is replicated several times. The default is three replication. Each replica is sorted by a particular attribute and contains metadata about the sorting information. The number of cluster indices is equal to the number of replicas. During execution, the replica is chosen according to the query condition attributes. The limitation of this method is that the number of indices is limited and has to be built in advance, before even knowing the most frequent attribute queries.

Create index file in HDFS The Hadoop pipeline contains many UDFs that could be customized to simulate indexing. [25] inject a trojan index for each block in HDFS by customizes Hadoop UDFs such as *group()* and *shuffle()* and create index for each block in HDFS. When a query executes, the index file points to the related records. A similar approach that fetches the index file as an input to the MapReduce framework instead of the original file required developing a new reader to read the file index first and then only process blocks with related records.

Utilizing a database storage layer indexing technique In the case of MapReduce and DBMS hybrid systems, the queries are distributed by MapReduce into single database nodes where they are executed and the results are returned. MapReduce gathers the results from all nodes and completes aggregation and any global operations. The data is stored in the database layer, so the index is built in the same layer. During query execution in the database layer, the query utilizes the available indices. Queries run under the DBMS umbrella, follow the optimizer's rules and benefit from its features. The database index only speeds up the part of the query is executed in the database layer, not the part executed by the MapReduce layer.

Utilizing high-level language indexing techniques Complex analytical queries can either be expressed directly in MapReduce code or through high-level languages like Hive or Pig. Some applications rely on those languages' indexing techniques which could improve the queries that run over that layer but not overall queries. Indexing techniques in high-level languages do not work at the system level, so extra work is required. For example, when the query contains a join operation, there is no guarantee that the HDFS files in the query are matching the attributes. Thus join operation on those languages is more expensive than join operation on a database. In addition, their indexing technique is new and not all types of indices are supported.

2.4 DBMS Indexing

Indexing is one of the main physical database structures for speeding up data retrieval. Improved performance can be achieved by employing indices in logical execution plan of queries, avoiding scanning whole data and reading records unrelated to the query. It must be remembered that, although indices can improve performance, creating many indices randomly could have a negative impact and result in waste of space. Designing system index configuration is complex and not a straightforward task that can be administrated totally by humans. Below are some difficulties of indexing creation .

Number of Possible indices The amount of possible indices increases with datasets number and the number of attributes in each dataset. In fact, indices could be structured for different environments such as: (I) for single or multiple attributes. (II) clustered or non-clustered data. (III) hash functions or B-tree, among others. This leads to a potentially large number of indices, which would require a large amount of space and time to build the index.

Dataset Properties Dataset size, content type, and its relation with other datasets affect the index configuration design. Indices may not be useful for small datasets when scanning whole dataset is more cost effective than first reading the index then fetching the relevant records. In addition, dataset contents determine the index type. For example, B-trees are suitable for number ranges, while foreign keys are required when tables have relations between each other.

Index Interaction Indices do not work in isolation from others; they are influenced by other index configurations. In other words, not only individual index should be selected properly, but also the set of indices must be chosen carefully [26].

Operational Parameter Each new release of a DBMS could have new or modified operational parameters which makes building an optimal index different from one release to the next.

Index Creation Schedule The order of attributes for which an index is created impacts the index configuration on the query execution plan [26].

Changeable Workload Indices could be created for a particular workload for performance improvement. However, the same index configuration may not benefit or may negatively impact the performance when the workload or the dataset is changed. Thus, performance should be analyzed for changes in workload.

2.5 Index Selection

In the remainder of this chapter, some of the exiting work related to my proposed index selection tool is presented. First, work on RDBMS index selection are discussed. Then, some of the research on parallel Apriori algorithms developed on the MapReduce framework are presented. Finally, MapReduce-based system optimization works are presented.

Index selection tools are developed for database systems to build the physical design effectively and to improve system performance. The tools help the database administrator easily and efficiently manage physical design challenges generally, including index selection. Index selection is not a new idea and has been studied in database environments since the late 1980's by Finkelstein *et al.*[27]. In commercial applications, Microsoft developed indices and materialized view selection tools for Microsoft SQL Server 2000 [28], which was generalized later to provide a database structure modification for query execution performance in SQL Server 2005, called the Database Engine Tuning Advisor (DTA). In fact, several index selection solutions were developed focusing on various contexts such as online or off-line; automated, semi-automated or fully automated; optimized index maintenance cost or query response time; index set configuration selection approach and algorithm. There is a very wide variety of research in database index selection, thus we will focus on the work most closely related to the proposed tool. Most database index selection approaches are based on the database optimizer content which is not available in MapReduce-base systems. In [29], Aouiche and Darmont applied data mining-based optimization approach for index and view selections. Their approach extracts information from the query workload without utilizing a database optimizer. For index selection, they supply one Apriori algorithm for frequent itemsets to the database query workload in order to extract the frequently indexed attributes and recommend an ideal index configuration. Our proposed index selection tool applies the same approach to HadoopDB where the Apriori algorithm is implemented and the query workload is collected in the MapReduce layer instead of having the database gather information from all storage

nodes. In fact, HadoopDB does not have a component to manage the optimizer in the storage layer which restricts developing an advanced index selection tool in that environment.

Running the Apriori algorithm on massive datasets requires very expensive and complex computing infrastructure. It needs to scan massive data K times to generate candidate of K -itemset, Apriori algorithm definition and parameters are described in details in next chapter. A single processor's memory and CPU resources could not handle that workload. Thus parallel and distributed technology is used for accelerating Apriori algorithm performance. Recently, [30, 31, 32, 33] developed Apriori-based algorithms in Hadoop- MapReduce which benefit from frameworks parallel and distributed computing, load balancing and resource monitoring. In [30, 32, 33], authors compute each K -itemset candidate from level one to level K in separate MapReduce phases where level one contains list of single attributes and level K contains list of set of K attributes. Authors of [30, 32] implemented two MapReduce jobs, one for 1-itemsets and the other for K -itemsets whenever $K > 1$. Work in [33] provides three MapReduce jobs, Single Pass Counting (SPC), Fixed Passes Combined-counting (FPC), and Dynamic Pass Counting (DPC). SPC computes K -itemsets and feeds the results to the next phase to compute $(K + 1)$ -itemsets and so on. FPC computes 1-itemsets and 2-itemsets as SPC. After that, item set candidates of user defined fixed number are combined in one phase. For example when the fixed number equal to three C_K, C_{K+1} and C_{K+2} are computed in the same MapReduce job to find the frequent itemset of all three level on one step where C_K represents candidate index of level K . DPC addresses FPC issues overloading candidates when the fixed number is small and the candidate number is too large. DPC provides dynamic candidate combinations instead of fixed ones depending on the analysis of the workload to balance between number of phases and the number of candidates in the workload.

