ART: A Large Scale Microblogging Data Management System

Li Feng

Bachelor of Science

Peking University, China

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2014

DECLARATION

I hereby declare that this thesis is my original work and it has been written by me in its entirety.

I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has not been submitted for any degree in any university previously.

Li Feng 19 May 2014

ACKNOWLEDGEMENT

This thesis would not have been possible without the guidance and help of many people. It is my pleasure to thank these people for their valuable assistance to my PhD study in these years.

First and foremost, I would like to express my sincere gratitude to my supervisor, Prof Beng Chin Ooi, for his patient guidance throughout my time as his students. He taught me the research skills and right working attitude, and offered me the internship opportunities at research labs.

I would like to thank Prof M. Tamer Ozsu, for his valuable guidance for my third work and the survey, as well as his painstaking effort in correcting my writings. I would also like to thank Dr Sai Wu, who is also a close friend to me, for his support and advice to my first two works. In addition, I would like to thank Vivek Narasayya, Manoj Syamala, Sudipto Das, and all the other researchers in Microsoft Research Redmond, from who learned the right working style of a good researcher.

I would also like to thank all my fellow labmates in database research lab, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last four years.

At last, I would like to thank my family: my parents Fusheng Li and Zhimin Liu, and my wife Lian He. They were always supporting me and encouraging me with their best wishes.

CONTENTS

ABSTRACT

Microblogging, a new social network, has attracted the interest of billions of users in recent years. As its data volume keeps increasing, it has becomes challenging to efficiently manage these data and process queries on these data. Although considerable researches have been conducted on the large scale data management problems and the microblogging service providers have also designed scalable parallel processing systems and distributed storage systems, these approaches are still inefficient comparing to traditional DBMSs that have been studied for decades. The performance of these systems can be improved with proper optimization strategies.

This thesis is aimed to design a scalable, efficient and full-functional microblogging data management system. We propose ART (AQUA, R-Store and TI), a large scale microblogging data management system that is able to handle various user queries (such as updates and real-time search) and the data analysis queries (such as join and aggregation queries). Furthermore, ART is specifically optimized for three types of queries: multi-way join query, real-time aggregation query and real-time search query. Three principle modules are included in ART:

1. Offline analytics module. ART utilizes MapReduce as the batch parallel processing engine and implements AQUA, a cost-based optimizer on top of MapReduce. In AQUA, we propose a cost model to estimate the cost of each join plan, and the near-optimal one is selected by the plan iteration algorithm.

2. OLTP and real-time analysis module. In ART, we implement a distributed key/value store, R-Store, for the OLTP and real-time aggregation query processing. A real-time data cube is maintained as the historical data, and the newly updated data are merged with the data cube on the fly during the processing of the real-time query.

3. Real-time search module. The last component of ART is TI, a distributed real-time indexing system for supporting real-time search. The ranking function considers the social graphs and discussion topics in the microblogging data, and the partial indexing scheme is proposed to improve the throughput of updating the real-time inverted index.

The result of experiments conducted on TPC-H data set and the real Twitter data set, demonstrates that (1) the join plan selected by AQUA outperforms the manually optimized plan significantly; (2) the performance of the real-time aggregation query processing approach implemented in R-Store is better than the default one when the selectivity of the aggregation query is high; (3) the real-time search results returned by TI are more meaningful than the current ranking methods. Overall, to the best of our knowledge, this thesis is the first work that systematically studies how these queries are efficiently processed in a large scale microblogging system.

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Microblogging is an emerging social network that has attracted many users in recent years. It is well known for its distinguishing features, which can be summarized as follows:

1. Limited length of content. Different from traditional blogging system, the length of a microblog is fairly short (e.g. in Twitter, it is capped at 140 characters).

2. Real-time information sharing. Due to the limited length of the microblogs, it is quite convenient for users to post their opinions or the surrounding events, and this information is immediately shared to their friends. Thus, the microblogs contain the most real-time information about what are happening in the world.

3. Massive amount of data. The number of users and the amount of data in a microblogging system have been dramatically increased in the past a few years. It is reported that the number of twitter (one of the most popular microblogging vendors[1]) accounts has reached 225 million by the end of 2011. And there were more than 250 million tweets posted per day.

Because of the popularity of microblogging and the valuable information contained in the microblogging data, it is important that a microblogging data

[1] https://twitter.com/

management system should be able to efficiently process various OLTP and OLAP queries. However, due to the unexpected increase of microblogging data, the existing database management systems are no longer qualified for processing the queries on the data at such a scale. Therefore, many researches have been proposed to investigate how a microblogging data management system should be designed. For example, twitter has designed a distributed datastore, Gizzard, for accessing the distributed data quickly [13], and Facebook has implemented Cassandra [70] to store the large amount of data. In addition, MapReduce [44] has been widely used by these social network companies to handle the data analysis jobs. However, most of these works only focus on the subsystem (storage, parallel processing or search engine) of a microblogging system, and the performance of these subsystems can be further improved with proper optimization strategies. In this thesis, instead of delving in only a specific subsystem of a microblogging system, we design a complete and scalable microblogging data management system, ART (AQUA, R-Store and TI), that can process the major queries in microblogging systems. These queries include the basic user queries (such as update, insert, delete and real-time search) and the complex data analysis queries (like join and aggregation). In addition to simply supporting these queries, ART is specifically designed to improve the performance of multi-way join query, aggregation query and real-time search query compared to the existing systems.

In this chapter, we will first introduce the overview of ART in Section 1.1. We then discuss the research challenges in microblogging data management in Section 1.2. Specifically, we will show the limitations of the methods for processing the multi-way join query, real-time aggregation query and real-time search query in existing systems, and briefly discuss our solution. At last, we will summarize the objectives and significance of this work (Section 1.3) and introduce the synopsis of this thesis (Section 1.4).

1.1 Overview of ART

A microblogging data management system typically has two major modules: the offline analytics module that is used to analyze the microblogging data; and the OLTP and online analytics module for updating the data based on user actions and supporting the real-time analytics. These two modules must

be scalable in order to cope with the increasing data volume in microblogging system. In addition, in microbloging system, a search module is also required to support the real-time search query, which has attracted much research since the emergence of microblogging.