The K -phases approach focuses on benefiting from parallel computing and reduces Apriori algorithm costs. However, it does not necessarily reduce dataset scanning time. Candidate item sets are generated level by level. Li and Zhang [31] proposed a one phase MapReduce Apriori algorithm to compute all candidate itemsets from level 1 to level K in one MapReduce job. The one phase approach scans the

dataset just one time, Map phase reads each transaction once, extracts the items and generates all candidate itemsets. The candidate itemset frequencies are counted in the reduce phase. Their aim is to estimate Apriori algorithm performance on different MapReduce cluster sizes. Our proposed tool utilizes the same approach but for a very different goal. We use one phase MapReduce Apriori algorithm for gathering the most frequent item set results then the proposed index-selection tool recommends the optimal index configuration to the storage layer in HadoopDB. Lastly, we compared query execution times in different configuration settings that are generated by the algorithm.

Although MapReduce provides simple parallel computing, it leaves the optimization tasks to the programmer. The MapReduce model does not have a built in optimizer for logical MapReduce job execution or for physical data organization. The original MapReduce model lacks the main physical components like view and index, some work has emerged to support those components as discussed in the Hadoop indexing section. In fact, the optimization technique does not only rely on the system MapReduce communication layer, but also on the system underlying the storage layer and its physical components. Optimization technique that works for one MapReduce-based system is not necessarily working with other systems when they use different storage layer. Authors of [34, 35] developed a job execution optimizer on the original MapReduce model and HDFS to automatically select the ideal MapReduce logical execution plan not manage HDFS physical design. The implementation in [36] has the same goal for optimizing MapReduce logical execution but over Hive interface-level. The optimizer design in [34, 35, 36] chooses the appropriate index among several existing indices in HDFS for a MapReduce logical execution plan. They aim to choose indices not to build appropriate index configuration for the system. [25] Developed an index selection technique on Hadoop with its default HDFS storage layer. It creates the appropriate index for Hadoop system during data loading the HDFS according to the data characteristics and the user defined splitting algorithm. The index is built in advance and before gathering any workload information. Our index selection tool runs over a hybrid system that use database as storage layer instead of HDFS, thus we build the index differently. In addition, we select

index configuration on demand after analyzing the system workload instead of selecting the index in advance.

Chapter 3

Our Index Selection Tool

This chapter contains two sections: a frequent itemset mining tool and the proposed index-selection tool. The first section describes a frequent itemset and the Apriori algorithm that is used in the proposed tool's core design. The index-selection tool section presents the proposed tool's objective and its design details.

3.1 Frequent Itemset Mining

Frequent itemset mining is a basic method and first stage of association rule mining that was initially proposed in 1993 [37]. It is a well-known method in knowledge discovery research and an essential concept for many research areas such as business, network security, artificial intelligence and database index tuning where item relationship information must be extracted from large transaction logs. Market Basket Analysis is a well-known frequent itemset mining example, where the method detects the usual items bought together at supermarkets. For example, {milk, bread} is a frequent itemset, if milk and bread are bought frequently together. Such information could help managers organize products on shelves more effectively.

Frequent itemset mining algorithms find all frequent itemsets in a practical transaction database. Generally, the transaction log is scanned to count the itemset frequency, then the itemset *support*, the ratio of transactions containing the itemset is calculated and finally the *support* is compared to a *minimum support* threshold defined by the user. Wherever an itemset meets the *minimum support* threshold, it is set as a frequent itemset. Several algorithms have been developed to generate frequent itemsets for association rule mining such as Apriori and FP-Growth. They find frequent itemsets with various computing details but they follow the same general concept and definitions.

Frequent Itemset General Definition:

- **Database D** : Set of transactions on the transaction log. $D = t_1, t_2, t_3, \dots, t_m$ where m is the number of transactions in *database* D as presented in Table 3.1.
- **I** : Set of different items appearing in D where $I = i_1, i_2, i_3, \dots, i_n$ and n is the number of items in I .
- **Itemset** : A subset collection of items in I appearing together.
- **k-itemset** : An itemset of size K .
- **Support Count $\sigma(X)$** : Frequency of appearances of Itemset X in D .
- **$Supp(X)$** : Support of an itemset X . It is the fraction of all transactions that contain itemset X , so $Supp(X) = \sigma(X)/m$
- **$minSup$ ϵ** : Minimum support threshold determined by the user.
- **Frequent itemset** : An itemset that has $Supp$ greater than or equal to a $minSupp$.
- **L_k** : All frequent itemsets for level K
- **C_k** : Candidate itemsets for level K

Table 3.1: Example Transactions Database

TransactionID	Items
T1	a,b,d
T2	a
T3	a,b,c,b
T4	a,b,c
T5	a,c

3.1.1 Apriori Algorithm

Apriori algorithm [37] is one of the best known solutions for finding frequent itemsets for association rule mining. The algorithm adopts a “bottom up” approach that counts frequent itemsets of low level and uses it to generate other levels one by one. In the initial pass, *databaseD* is scanned to extract all individual itemset candidates C_1 and calculate their frequencies. Then *Supp* is computed for each of them. Finally, the *Supp* is compared with the defined *minSup*. The item is added to L_1 if its *Supp* is greater than or equal to the *minSup*. Other levels are generated from the initial pass. For example, the C_2 candidate item sets are generated by first scanning L_1 . Only those item set pairs where both items are in L_1 are stored in C_2 . If either item does not meet the frequency threshold, the item is ignored. *database D* is scanned again to count the frequency of itemsets in C_2 . They are stored in L_2 if they meet the *minSup* threshold. The same steps are repeated level by level to level L_K or until no candidate set exists in *database D*. The point of adopting a “bottom up” approach in the Apriori algorithm is to avoid computing the frequency of an itemset if any of its sub-items are not frequent. This is because the frequency of an itemset is always greater than or equal to the frequency of its individual sub-items. If $Supp(i) < minSup$, then $Supp\{i, j\} \leq minSup$. Similarly, for itemsets J and X , if $J \subseteq X$ then $Supp(J) \geq Supp(X)$. The Apriori algorithm pseudo code is presented in Algorithm 1.

Algorithm 1 Apriori algorithm (D, ϵ)

Pass 1: $C_1 \leftarrow$ Generate the candidate itemsets**For Each** itemset $i \in C_1$ Count i frequency from D Count $Supp(i)$ **IF** $Supp(i) \geq \epsilon$ $L_1 \leftarrow i$ **End For****Pass K:** $K \leftarrow 2$ $C_K \leftarrow$ Generate the candidate itemsets from (L_{K-1}) **While** $C_K \neq \emptyset$ **For Each** itemset $i \in C_K$ Count X frequency from D Count $Supp(i)$ **IF** $Supp(i) \geq \epsilon$ $L_K \leftarrow i$ **End For****End While****return** $\bigcup_i L_K$

Although, the Apriori algorithm has significant history in association rule mining, it suffers from a number of drawbacks. The algorithm generates frequent itemset for one level and saves it in memory for generating the next one. This process demands high memory. Furthermore, the number of candidates grows rapidly as the number of items in I increases. Computing frequent itemsets becomes more and

Figure 3.1: Apriori Algorithm Example.

more complex and expensive for large K , since the number of candidates grows exponentially. In addition to the huge amount of computing that the algorithm needs, it consumes significant I/O for each itemset level during the load of the previous level frequent itemset phase.

Even high quality processors cannot handle the algorithm's work load when it runs over large D . Distributing data and computing is an axiomatic solution for accelerating the algorithm performance. Subsequently, several algorithms were developed for parallel Apriori algorithm such as three algorithms implemented and tested on IBM research center for parallel mining area [38]. Efficient and scalable parallel systems have notable impact on the algorithm's performance. Along with the rapid growth of data in applications today, many algorithms including the Apriori algorithm are developed on the MapReduce framework because it provides an efficient computing and load balancing parallel processing mechanism in low cost machines for large datasets as discussed in Section 2.2.2.

Figure 3.2: Index Selection Flow Chart.

3.2 Index-Selection Tool

Indexing techniques are essential to physical aspect of data storage engine design to speed data retrieval, but they are also complex. Thus several tools have been developed for database environments to help data administrators design index configurations efficiently. The need for index-selection tools in Big Data systems like MapReduce and DBMS hybrid systems is more urgent and more complex due to the data size. Nevertheless, to the best of our knowledge, there is no research on hybrid system physical design optimization to date. This is because hybrid systems are still new systems that many companies recently started to adopt and they are changing very frequently. In the case of open-source projects like HadoopDB, support is lacking and not all components are fully functional. In addition, the lack of optimizers for Mapreduce-based systems restricts implementation of an advanced index-selection tool, since most such tools rely on a database optimizer contents.

3.2.1 Objectives

The aim of this work is to develop an index-selection tool for the HadoopDB system. The tool recommends the optimal index configuration to help the administrator manage the physical design of complex systems efficiently. The tool collects system workload and applies frequent itemset mining methods for index selection in the MapReduce layer then builds the ideal index configuration on the database layer.

Query performance in hybrid systems is influenced by the physical and logical design of the MapReduce and database layers. In this thesis, I focus on building index-selection tools to improve the physical design of database layer, but not the physical design of the MapReduce or the logical execution design of any of the layers. The tool is injected on the HadoopDB system, but it could easily be modified to work with any MapReduce/DBMS hybrid systems or be injected on any new releases and benefit from the system's development.

The tool is designed to improve analytical task performance, not update and delete transactions since

the MapReduce framework was mainly created to support analytical tasks.

3.2.2 Design

The proposed tool is implemented in the MapReduce environment and is injected into the HadoopDB system. It collects query indexable attributes from tasks running over the MapReduce layer and stores them on HDFS as a *query database*. Then the *query database* is analysed to extract execution details such as number of queries and number of indexable attributes. Depending on the indexable attribute set frequencies in the *query database*, optimal index-set configurations are selected. Finally, the optimal index-sets are built on the database layer.

MapReduce divides the problem into small chunks and assigns them to different nodes in a cluster for parallel processing. The advantage of this model is that the distribution and networking management details are hidden from the programmer since applications write only on map and reduce functions. However, this simplicity can also introduce challenges as data input and output must be convertible to the KVP data representation. This requires understanding the problem and data model and sometimes the need to implement a new reader. Another drawback is having to simulate recursive functions that are needed for some algorithms like permutation. Map function output could not be an input for the same map function and also reduce function's output could not be an input for the same reduce function. Developing such functions on the map reduce model requires additional effort. In addition, data cannot be exchanged between map and reduce functions until all maps and reductions in the job are complete. The job output then could get inputs from HDFS or the job configuration data.

The tool processes on three stages: *Collector*, *Analyzer* and *Recommender* as shown in Figure 3.2 and described in the following.

Collector

The collector extracts all indexable attributes from system-executed tasks. Each individual query-indexable attribute I_q is stored on a separate line in the *query database* D , so queries are identified by offset. I is the set of all different indexable attributes in D . Since the same item may be written in different letter case and the MapReduce framework is case sensitive, all attributes are converted to lower case to avoid duplication. In addition, attributes may appear in a query in different orders each time. Frequent itemset algorithm generates the itemsets that are independent of order. For this reason, the attributes are sorted to avoid generating the same index-set in different order. Again, the collector gets the I_q , candidate index for each system-executed task. Then each tasks' candidate index is added to the *query database* D . The database is stored on HDFS for future processing by the Analyzer and the Recommender.