- **Offline Analytics Module.** Offline data analytics module is an important part of a microblogging data management system. It is used to analyze microblogging data in order to extract some valuable information that will be used for decision making. DBMSs have evolved over the last four decades as platforms for managing the data and supporting data analysis, but they are now being criticized for their monolithic architecture that is hard to scale to satisfy the requirement of current microblogging companies. Instead, MapReduce[44], a parallel query processing platform that is well known for its scalability, flexibility and fault-tolerance, has been widely used as the offline analytics module[2]. However, since MapReduce has a simplified programming language that requires a large amount of work from the programmers, the high level systems such as Hive [101] and Pig [83] are usually used to automatically translate the OLAP queries to MapReduce jobs.

  In ART, we adopt an open sourced implementation of MapReduce, Hadoop, as the parallel processing module. In addition, we propose AQUA, a high level system that is implemented by embedding a cost based query optimizer into Hive. AQUA provides similar functionality to Hive, which automatically translates a SQL query into a sequence of MapReduce jobs. In addition, for a multi-way join query, AQUA is able to iterate the possible join plans using a heuristic plan iteration algorithm and estimate the cost of each plan based on the proposed cost model. Finally, the near-optimal join plan is selected by AQUA and will be submitted to MapReduce for execution.

- **OLTP and Real-Time Analytics Module.** To store and update the microblogging data at such a scale, distributed key/value stores, instead of the single node database management systems (DBMSs), have been adopted. For example, Cassandra[3] has been used by the GEO team in

[2]shading https://blog.twitter.com/2012/generating-recommendations-mapreduce-and-scalding

[3]https://blog.twitter.com/2010/cassandra-twitter-today

twitter to store the tweet data, and HBase[4] has been adopted by tumblr as part of their storage system. User actions such as posting a new microblog or replying to friends incur OLTP operations (update, delete, insert, etc) to the storage system.

ART also uses a distributed key/value store to store and update the microblogging data. Different from the other distributed key/value stores, to enable real-time data analytics, the underlying storage module in ART, R-Store, is redesigned so that the latest data can be quickly accessed by the analysis engine. We implement R-Store by extending an open source distributed key/value system, HBase, to store the real-time data cube and the microblogging data. R-Store can handle the OLTP queries and update the tables according to the user queries. In addition, these updates are shuffled to a streaming module inside R-Store, which updates the real-time data cube on incremental basis. We propose techniques to efficiently scan the microblogging data in R-Store, and these data will be combined with the real-time data cube during the processing of the real-time aggregation queries. We will discuss R-Store in detail in Chapter 5;

- **Real-time Search Module.** The increasing popularity of social networking systems changes the form of information sharing. Instead of issuing a query to a search engine, the users log into their social networking accounts and retrieve news, URLs and comments shared by their friends. Therefore, in addition to the basic data storage and analytics, supporting real-time search is a new requirement for microblogging system. (e.g., Twitter [16] has released their real-time search engines recently.) A real-time search query consists of a set of keywords issued by the users, and it requires that the microblogs are searchable as soon as they are generated. For example, users may be interested in the latest discussion on the pop star Britney Spears and thus submit the query "Britney spears" to the system. Different from the traditional search engine where the inverted index is built in batch, the index in microblogging system must be maintained in real-time to ensure that the latest microblogs posted should be

[4]http://www.cloudera.com/content/cloudera/en/resources/ library/hbasecon/hbasecon-2012–growing-your-inbox-hbase-at-tumblr-bennett-andrews.html, http://highscalability.com/blog/2012/2/13/tumblr-architecture-15-billion-page-views-a-month-and-harder.html

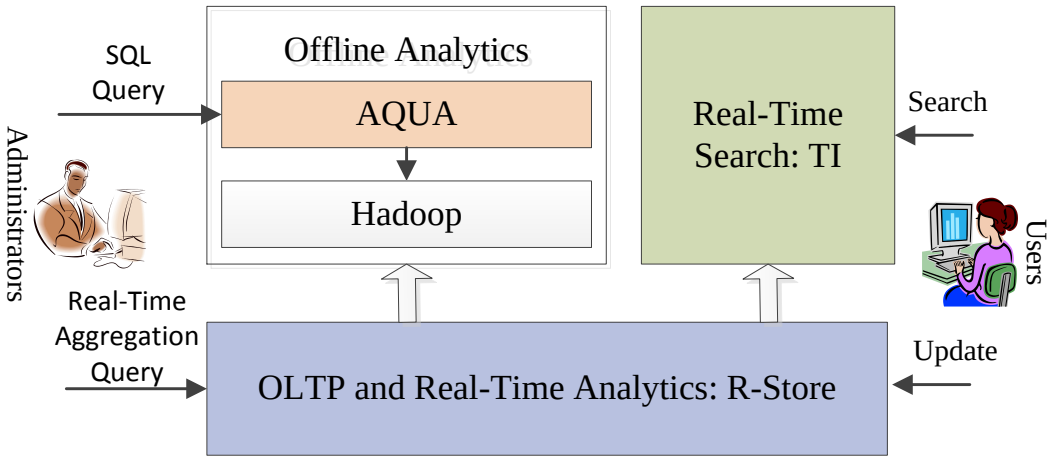


Figure 1.1: Overview of ART

considered if they contain the keywords in the queries.

In ART, a distributed adaptive indexing system, TI, is proposed to support real-time search. The intuition of TI is to index the microblogs that may appear as a search result with high probability and delay indexing some other microblogs. This strategy significantly reduces the indexing cost without compromising the quality of the search results. In TI, we also devise a new ranking scheme by combining the relationship between the users and microblogs. We group microblogs into topics and update the ranking of a topic dynamically, and the popularity of the topic will affect the ranking scores of the microblogs in our ranking scheme. In TI, each search query is issued to an arbitrary query processor (in TI slaves), which collects the necessary information from other nodes and sorts the search results using our ranking scheme. We will discuss TI in detail in Chapter 6.