Analyzer

The Analyzer extract information from the *query database* and feeds it to the Recommender to run the frequent itemset algorithm. The Analyser counts the number of transactions(queries) $\sigma(q)$ and the number of different items (attributes) $\sigma(i)$ in the *query database*. When the job is complete, more information about the *query database* can be gathered either from the output stored on the HDFS or from the job configuration MapReduce Counter. To avoid I/O traffic, the Recommender gets the information from the job configuration. MapReduce Job Counter information is only available when the job is complete and for jobs running on the same task. The Analyzer job `InputFormat` class is `Textinputformat` where dataset is read line-by-line and each line's processed by a different mapper. The `InputFormat` process is described in detail in Subsection 2.2.2. In practice, the number of queries $\sigma(q)$ is the number of record input into the map function that specifically exist on "`MAP_INPUT_RECORDS`" from "`org.apache.hadoop.mapred.Task$Counter`". The number of attributes $\sigma(i)$ is the number of groups entered in the reduce function that specifically exist on "`REDUCE_INPUT_GROUPS`" from

"org.apache.hadoop.mapred.Task\$Counter". Each attribute represents a group in the reduce phase since the attributes were tokenized by the map function and emitted as a key. In other words, each individual key sent from the map phase is processed as a group in the reduce phase.

Recommender

The Recommender is the core of the index-selection tool and is executed after the Collector generates *query database* and the Analyzer extracts the *query database* details. First, the mapper reads the *query database* in `Textinputformat` where each record contains I_q for a particular query and is processed by a different mapper. The data then proceeds to the mapper as KVP where the key is the query offset in the *query database* and the value is the set of indexable attributes I_q . The mapper counts the number of different indexable attributes in the query $\sigma(i_q)$ through tokenized values. $\sigma(i_q)$ is the maximum frequent itemset candidate size. Then all index-sets (candidates) C_q in the query are generated from level-1 to level- $\sigma(i_q)$. The mapper sets key to index-sets $c \in C_q$ and set the value to one, where one means existing in that particular query regardless of how many times the attribute or attribute set appears in the same query. This is the rule in the Apriori algorithm: the item occupancy count is one per transaction even if it appears several times. All mappers sending the keys and values to the reduce phase. The Reducer groups the values by the key which is index-sets c . Then it sums the values of that key to find the index-sets frequency on the *query database*. Next, compute the index-sets $Supp(c)$ and compares it with the $minSupp$ threshold. If $Supp(c)$ is greater than or equal to $minSupp$ then the index-set c is a frequent itemset, and is saved as an optimal index-set configuration in L . Finally, the optimal configuration is built in each single database node in the storage layer. The Recommender pseudo-code is shown in Algorithm 2.

Algorithm 2 Recommender($\sigma(q), \epsilon$)

Mapper:**for** all $q \in D$ $\sigma(i_q) \leftarrow \text{tokenizeValue.count}$ $C_q \leftarrow \{c \mid c \in \bigcup_{x=[1-\sigma(i_q)]} C_x \wedge c \subseteq q\}$

// Generate all candidate in q from level 1 to K

end for

for all candidate $c \in C_q$ appendOutput($c, 1$)**Reducer:**receive($c, \text{valueList}$) $c.\text{frequency} = 0$ **for** all $\text{value} \in \text{valueList}$ $c.\text{frequency} = + \text{value}$ $\text{Supp}(c) = (c.\text{frequency} \div \sigma(q))$ **if** $\text{Supp}(c) \geq \epsilon$ appendOutput($c, \text{Supp}(c)$)

Chapter 4

Experimental Results

In this chapter, I have described the testing environment used, and the results of the performance evaluation for the proposed index-selection tool used on a HadoopDB system. The objective of the experiments performed were to assess performance improvement in different index configurations provided by the proposed tool in a HadoopDB system compared to the performance in index-less configuration. Experiments show that the tasks response time with proposed tool index configuration greatly outperforms the ones using a brute-force approach.

4.1 Experimental Environment

The experiments were conducted on a HadoopDB cluster with two nodes connected through Ethernet. Each node was equipped with 12 GB memory, 2.67GHz core processors and 250 GB storage. The operating system was 64-bit Linux Ubuntu v14.04. One node acted as a master and both nodes were used as slaves. All experiments were conducted on the HadoopDB open source system [3] that has Hadoop as the communication layer and PostgreSQL database as the storage engine. Hadoop v0.19.1 was installed on each node in the cluster and PostgreSQL version 9.3 database single node was installed

on each slave. The Java Virtual Machine (JVM) version used was 1.6.0_30 and JDBC version used was PostgreSQL-9.3 – 1101.jdbc3.

The performance of the proposed tool is evaluated using the web analytic benchmark tasks tool that is included in a HadoopDB system, and which was originally developed in [9]. The benchmark has four analytical tasks: *Aggregation Task*, *Join Task*, *Selection Task* and *UDF Aggregation Task*. The last task is not used in our experiments, because it does not contain a condition expression syntax. So, there is no candidate index set for this task. The rest of the tasks are described in the Analytical Tasks subsection 4.1.3. The tasks run over the MapReduce framework and executed on two dataset: Rankings and UserVisits. The web analytic tasks do not have lots of attributes in their condition expression syntax. Consequently, the number of index candidates in the workload are limited. With few items (index candidate) to be selected from, the generated frequent itemset are not changed with large *minsupp* values. For example, if a given itemset $\{i, j\}$ was based on *minsupp*= 30% and changing *minsupp* to 40% and 50% results in no more itemsets, then the frequent itemset will not change with the change in *minsupp*. In other words, itemsets are stable for different *minsupp* percentages, when there is no more index to be selected from the workload. The web analytic tool is the only benchmarking tool suitable for the required testing the tool. However, and to the best of our knowledge, there is no standard decision-support benchmark for Big Data DBMS as of yet. For example the Transaction Processing Performance Council (TPC) has the TPC-R decision-support benchmark standard that are typically used in DBMS index tuning research, but the Big Data standard benchmarking (TPC-BD) tool is currently in the development stage [39].