In summary, Figure 1.1 shows an overview of ART. ART consists of three major modules, and we focus on AQUA, R-Store and TI. In ART, the microblogging data are stored in R-Store. The user actions such as posting a microblog incur the OLTP transactions, and the microblogging data is updated accordingly. The data are periodically exported to the file system of Hadoop (HDFS), and AQUA will translate the SQL queries to MapReduce jobs to analyze these data offline. Different from the offline analysis queries, the real-time analysis queries are directly handled by R-Store. In addition, the newly pub-

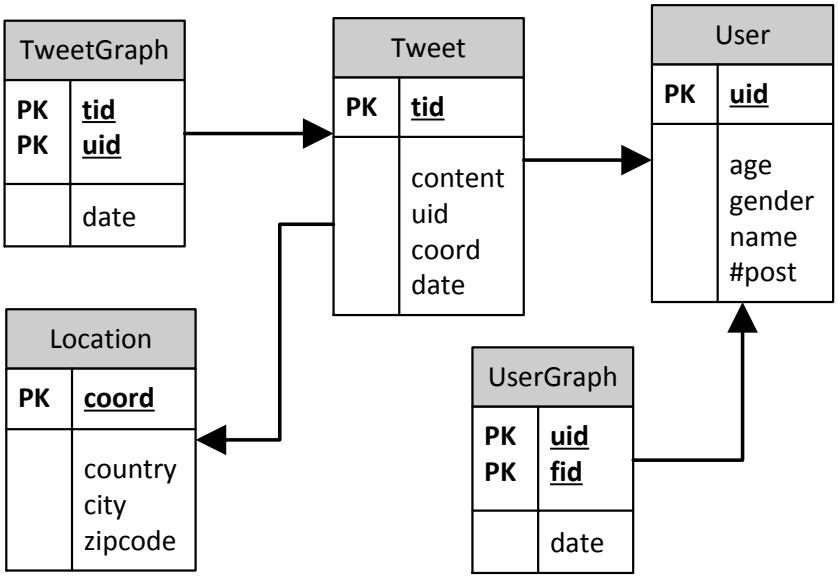


Figure 1.2: Example Twitter Tables

lished microblogs in R-Store are shuffled to TI, and the real-time inverted index are updated accordingly. With these three modules, ART is able to support the requirements of a microblogging data management system. Furthermore, ART is also specifically designed for efficiently processing the multi-way join query, real-time aggregation query and real-time search query. In the next section, we will briefly discuss the research challenges in processing these queries in existing work and how ART addresses these challenges.

1.2 Query Processing in Microblogging Data Management System

Various queries are being executed in the microblogging system, such as OLTP queries, OLAP queries, search queries etc. In this section, we discuss three query types that are common in a microblogging system: multi-way join query and aggregation query are data analysis queries, while real-time search query is a fundamental requirement of microblogging system to ensure that the users can obtain the real-time information about what they are interested in.

To demonstrate these queries more clearly, we first give an example for the schema of the Twitter data. As shown in Figure 1.2, there are five tables in the schema: the *Tweet* table stores the content of each tweet published by

the users; the *User* table stores the information of each user, such as age and gender; the *UserGraph* table stores the following relationship between users; *TweetGraph* stores the replying/retweeting relationship between tweets; and *Locations* stores the mapping between the coordinates and the address. We will refer to this schema in the rest of this thesis.

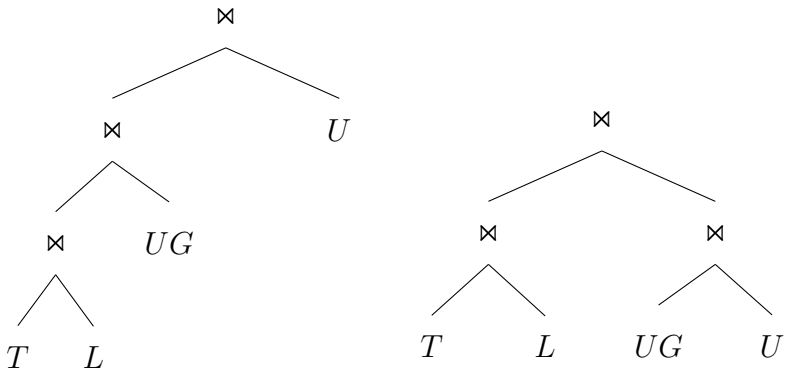
1.2.1 Multi-Way Join Query

In a data management system, multi-way join query is used most frequently and has by far attracted most attention. For example, the administrator of a microblogging system may be interested in the number of tweets published in USA by the followers of Obama, and the following query could solve this problem:

$$
\begin{aligned}
&SELECT\ count(*)\\
&FROM\quad Tweet\ T, User\ U, Location\ L, UserGraph\ UG\\
&WHERE\ T.coord = L.coord\\
&AND\quad T.uid = UG.uid\\
&AND\quad UG.fid = U.uid\\
&AND\quad L.country = USA\\
&AND\quad U.name = \text{“}Obama\text{”}
\end{aligned}
$$

The above multi-way join can be executed as a sequence of equi-joins represented as a tree (as shown in Figure 1.3(a)). Equi-join is an atomic operator of multi-way join. Given tables $Tweet$ and $User$, the equi-join operator creates a new result table by combining the columns of $Tweet$ and $User$ based on the equality comparisons over one or more column values such as *uid*.

To implement the multi-way join in MapReduce, each of the equi-joins in the join tree is performed by one MapReduce job. Starting from the bottom of the tree, the result of each MapReduce job is treated as an input for the next (higher-level) one. The multi-way join has been implemented on top of MapReduce in [101]. However, the order of the equi-join operator is specified by the users. As expected, different join orders lead to different query plans with significantly different performance, but even skilled users cannot select the



(a) Left Deep Plan (b) Bushy Plan

Figure 1.3: Multi-way Join

best join orders when the number of tables involved in the multi-way join is large.

To find the best join order, we need to collect the statistics of the data [60] and estimate the processing cost of each possible plan using a cost model. Many plan generation and selection algorithms [95] that were developed for relational DBMSs can be applied here to find a good plan, but these algorithms have not been designed specially for MapReduce and can be further improved in a MapReduce system. In particular, more time-consuming algorithms may be employed for two reasons. First, the relational optimization algorithms are designed to efficiently balance query optimization time and the query execution time. MapReduce jobs usually run longer than relational queries, and thus call for more time-consuming algorithms that require longer query optimization time to reduce the query execution time. Second, in most relational DBMSs, only left-deep plans [53] (Figure 1.3(a)) are typically preferred to reduce the plan search space and to pipeline the data between operators. There is no pipeline between the operators in the original MapReduce, and, as we indicated above, query execution time is more important. Thus, the bushy plans (Figure 1.3(b)) are often considered for their efficiency.

In ART, to efficiently find a better plan for the multi-way join query in MapReduce, we propose a cost based query optimizer, which uses a heuristic plan generator to reduce search space and considers the bushy plans.

1.2.2 Real-Time Aggregation Query

Aggregation query is usually used to compute a summary of the data stored in data warehouse. For example, if the administrator would like to compute the number of tweets published in a certain day, he may write the following aggregation query to solve the problem:

$$\begin{aligned} &SELECT\ sum(\#post) \\ &FROM\ \ \ User\ U \\ &WHERE\ U.age = 30 \end{aligned}$$

However, in the current data management system, the freshness of the above query has become an issue. Currently, data management systems implemented for large scale data processing (including microblogging system) are typically separated into two categories: OLTP systems and OLAP systems. The data stored in OLTP systems are periodically exported to OLAP systems through Extract-Transform-Load (ETL) tools. In recent years, MapReduce framework has been widely used in implementing large scale OLAP systems because of its scalability, and these include Hive [101], Pig [83] and HadoopDB [17]. Most of these only focus on optimizing OLAP queries, and are oblivious to updates made to the OLTP data since the last loading. However, with the increasing need to support real-time online analytics, the issue of freshness of the OLAP results has to be addressed, for the simple fact that more up-to-date analytical results would be more useful for time-critical decision making.

The idea of supporting real-time OLAP (RTOLAP) has been investigated in traditional database systems. The most straightforward approach is to perform near real-time ETL by shortening the refresh interval of data stored in OLAP systems [102]. Although such an approach is easy to implement, it cannot produce fully real-time results and the refresh frequency affects system performance as a whole. Fully real-time OLAP entails executing queries directly on the data stored in the OLTP system, instead of the files periodically loaded from the OLTP system. To eliminate data loading time, OLAP and OLTP queries should be processed by one integrated system, instead of two separate systems. However, OLAP queries can run for hours or even days, while OLTP queries

Figure 1.4: Example of Twitter Search obtained on 10/29/2010

take only microseconds to seconds. Due to resource contention, an OLTP query may be blocked by an OLAP query, resulting in a large query response time. On the other hand, if updates by OLTP queries are allowed as a way to avoid long blocking, since complex and long running OLAP queries may access the same data set multiple times, the result generated by the OLAP query would be incorrect (the well-known dirty data problem).

Fully supporting real-time OLAP in a distributed environment is a challenging problem. Since a complex analysis query can be executed for days, by the time that the query is completed, the result is in fact not "real-time" any more. In this thesis, we focus on supporting real-time processing for a subset of the OLAP queries: aggregation queries. A real-time aggregation query in our system accesses, for each key, the latest value preceding the submission time of the query [52]. Compared to the other queries such as join queries, pure aggregation query only involves one table, and thus its processing logic is much simpler and has more opportunities to be improved. We will discuss how we optimize the real-time aggregation query in Chapter 5

1.2.3 Real-Time Search Query

To ensure that the users can search for the current happening events or discussions that they're interested in, the search service is a required component of microblogging system. Figure 1.4 illustrates an example on the search results of Twitter for the keyword "inception". As shown in this figure, even the tweets that are published less than 20 seconds ago can be searched. However, in conventional search engines, the search service is provided via crawling the web pages and updating the index periodically. The freshness of the index and relevance of the web pages with respect to the search results would therefore rely on the frequency in which pages are crawled and the indexes are updated. Such approach is not ideal for supporting search in microblogging systems, where thousands of concurrent users may upload their microblogs or tweets simultaneously. To make a blog or tweet searchable as soon as it is produced, the index must be created or updated in real time.

Providing real-time search service is indeed very challenging in large-scale microblogging systems. In such a system, thousands of new updates need to be processed per second. To make every update searchable, we need to index its effect in real time and provide effective and efficient keyword-based retrieval at the same time. The objectives are therefore contradictory since maintenance of up-to-date index will cause severe contention for locks on the index pages.

Another problem of real-time search is the lack of effective ranking functions. For example, the user is perhaps looking for the reviews and comments about the movie "Inception". However, most search results in Figure 1.4 are not related to the movie, and most of returned tweets do not even provide any useful information. This is because the current Twitter search engine sorts the results based on time, and therefore, the latest tweets have the higher rankings. Recall that one key factor of Google's early success is its PageRank [85] algorithm. Without proper ranking functions, the search results are meaningless.

In ART, we propose *TI*, a distributed adaptive indexing system for supporting real-time search. It only indexes the tweets when they have high probability to be searched by a search query, and offers a new ranking scheme that considers the relationship between the tweets and the users.

1.3 Objectives and Significance

Microblogging is a popular social network that has attracted billions of users throughout the world. Because of the huge amount of data generated in the microblogging systems, it has become more challenging to efficiently process the queries using existing DBMSs. Therefore, the large scale systems discussed in section 1.1 has been used by the microblogging service providers. However, there are still some unsolved problems in existing systems, which are summarized as follows:

- Most of the existing systems only focus on a particular subsystem of a microblogging data management system.
- In current offline analytics module, the order of the multi-way join is decided by the programmer. Unfortunately, manual query optimization is time-consuming and difficult, even for an experienced database user or administrator.
- The microblogging data are usually stored in OLTP module and periodically exported to OLAP module. The OLAP query, such as aggregation, does not consider the newly updated data, and the freshness of the query result becomes a concern.
- The ranking scheme of exiting real-time search query is not proper, and thus the search results are meaningless. In addition, as there are huge amount of microblogs updated per day, the exiting indexing scheme may not be able to index these updates in real-time.

The main aim of this thesis is to propose a full-functional and scalable microblogging data management system that is optimized for the three query types discussed in Chapter 1.2. The specific objectives of this thesis are:

- To design a full-functional microblogging data management system that supports OLTP, offline data anlytics, real-time data analytics and real-time search.
- To improve the performance of multi-way join query by a cost based optimization in the offline analytics module.

- To efficiently process the real-time aggregation queries in the real-time analytics module.