In order to evaluate the tool performance, I have observed tasks execution time in two stages. First, with index-less configuration where tasks are executed without building any index. Then, I executed the same tasks with various index configurations that used different user-specified *minsupp* values. The response times in seconds are the average recorded time and the experiments which were repeated 10 times .

4.1.1 HadoopDB Settings and Configuration

The experiments ran on a HadoopDB system, and I did use the default, suggested system setting based on their website recommendations.

Hadoop Setting HadoopDB system mainly utilizes Hadoop open source software model and applies some modification in the default configuration setting to meet the system requirements. Hadoop setting for HadoopDB are presented in Table 4.1.

DBMS (PostgreSQL) Setting PostgreSQL is installed on every slave storage node. Some of the default settings were changed to satisfy the HadoopDB system requirements. The shared buffer was set to 512MB and the working memory size was increased to 1GB

Table 4.1: HadoopSetting

variable	value
hadoopdb.config.file	HadoopDB.xml
hadoopdb.fetch.size	1000
hadoopdb.config.replication	false
mapred.child.java.opts	-Xmx1024m
mapred.job.reuse.jvm.num.tasks	-1
mapred.compress.map.output	true
io.sort.factor	100
io.sort.mb	200
io.file.buffer.size	131072
mapred.reduce.parallel.copies	50
tasktracker.http.threads	50
mapred.tasktracker.map.tasks.max	1

4.1.2 DataSets

Each node in the cluster contained twenty chunks distributed in different PostgreSQL databases and each chunk contained Rankings and UserVisits tables. First, the dataset was uploaded to HDFS as text where each line represented a record and the attributes in the record were separated by a vertical line, '|'. Next, the HadoopDB Global Hasher partitioned dataset in parallel into all nodes in the cluster. In our cluster, the Global Hasher partitioned the datasets into two chunks, one for each slave node. Then, the Local Hasher partitioned each chunk in each node into twenty small chunks so, the whole cluster contained forty chunks. The chunk size was 1GB for the UserVisits dataset and around 51MB for the Rankings dataset. The final step was to load the dataset into the database layer. Small chunks were bulk-loaded into a single PostgreSQL node using the standard SQL COPY command. The datasets, developed in our work, followed the description given the ordinal benchmark [9]. Data size details are presented in table 4.2. In addition, a workload was created from the analytical tasks and it contained around 36K transactions (tasks), taking around 1MB storage in our HDFS.

```
CREATE TABLE UserVisits (
    sourceIP VARCHAR(16),
    destURL VARCHAR(100),
    visitDate DATE,
    adRevenue FLOAT,
    userAgent VARCHAR(64),
    countryCode VARCHAR(3),
    languageCode VARCHAR(6),
    searchWord VARCHAR(32),
    duration INT);
```

```
CREATE TABLE Rankings(
```

Dataset	Data Size Per Node	Number of Record Per Node
Rankings	1 G	21,600,000
UserVisits	20 G	172,800,000

Table 4.2: *Datasets Details*

```

pageRank INT,

pageURL varchar(100),

avgDuration INT);

```

4.1.3 Analytical Tasks

As it was mentioned earlier, the proposed tool executes three web analytical tasks: *Aggregation*, *Join* and *Selection*. The Aggregation task has two subtasks: a large Aggregation task and a small Aggregation task, and they are treated as two separate tasks in our experiment. So, we had considered four tasks in total in our tests.

The tasks response time examined two possible settings: index-less and index-based. In the index-less configuration, the tasks executed were a sequential scan, that did not involve creating any indices on any tables in PostgreSQL instances. Then, the same tasks were executed several times for the index-based setting with various index configurations, based on the user selected *minsupp* value for the index-selection tool. Each index configuration was built on all single PostgreSQL nodes and ran the ANALYSE statements¹.

Overall the index configurations recommended by the tool achieved an average performance gain of up to 48% in all analytical tasks performed, compared to the index-less configuration. These results are presented in Figure 4.5. With high *minsupp* threshold, there is no performance distinction between index-less and index-based setting. This was a predictable result, since there is no index candidate to be

¹The use of the ANALYSE statement to help the query planner benefit from the created indices was a suggestion given to us by the authors of [2]

chosen for high threshold values. For example, to have index candidates with $minsupp=90\%$, the index candidates must appear nine times when the workload has ten transactions, which is not a typical case. Figures 4.1, 4.2, 4.3, and 4.4 and tables 4.3, 4.4, 4.5, and 4.6 display the results of executing large aggregation task, small aggregation task, join task and selection task, respectively. As it was explained earlier, the performance gain does flatten out, due to the limited number of index candidate available in our datasets. We also observed fluctuation in the performance for the large aggregation task and the selection task. These were in part due to the fact that part of tasks operations take place on database nodes, whereas other operations such as aggregation and grouping are finalized on the MapReduce layer. However, optimizing the MapReduce design that can address this issue is outside the scope of this thesis, and our proposed index recommendation tool works on physical design of the database physical design and not the control of database logical execution plans or the MapReduce layers.

Figure 4.6 summarizes the results of all tasks, and it shows that although join task is very expensive operation, it performs well with index-based configuration. Aggregation tasks do not exhibit huge improvements, since GROUP BY clause does not benefit from indices significantly.