- To devise a more effective ranking scheme for the real-time search module, and design a more efficient approach to build and update the real-time inverted index.

The main contributions of this thesis are as follows:

- First, we design a cost-based optimizer to efficiently translate the multi-way join queries to MapReduce jobs. Our proposed plan iteration algorithm can be completed within a short period of time compared to the execution time of the join queries, and the plan selected by our cost optimizer significantly outperforms the manually optimized plans.

- Second, we propose a large scale system for supporting real-time aggregation queries. The real-time data cube and the real-time data are stored in the system and will be used during the processing of the aggregation queries. We develop different algorithms for the real-time aggregations and the better algorithm is automatically selected based on the statistics of the data, which is transparent to the users.

- Third, we propose an adaptive indexing scheme for microblogging systems such as Twitter. It reduces the indexing cost by only indexing the tweets that may appear as a search result in real-time. The other tweets are indexed in batch. We also devise a new ranking scheme that considers the relationship between the users and tweets.

- Last, we implement subsystem for each of the methods we propose in the three works, and these systems are integrated to ART(AQUA, R-Store, TI), a large scale microblogging data management system. Though the purpose of this thesis is to efficiently process queries in a microblogging data management system, the approaches proposed can be applied to other large scale systems (such as blogging systems, search engines and distributed key/value stores) as well.

1.4 Thesis Organization

The remaining of this thesis is organized according to the three components (AQUA, R-Store and TI) that we have proposed in Figure 1.1:

1. Chapter 2 reviews the related work of the three works in this thesis.
2. Chapter 3 presents the design philosophy and architecture of ART.
3. Chapter 4 introduces AQUA, a cost-based query optimizer for multi-way join queries on MapReduce.
4. Chapter 5 presents R-Store, a modified version of HBase that supports large scale real-time aggregation query processing.
5. Chapter 6 presents TI, an efficient indexing system for supporting real-time search queries on tweets.
6. Chapter 7 concludes this thesis and discusses possible directions for future work.

CHAPTER 2

Literature Review

In recent years, microblogging systems such as Twitter and Tumblr have become basic communication methods for the people to share their opinions, discoveries and activities with their friends. According to a report, Twitter has more than 500 million active registered users by May, 2013[1]. Due to its popularity and the huge data volume, it has attracted the design of a distributed data management system to handle the OLTP or search queries issued by the users, and the data analysis queries submitted by the administrators. In this chapter, we shall first review some exiting large scale systems used in the industry (Section 2.1), and then review the related works on multi-way join, real-time aggregation and real-time search query processing.

2.1 Large Scale Data Storage and Processing Systems

Database management systems (DBMSs) [87] have evolved over the last four decades in managing business data and are now functionally rich. However, DBMSs have been criticized for their monolithic architecture that is hard to scale to satisfy the requirement of the current internet applications where petabyte of data are generated every day. There have been various proposals to

[1] http://www.statisticbrain.com/twitter-statistics/

restructure DBMSs (e.g., [33, 97]), but the basic architecture has not changed dramatically. Though database systems have been extended and parallelized to run on multiple hardware platforms to manage scalability [84], with the ever increasing amount of data and the availability of high performance and relatively low-cost hardware, some new "big data" platforms have been designed and implemented by companies such as Google, Facebook and Microsoft. These systems have the following two fundamental features:

1. **Scalability**. A major challenge in many existing applications is to be able to scale to increasing data volumes. In particular, *elastic scalability* is desired, which requires the system to be able to scale its performance up and down dynamically as the computation requirements change. Such a "pay-as-you-go" service model is now widely adopted by the cloud computing service providers.

2. **Fault tolerance**. The data are usually replicated in multiple machines, and the failure of a task or a machine is compensated by assigning the task to a machine that is able to handle the load.

We classify these systems into two categories: distributed storage systems and parallel processing systems.

2.1.1 Distributed Storage Systems

In recent years, the rapidly growing popularity of web applications, such as online social network and shopping, significantly raises the transaction volume, and the workload of the OLTP systems as a consequence. Nevertheless, it is found that the traditional databases, which enforce the strong consistency on data models, are incapable of achieving the requirements discussed above. An early study [51] proves that any binary combination of consistency, availability and scalability is achievable but not ternary. Hence, the consensus is to trade consistency for the other two metrics. Google's BigTable [30] is one of the first systems following this design philosophy. Bigtable is a sparse, distributed, persistent multi-dimensional sorted map, which is indexed by a row key, column key, and a timestamp. Rows are sorted by key and the whole Bigtable is partitioned into a number of tablets according to the specified row key ranges. In addition to row keys, columns have keys as well (equivalent the attribute names of tables in relational databases), and are grouped into the column family.

HBase [3] is an open source version of BigTable, which adopts BigTable's master-slave architecture as well. The master server is responsible for distributing tablets to tablet servers, monitoring the states of tablet servers, balancing the workload of them. Moreover, it handles metadata modifications such as table and column family creations and updates. Each tablet server hosts a set of tablets, handles read and write requests to the tablets, and also partitions the tablets if they have grown large enough.

Cassandra is a distributed storage system originating from Facebook [69], and is now a popular open source project under Apache Foundation [1]. It adopts similar data model as BigTable, but has a different system architecture: it uses consistent hashing to organize the data in the cluster. A hash function is employed to generate keys within some key space, which forms a circle by concatenating the largest value to the smallest one. Each node is assigned a key that represents the position of it in the system. Each data item also has a key to be identified. The key also determines on which node the data item is stored: the first node whose key is no larger than the data item's.

In addition to the BigTable-like storage model, Dynamo [45], which is designed by Amazon Inc, adopts the pure key/value storage model. Dynamo uses consistent hashing to partition data as well. Moreover, through real-world deployment and operation, Amazon found the basic partition method did not work well with nonuniform data distribution and heterogeneous node capacity. To improve the performance, they made some modifications: the whole key space is divided into a number of equal-size partitions; and each node is responsible for multiple partitions, proportional to its capacity. The replication strategy is straightforward in Dynamo. Assume k replications are required. Then, the data item is stored on the node that is responsible for its key, and is replicated on $k-1$ nodes who are the clockwise successors of the node.

Whereas Dynamo [45] and Cassandra [69] can only support eventual consistency, Cooper *et al.* [43] claims that the eventual consistency model is often too weak and hence inadequate for web applications. The argument given by Cooper *et al.* is based on the observation of Yahoo!'s applications. According the specific requirements of their applications, the authors designed and implemented a centrally-managed, geographically-distributed and automatically-load-balancing storage system, named PNUTS. With PNUTS [43], a considerable number of concurrent requests can be replied within a short latency. Table

	Dynamo	**Cassandra**	**PNUTS**	**HBase**
Consistency	Eventual	Eventual	Timeline	Full
Replication	Asynchronized	Asynchronized	Asynchronized	Asynchronized
Data Model	Key-value	Column-family	Table	Column-family
Underlying Storage	Local file system	Local file system	Local database	HDFS [2]
Architecture	P2P	P2P	Master-slave	Master-slave
Optimized For	Writes	Writes	Writes	Reads

Table 2.1: Summary of well-known OLTP systems.

Table 2.2: `map` and `reduce` Functions

`map`	$(k1, v1) \rightarrow list(k2, v2)$
`reduce`	$(k2, list(v2)) \rightarrow list(v3)$

2.1 summarizes the characteristics of the distributed storage systems discussed in this section.

2.1.2 Parallel Processing Systems

As the traditional DBMSs can hardly scale to thousands of nodes, many new parallel processing systems have been proposed recently. Among these systems, the most popular one is MapReduce [44]. MapReduce is a simplified parallel data processing approach for execution on a computer cluster. (We have written a detailed survey on MapReduce in [72].) Its programming model consists of two user defined functions, `map` and `reduce`(Table 2.2). The inputs of the `map` function is a set of key/value pairs. When a MapReduce job is submitted to the system, the map tasks (which are processes that are referred to as *mappers*) are started on the compute nodes and each map task applies the `map` function to every key/value pair $(k1, v1)$ that is allocated to it. Zero or more intermediate key/value pairs $(list(k2, v2))$ can be generated for the same input key/value pair. These intermediate results are stored in the local file system and sorted by the keys. After all the map tasks complete, the MapReduce engine notifies the reduce tasks (which are also processes that are referred to as *reducers*) to start their processing. The reducers will pull the output files from the map tasks in parallel, and merge-sort the files obtained from the map tasks to combine the key/value pairs into a set of new key/value pair $(k2, list(v2))$, where all values with the same key $k2$ are grouped into a list and used as the input for the `reduce` function. The `reduce` function applies the user-defined processing

logic to process the data. The results, normally a list of values, are written back to the storage system. MapReduce processing engine has two types of nodes, the *master* node and the *worker* nodes. The master node controls the execution flow of the tasks at the worker nodes via the *scheduler* module. Each worker node is responsible for a `map` or `reduce` process.

An interesting line of research has been to develop parallel processing platforms that have MapReduce flavor, but are more general. Two examples of this line of work are Dryad [58] and epiC [34].

Dryad [58] represents each job as a directed acyclic graph whose vertices correspond to processes and whose edges represent communication channels. Dryad jobs (graphs) consist of several stages such that vertices in the same stage execute the same user-written functions for processing their input data. Consequently, MapReduce programming model can be viewed as a special case of Dryad's where the graph consists of two stages: the vertices of the map stage shuffles their data to the vertices of the reduce stage.

Driven by the limitations of MapReduce-based systems in dealing with "varieties" in cloud data management, epiC [34] was designed to handle variety of data (e.g., structured and unstructured), variety of storage (e.g., database and file systems), and variety of processing (e.g., SQL and proprietary APIs). Its execution engine is similar to Dryad's to some extent. The important characteristic of epiC, from a MapReduce or data management perspective, is that it simultaneously supports both data intensive analytical workloads (OLAP) and online transactional workloads (OLTP). Traditionally, these two modes of processing are supported by different engines. The system consists of the Query Interface, OLAP/OLTP controller, the Elastic Execution Engine (E3) and the Elastic Storage System (ES2) [28]. SQL-like OLAP queries and OLTP queries are submitted to the OLAP/OLTP controller through the Query Interface. E3 is responsible for the large scale analytical jobs, and ES2, the underlying distributed storage system that adopts the relational data model and supports various indexing mechanisms [36, 103, 107], handles the OLTP queries.

2.2 Multi-Way Join Query Processing

The philosophy of MapReduce is to provide a flexible framework that can be used to solve different problems. Therefore, MapReduce does not provide a

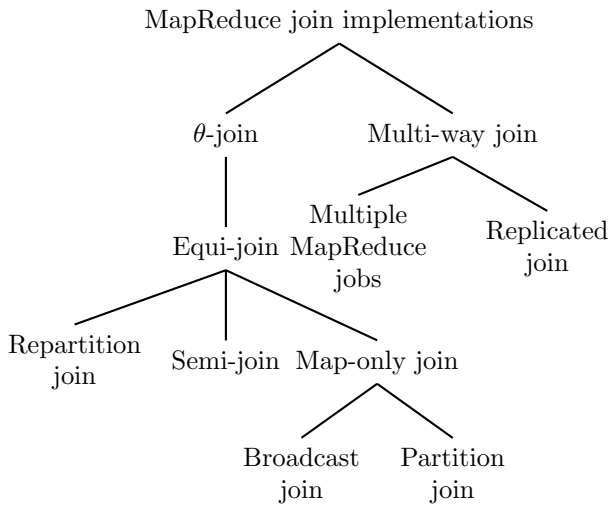


Figure 2.1: Join Implementations on MapReduce

query language, expecting the users to implement their customized `map` and `reduce` functions. While this provides considerate flexibility, it adds to the complexity of application development. To make MapReduce easier to use, a number of high-level languages have been developed, some of which are SQL-like (HiveQL [101], Tenzing [31]), others are data flow languages (Pig Latin [83]), and some are declarative machine learning language (SystemML [50]). Among these languages, HiveQL is the most popular one as SQL-like language has been used for years in data management system. In this section, we review how the SQL operators are implemented using the MapReduce interface. Simple operators such as `select` and `project` can be easily supported in the `map` function, while complex ones, such as `theta-join` [82], `equi-join` [26] and `multi-way join` [108, 62] require significant effort.