Task 1: Large Aggregation Task

The task ran on the UserVisits table to compute the overall adRevenue for each individual sourceIP and the results was grouped by the sourceIP. It involved all nodes id which were necessary for grouping in the aggregation task. Below is the corresponding SQL commands executed in the HadoopDB.

```
SELECT sourceIP, SUM(adRevenue)
```

```
FROM UserVisits
```

```
GROUP BY sourceIP;
```

Task 2 : Small Aggregation Task

Similar to the large aggregation task, the task ran on the UserVisits table to compute the overall adRevenue for each individual sourceIP, but it only grouped the first seven-character of the sourceIP. Specifying grouping characters decreases the number of unique groups on the query. For this reason, small aggregation task runs faster than the large one. Below is the corresponding SQL commands executed in the HadoopDB.

```
SELECT SUBSTR(sourceIP, 1, 7), SUM(adRevenue)
FROM UserVisits
GROUP BY SUBSTR(sourceIP, 1, 7);
```

Task 3 : Join Task

This task computed the pageRank average and the adRevenue sum of the set of sourceIP during specific days. Below is the corresponding SQL commands executed in HadoopDB.

```
SELECT INTO Temp sourceIP,
AVG(pageRank) as avgPageRank,
SUM(adRevenue) as totalRevenue
FROM Rankings AS R, UserVisits AS UV
WHERE R.pageURL = UV.destURL
AND UV.visitDate BETWEEN Date('xxx - xx - xx')
AND Date('xxx - xx - xx')
GROUP BY UV.sourceIP;
```

Task 4 : Selection Task

Last task is a the selection task over the rankings table condition set by the pageRank. Below is the corresponding SQL commands executed in the HadoopDB.


```

SELECT pageURL, pageRank
FROM Rankings
WHERE pageRank > X;

```

Figure 4.1: Large Aggregation Task.

Trial	Index-based											Index-less
	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%	
1	306.5	306.5	267.4	267.4	267.4	267.4	296.8	296.8	296.8	301.7	301.7	301.7
2	302.9	302.9	262.8	262.8	262.8	262.8	280.7	280.7	280.7	313.8	313.8	313.8
3	279.6	279.6	255.6	255.6	255.6	255.6	275.7	275.7	275.7	310.7	310.7	310.7
4	276.6	276.6	250.6	250.6	250.6	250.6	273.8	273.8	273.8	292.6	292.6	292.6
5	268.8	268.8	267.6	267.6	267.6	267.6	278.7	278.7	278.7	273.6	273.6	273.6
6	275.6	275.6	266.7	266.7	266.7	266.7	268.7	268.7	268.7	267.6	267.6	267.6
7	268.6	268.6	278.6	278.6	278.6	278.6	282.7	282.7	282.7	269.6	269.6	269.6
8	270.6	270.6	253.7	253.7	253.7	253.7	272.7	272.7	272.7	270.7	270.7	270.7
9	267.6	267.6	256.7	256.7	256.7	256.7	270.7	270.7	270.7	266.6	266.6	266.6
10	267.6	267.6	262.7	262.7	262.7	262.7	275.7	275.7	275.7	266.6	266.6	266.6
Mean	278.4	278.4	262.3	262.3	262.3	262.3	277.6	277.6	277.6	283.3	283.3	283.3
SD	14.5	14.5	8.4	8.4	8.4	8.4	8.0	8.0	8.0	19.3	19.3	19.3

Table 4.3: Large Aggregation Task Response Time in seconds with and without index.

Figure 4.2: Small Aggregation Task.

Trial	Index-based											Index-less
	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%	
1	269.5	269.5	260.5	260.5	260.5	260.5	274.7	274.7	274.7	276.6	276.6	276.6
2	262.5	262.5	265.5	265.5	265.5	265.5	273.6	273.6	273.6	274.5	274.5	274.5
3	265.5	265.5	267.5	267.5	267.5	267.5	270.6	270.6	270.6	271.5	271.5	271.5
4	265.4	265.4	264.5	264.5	264.5	264.5	275.7	275.7	275.7	277.5	277.5	277.5
5	266.4	266.4	263.4	263.4	263.4	263.4	278.7	278.7	278.7	277.7	277.7	277.7
6	264.6	264.6	274.7	274.7	274.7	274.7	276.7	276.7	276.7	278.7	278.7	278.7
7	268.6	268.6	279.6	279.6	279.6	279.6	274.6	274.6	274.6	271.5	271.5	271.5
8	262.4	262.4	269.6	269.6	269.6	269.6	274.7	274.7	274.7	277.6	277.6	277.6
9	264.5	264.5	269.5	269.5	269.5	269.5	276.7	276.7	276.7	270.6	270.6	270.6
10	264.4	264.4	262.5	262.5	262.5	262.5	272.6	272.6	272.6	273.6	273.6	273.6
Mean	265.4	265.4	267.7	267.7	267.7	267.7	274.9	274.9	274.9	275	275	275
SD	2.3	2.3	5.9	5.9	5.9	5.9	2.3	2.3	2.3	3.0	3.0	3.0

Table 4.4: Small Aggregation Task Response Time in seconds with and without index.

Figure 4.3: Join Task.

Trial	Index-based											Index-less
	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%	
1	128.1	128.1	125.3	125.3	125.3	125.3	301.8	301.8	301.8	752.9	752.9	752.9
2	108.9	108.9	125.2	125.2	125.2	125.2	299.8	299.8	299.8	768.1	768.1	768.1
3	96.9	96.9	126.1	126.1	126.1	126.1	297.7	297.7	297.7	764.8	764.8	764.8
4	97.9	97.9	100.4	100.4	100.4	100.4	298.8	298.8	298.8	762.8	762.8	762.8
5	103.9	103.9	100.9	100.9	100.9	100.9	298.8	298.8	298.8	759.8	759.8	759.8
6	98.9	98.9	98.9	98.9	98.9	98.9	297.7	297.7	297.7	758.7	758.7	758.7
7	103.9	103.9	106.9	106.9	106.9	106.9	300.8	300.8	300.8	756.8	756.8	756.8
8	107.9	107.9	100.9	100.9	100.9	100.9	299.8	299.8	299.8	762.8	762.8	762.8
9	97.9	97.9	99.9	99.9	99.9	99.9	295.9	295.9	295.9	754.8	754.8	754.8
10	103	103	96.9	96.9	96.9	96.9	299.8	299.8	299.8	757.7	757.7	757.7
Mean	104.7	104.7	108.1	108.1	108.1	108.1	299.1	299.1	299.1	759.9	759.9	759.9
SD	9.2	9.2	12.3	12.3	12.3	12.3	1.7	1.7	1.7	4.7	4.7	4.7

Table 4.5: Join Task Response Time in seconds with and without index.