The projection and filtering can be easily implemented by adding a few conditions in the `map` function to filter the unnecessary columns and tuples. The implementation of aggregation was discussed in the the original MapReduce paper. The mapper extracts an aggregation key for each incoming tuple (transformed into key/value pair). The tuples with the same aggregation key are shuffled to the same reducers, and the aggregation function (e.g., `sum`, `min`) is applied to these tuples. Join operator implementations have attracted by far the most attention, as it is one of the most expensive operators and a better implementation may potentially lead to a significant performance improvement. Therefore, in this section, we will focus our discussion on the join operator. We

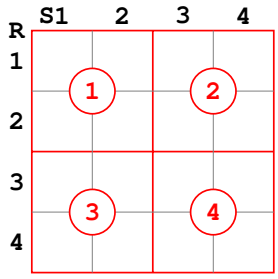


Figure 2.2: Matrix-to-reducer mapping for cross-product

summarize the existing join algorithms in Figure 2.1.

2.2.1 Theta-Join

Theta-join [113] is a join operator where the join condition θ belongs to $\{<, \leq, =, \geq, >, \neq\}$. It is a very expensive database operator, since $|R \bowtie_\theta S|$ is close to $|R| \times |S|$ and few optimization techniques are available. To efficiently implement theta-join on MapReduce, the $|R| \times |S|$ tuples should be evenly distributed on the r reducers, which means that each reducer generates about the same number of results: $\frac{|R| \times |S|}{r}$. To achieve this goal, a randomized algorithm, 1-Bucket-Theta algorithm, was proposed [82] that evenly partitions the join matrix into buckets (Figure 2.2), and assigns each bucket to only one reducer to eliminate the duplicate computation, while also ensuring that all the reducers are assigned the same number of buckets to balance the load.

2.2.2 Equi-Join

Equi-join is a special case of θ-join where θ equals to "=". The strategies for MapReduce implementations of the equi-join operator follows earlier parallel database implementations [90]. Given tables R and S, the equi-join operator creates a new result table by combining the columns of R and S based on the equality comparisons over one or more column values. There are three variations of equi-join implementations (Figure 2.1): repartition join, semijoin-based join, and map-only join (joins that only require map side processing).

Repartition Join [26] is the default join algorithm for MapReduce in Hadoop. The two tables are partitioned in the map phase, followed by shuffling the tuples with the same key to the same reducer that joins the tuples.

Semijoin-based join has been well studied in parallel database systems (e.g., [24]), and it is natural to implement it on MapReduce [26]. The semijoin operator implementation consists of three MapReduce jobs. The first is a full MapReduce job that extracts the unique join keys from one of the relations, say R, where the map task extracts the join key of each tuple and shuffles the identical keys to the same reducer, and the reduce task eliminates the duplicate keys and stores the results in DFS as a set of files $(u_0, u_1, ..., u_k)$. The second job is a map-only job that produces the semijoin results $S' = S \ltimes R$). In this job, since the files that store the unique keys of R are small, they are broadcast to each mapper and locally joined with the part of S (called *data chunk*) assigned to that mapper. The third job is also a map-only job where S' is broadcast to all the mappers and locally joined with R.

Map-only join can be used if the tables are already co-partitioned based on the join key. In this case, for a specific join key, all tuples of R and S are co-located in the same node. The scheduler loads the co-partitioned data chunks of R and S in the same mapper to perform a local join, and the join can be processed entirely on the map side without shuffling the data to the reducers. This co-partitioning strategy has been adopted in many systems such as Hadoop++ [46].

2.2.3 Multi-Way Join

The multi-way join can be executed as a sequence of equi-joins, each of which is performed by one MapReduce job. The result of each MapReduce job is treated as input for the next MapReduce job. As different join orders lead to different query plans with significantly different performance, it is important to find the best join order for a multi-way join. The first step is to collect the statistics of the data (e.g., in [60], the problem of efficiently building histogram on MapReduce was investigated), and the second step is to estimate the processing cost of each possible plan using a cost model. Using the estimated cost for each binary join in the join tree, we can step-by-step calculate the cost of the multi-way join.

Many plan generation and selection algorithms that were developed for relational DBMSs can be directly applied here to find the optimal plan. These optimization algorithms can be further improved in a MapReduce system [108];

in particular, more elaborate algorithms may be deployed. As MapReduce jobs usually run for a long time, justifying more elaborate algorithms (i.e., longer query optimization time) if they can reduce query execution time. In addition, instead of considering only the left-deep plans, the bushy plans are often considered for their efficiency.

2.3 Real-time Aggregation Query Processing

As discussed in previous section, the large scale data management system usually consists of two parts: OLTP module and OLAP module. The OLTP module handles a large number of short transactions oriented from the user interactions in the website, while the OLAP module processes the data analysis queries issued by the administrators or database users. The data stored in OLTP module are periodically exported to OLAP module. Therefore, freshness of the OLAP results is an issue that needs to be resolved. The idea of supporting real-time OLAP has been studied in traditional database systems, and we will review these work in this section. In addition, there have been some work on distributed stream processing, which are related to our work as they also focus on how the timely results are returned.

2.3.1 Real-Time Data Warehouse

The growing demand for fast business analysis coupled with increasing use of stream data have generated great interest in real-time data warehousing [105]. Some have proposed near real-time ETL [64, 102], as a means to shorten the data warehouse refreshing intervals. These works require fewer modifications to the existing systems, but they cannot achieve 100% real-time. Other studies proposed online updates in data warehouses by using differential techniques [56, 98], or multi-version concurrency control [68]. In C-store [98] two separate stores are used to handle in-place updates. The updates are stored in a write-store (WS), while queries run against the read-store (RS), and merged with the WS during execution. In existing studies, the incoming updates are usually cached to improve the performance. The cached data are then flushed to disk once the size exceeds the upper bound. The performance of these studies are limited by the size of the memory, and MaSM [21] overcomes these limitations

by utilizing the SSDs to cache incoming updates. Recently, with the drastic increase of main memory capacity, some in-memory data warehouses have been proposed to process both OLTP and OLAP queries together, and these work include SAP Hana [47] and Hyper [66]. In the main memory data warehouse, the OLAP queries are run on the up-to-date snapshot of the real-time data. The tuples in the snapshot are deleted once the OLAP query is completed and new updates are applied to the tuples. In R-Store, the similar approach is adopted to compact the tuples that have multiple versions. However, our approach are disk-based as in a "big data" system where thousands of nodes are deployed on commodity machines, it is not cost-effective to use the pure in-memory structure.