Figure 4.4: Selection Task.

Trial	Index-based											Index-less
	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%	
1	50.3	50.3	65.1	65.1	65.1	65.1	56	56	56	59	59	59
2	49.8	49.8	53.8	53.8	53.8	53.8	52.8	52.8	52.8	53.8	53.8	53.8
3	50.7	50.7	52.2	52.2	52.2	52.2	53.9	53.9	53.9	53.8	53.8	53.8
4	49.7	49.7	51.8	51.8	51.8	51.8	48.8	48.8	48.8	60	60	60
5	49.8	49.8	53.8	53.8	53.8	53.8	52.8	52.8	52.8	52.8	52.8	52.8
6	48.8	48.8	53.4	53.4	53.4	53.4	52.8	52.8	52.8	58.1	58.1	58.1
7	49.8	49.8	51.8	51.8	51.8	51.8	50.8	50.8	50.8	52.8	52.8	52.8
8	48.8	48.8	51.8	51.8	51.8	51.8	49.9	49.9	49.9	52.8	52.8	52.8
9	48.9	48.9	52	52	52	52	50.8	50.8	50.8	52.1	52.1	52.1
10	49.8	49.8	54.2	54.2	54.2	54.2	49.8	49.8	49.8	51.8	51.8	51.8
Mean	49.6	49.6	54	54	54	54	51.8	51.8	51.8	54.7	54.7	54.7
SD	0.6	0.6	4.0	4.0	4.0	4.0	2.2	2.2	2.2	3.1	3.1	3.1

Table 4.6: Selection Task Response Time in seconds with and without index.

Figure 4.5: Tasks Response Time on Different Index Configurations.

Figure 4.6: Compare Tasks Response Time on Index-based and Index-less configuration.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

The new large-scale systems bring many benefits for processing and analyzing Big Data that traditional databases does not have. However, these systems are still relatively new comparing their DBMS counterparts. The new systems could learn many concepts from DBMS such as indexing and the use of views for physical design and query execution for logical design. Similarly, the large-scale systems that rely on the database as a storage layer, such as HadoopDB, could learn from DBMS since they only utilize the storage engine, and do not adopt all of the whole system's features and they just utilize the storage engine. For example, they provide indexing and views, but not query optimizers.

An index is one of the most important components in a database that can speed-up retrieval of data and avoid accessing irrelevant records. However, not all indices can a system. Some just reserve a space and are not accessed by any queries. Thus, choosing an optimal index configuration could improve system performance significantly. Hence, index selection has received a great deal of attention in database design and research.

HadoopDB benefits from an indexing database layer, but it loses its optimization features due to

MapReduce’s query execution controller. MapReduce adopts a brute-force scanning method to execute query. In hybrid systems, most of the query operations execute in the database layer and can benefit from the existence of an index in a single database node. The HadoopDB system supports analyzing Big Data and it expects to store huge amounts of data. Accordingly, the system is required to build a massive index. Thus, carefully selecting the index-set configuration is more urgent than in smaller systems. From this perspective, we adopt index-selection optimization into HadoopDB to add one of the most important elements of query optimization for performance improvement. The main challenges to developing an index-selection tool on hybrid systems is the lack of a query optimizer. The optimizer not only chooses the optimal execution plan but it also provides information about physical components to help to administrate and design tools for the physical layer efficiently. An alternative way to design the physical layer is to extract information from the system workload, thus we rely on frequent itemset mining to extract information about attributes frequency in the workload and recommend an optimal index configuration for the physical design of the databases in MapReduce and DBMS hybrid system. The tool developed shows significant improvement in the system performance.

A query optimizer component in a hybrid system is a very important research area that has not yet been studied, to best of my knowledge. The challenges in developing a query optimizer on a hybrid system is that the query is executed on two layers: MapReduce and database. The logical execution plan must analyze both layers’ physical components to chooses the optimal plan. In addition, the query is executed in a cluster environment that requires developing a global logical plan for all nodes.

5.2 Future Work

The tool developed in this thesis can be utilized with other frequent itemset algorithms as well as the Apriori algorithm. Two such possible algorithms are the FP-Growth algorithm and the Close Frequent Itemset algorithm. It would be interesting to compare the system performance improvements when using

these algorithms.

Materialized views are an important part of physical design. They are built to store and precompute expensive operations such as join and aggregation. They have an impact on the system performance and on index selection since these might require a specific index configuration. Indeed, materialized views and indexing have the same aim, to speed-up data retrieval and each of them interacts with the other. Thus, extending our proposed tool to include materialized views would probably improve performance.

Our tool runs on MapReduce and DBMS hybrid systems at the system level. Index-selection tools are also needed in MapReduce and DBMS hybrid projects like Hive that work at the interface level. Hive, a major project built on top of Hadoop, provides indexing techniques. However, it naïvely applies the first applicable index. Thus, modifying our tool to make it work with Hive would be an important step forward.

To avoid the limitation on the number of index configurations in analytical benchmarks, the tool could be tested on the TPC-R benchmark to provide diverse index configurations. TPC-R queries could either be coded in on MapReduce or written in Hive after implementing a HadoopDB SMS interface.

References

- [1] SINTEF. "big data, for better or worse: 90last two years". *ScienceDaily*, Last accessed May 2013.
- [2] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proceedings of the VLDB Endowment*, 2(1):922–933, 2009.
- [3] HadoopDB Team. Web page. <http://hadoopdb.sourceforge.net/guide/>, Last accessed July 2009.
- [4] John Gantz and David Reinsel. Extracting value from chaos. *IDC iview*, pages 1–12, 2011.
- [5] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.
- [6] Google. Cloudstore. <http://kosmosfs.sourceforge.net/>, Last accessed August 2014.
- [7] Doug Beaver, Sanjeev Kumar, Harry C Li, Jason Sobel, Peter Vajgel, et al. Finding a needle in haystack: Facebook’s photo storage. In *OSDI*, volume 10, pages 1–8, 2010.
- [8] Jing Han, E Haihong, Guan Le, and Jian Du. Survey on nosql database. In *Pervasive computing and applications (ICPCA), 2011 6th international conference on*, pages 363–366. IEEE, 2011.
- [9] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J Abadi, David J DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of*

- the 2009 ACM SIGMOD International Conference on Management of data*, pages 165–178. ACM, 2009.
- [10] David J DeWitt, R Gerber, Goetz Graefe, M Heytens, K Kumar, and M Muralikrishna. *A High Performance Dataflow Database Machine*. Computer Science Department, University of Wisconsin, 1986.
- [11] Shinya Fushimi, Masaru Kitsuregawa, and Hidehiko Tanaka. An overview of the system software of a parallel relational database machine grace. In *VLDB*, volume 86, pages 209–219, 1986.
- [12] ebay two enormous data warehouses tm. <http://www.dbms2.com/2009/04/30/ebays-two-enormous-data-warehouses/>, Last accessed April 2009.
- [13] Hadoop. Web page. hadoop.apache.org/docs/r1.2.1/hdfs_design, Last accessed August 2014.
- [14] Ralf Lämmel. Google mapreduce programming model revisited tm. *Science of computer programming*, 70(1):1–30, 2008.
- [15] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
- [16] Tom White. *Hadoop: The definitive guide*. ” O’Reilly Media, Inc.”, 2012.
- [17] Apache Hadoop Foundation. Apache hive tm. <http://hive.apache.org/>, Last accessed January 2014.
- [18] Apache Hadoop Foundation. Apache pig tm. <http://pig.apache.org/>, Last accessed July 2014.
- [19] Apache Hadoop Foundation. Apache hbase tm. <http://hbase.apache.org/>, Last accessed August 2014.
- [20] Cloudera. Impala. <http://www.cloudera.com>, Last accessed August 2014.

-
- [21] Apache Hadoop Foundation. Apache sqoop tm. <http://sqoop.apache.org/>, Last accessed August 2014.
- [22] Java OpenSCG-postgreSQL and linux Experts. Bigsql tm. <http://www.bigsql.org/se/>, August 2014.
- [23] Postgresql. <http://www.postgresql.org/about/>, Last accessed August 2014.
- [24] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Stefan Richter, Stefan Schuh, Alekh Jindal, and Jörg Schäd. Only aggressive elephants are fast elephants. *Proceedings of the VLDB Endowment*, 5(11):1591–1602, 2012.
- [25] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schäd. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *Proceedings of the VLDB Endowment*, 3(1-2):515–529, 2010.
- [26] Karl Schnaitter, Neoklis Polyzotis, and Lise Getoor. Index interactions in physical design tuning: modeling, analysis, and applications. *Proceedings of the VLDB Endowment*, 2(1):1234–1245, 2009.
- [27] Schkolnick Finkelstein, Mario Schkolnick, and Paolo Tiberio. Physical database design for relational databases. *ACM Transactions on Database Systems (TODS)*, 13(1):91–128, 1988.
- [28] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, volume 2000, pages 496–505, 2000.
- [29] Kamel Aouiche and Jérôme Darmont. Data mining-based materialized view and index selection in data warehouses. *Journal of Intelligent Information Systems*, 33(1):65–93, 2009.
- [30] Ning Li, Li Zeng, Qing He, and Zhongzhi Shi. Parallel implementation of apriori algorithm based on mapreduce. In *Software Engineering, Artificial Intelligence, Networking and Parallel & Distributed Computing (SNPD), 2012 13th ACIS International Conference on*, pages 236–241. IEEE, 2012.

- [31] Lingjuan Li and Min Zhang. The strategy of mining association rule based on cloud computing. In *Business Computing and Global Informatization (BCGIN), 2011 International Conference on*, pages 475–478. IEEE, 2011.
- [32] Xin Yue Yang, Zhen Liu, and Yan Fu. Mapreduce as a programming model for association rules algorithm on hadoop. In *Information Sciences and Interaction Sciences (ICIS), 2010 3rd International Conference on*, pages 99–102. IEEE, 2010.
- [33] Ming-Yen Lin, Pei-Yu Lee, and Sue-Chen Hsueh. Apriori-based frequent itemset mining algorithms on mapreduce. In *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*, page 76. ACM, 2012.
- [34] Michael J Cafarella and Christopher Ré. Manimal: relational optimization for data-intensive programs. In *Proceedings of the 13th International Workshop on the Web and Databases*, page 10. ACM, 2010.
- [35] Eaman Jahani, Michael J Cafarella, and Christopher Ré. Automatic optimization for mapreduce programs. *Proceedings of the VLDB Endowment*, 4(6):385–396, 2011.
- [36] Sai Wu, Feng Li, Sharad Mehrotra, and Beng Chin Ooi. Query optimization for massively parallel data processing. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 12. ACM, 2011.
- [37] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. In *ACM SIGMOD Record*, volume 22, pages 207–216. ACM, 1993.
- [38] Rakesh Agrawal and John C Shafer. Parallel mining of association rules. *IEEE Transactions on knowledge and Data Engineering*, 8(6):962–969, 1996.
- [39] TPC Transaction Processing Performance Council. <http://www.tpc.org/tpcbd/default.asp>. Last accessed August 2014.

Glossary

FP-Growth algorithm Algorithm for Associating Rules Mining.It utilizes frequent-pattern tree(FP-tree) data structure to generate frequent itemset instead of generating candidate to reduce I/O cost comparing with Apriori algorithm.On the negative side, the algorithm required huge memory to store the FP-tree.. 64

MapReduce Counter MapReduce framework provides a number of build in counters and allow user to define others. Counters contain useful information about the running MapReduce job such as number of input, output, map, combine and reduce records.. 47

Acronyms

DAG Directed Acyclic Graphs. 29

DFS Distributed File System. 7

DTA Database Engine Tuning Advisor. 35

HDFS Hadoop Distributed File System. 5

JDBC Java Database Connectivity. 10

JVM Java Virtual Machine. 51

PDBMS Parallel Database Management System. 9

SMS SQL-MapReduce-SQL. 27

TPC Transaction Processing Performance Council. 51