2.3.2 Distributed Processing

Some recent distributed stream systems support real-time data stream processing that returns the aggregation result of the up-to-date data. HStreaming [4] and MapReduce Online [42] are extensions to the MapReduce framework, which support stream processing by the following three aspects: (1) the input of the mappers could be stream data; (2) the data are streamed from mappers to reducers; and (3) the output of MapReduce job can be streamed to the next job.

Different from the above two systems that extends MapReduce to process data streams, S4 [80] is a distributed stream processing system that follows the Actor programming model. Each keyed tuple in the data stream is treated as an event and is the unit of communication between Processing Elements (PEs). PEs form a directed acyclic graph, which can also be grouped into several stages. At each stage, all the PEs share the same computation function, and each PE processes the events with certain keys. The architecture of S4 is different from the MapReduce-based systems: it adopts a decentralized and symmetric architecture. In S4, there is no master node that schedules the entire cluster. The cluster has many processing nodes (PNs) that contains several PEs for processing the events. Since the data are streaming between PEs, there's no on disk checkpoint for the PEs. Thus, the partial fault tolerance is achieved in S4: if a PN failure occurs, its processes are moved to a standby server, but the state of these processes is lost and cannot be recovered.

Storm is another stream processing system in this category that shares many features with S4. A Storm job is also represented by a directed acyclic graph, and its fault tolerance is partial due to the streaming channel between vertex. The difference is the architecture: Storm is a master-slave system like MapReduce. A Storm cluster has a master node (called Nimbus) and worker nodes (called supervisor).

Different from these streaming systems where new tuples are appended to the existing table, in R-Store, we also consider the case in which the tuples are updated (e.g., the users of the microblogging system may update their status, current address, etc).

There have been some researches on supporting both OLTP and OLAP, such as Cloudera [67]. It adopts similar architecture as R-Store: the MapReduce framework is directly run on top of HBase. And thus, it can also support real-time analytics using MapReduce. Different from systems like Cloudera, in this thesis, we investigate how to efficiently process the RTOLAP queries in such a hybrid architecture by materializing the historical data into a data cube and dynamically combining the data cube with the real-time data.

2.3.3 Data Cube Maintenance

As introduced in [54], data cube is N-dimensional array, in which each dimension represents a dimension attribute of the original table, while the value of the array stores the aggregated value of a numerical attribute. Data cube maintenance has been studied for a long time. The earliest works focused on efficient incremental view maintenance for data warehouses [29, 55]. However, as the number of dimension attributes increases, the cost of incrementally updating data cube increases significantly. To improve the performance of data cube maintenance, instead of generating the delta value for all the cuboids during the update process, an method of refreshing multiple cuboids by the delta value of a single cuboid has been proposed [71]. Most of these algorithms were designed for a single node configuration and are not scalable to a distributed environment. However, MapReduce has been used to construct data cube in a large scale distributed environment [92]. The MR-Cube algorithm [79] was proposed to efficiently compute the data cube for holistic measures. In these works, the data cube is usually used for processing OLAP queries without the

real-time requirement, while our system considers both the data cube and the real-time data to process RTOLAP queries.

2.4 Real-Time Search Query Processing

In microblogging social networking, people instantly upload their opinions on the current happening events. If these microblogs can be searched in real-time, the users can understand what other people are discussing and utilize these information to help themselves make some decisions. For example, if a user plans to buy a new ipad in New York but he is not willing to queue in the apple store for a long time, he can search for the keywords such as "apple store New York queue" in twitter. And if he can find some complaint about the length of the queue or the long waiting time, he'd better chose another time to go to the apple store. Because of the importance of supporting real-time search, it has been a basic requirement of traditional search engine since microblogging social networking became popular. Recently, Google and Bing added the real-time search feature into their search engine, where the latest post published in twitter and facebook are returned as part of the search result if they are related to the query[2]. In addition, microblogging companies such as Twitter also have their own real-time search engine. In this section, we will first review the status of the research in real-time search. Moreover, since our proposed real-time search engine, TI, adopts the partial indexing strategy to reduce the cost of indexing, we also review some related work in this area.

2.4.1 Microblog Search

Google [11] and Twitter [16] have released their real-time search engines recently. Google designs its web crawler to adaptively crawl the microblogs, while Twitter relies on an existing technique, such as Lucene [5], to provide the search service. Both of them treat a query as a continuous query and update the results in real time. However, the ranking function only considers the time dimension, and as a result, the results are sorted by time. By studying the users' behavior in the microblogging systems [59], more sophisticated ranking schemes, such as [88], were proposed. However, most ranking schemes are too

[2]http://en.wikipedia.org/wiki/Google_Real-Time_Search

complex and therefore too expensive to compute in real-time, and hence they are precomputed in an offline manner. To address this problem, in [89], noisy tweets are pruned and similar tweets are clustered together. Ranking is computed for the tweets of the same cluster so that the computation cost can be significantly reduced.

Similar to the ranking scheme in [104], in the *TI*, we also considers the relationships between the users to identify the influential tweets. In addition, we group tweets into some topics by examining their relationships captured in a tree structure. In particular, tweets replying to the same tweet or belong to the same thread are organized as a tree. Similar schemes were adopted for forum search [91, 110]. To reduce the ranking cost, *TI* maintains the popular topics in memory and modifies the structure of an inverted index. Compared to the previous work, *TI*'s ranking function is more efficient and incurs less overhead.

2.4.2 Partial Indexing and View Materialization

In database systems, indexes are created to facilitate efficient query processing. However, index maintenance incurs significant overhead and causes lock contention if the update load is high. Instead of indexing the whole dataset, a partial index was proposed for indexing the records that may be queried with high probability. The idea of partial indexing was first proposed in [96], where the advantages of a partial index are analyzed. In [93], a statistical model is built to monitor the query distribution and the partial index is created adaptively. Partial indexing technique is also adopted in the distributed environment. In PIER [77], only rare items are indexed in the DHT (Distributed Hash Table) based peer-to-peer network, while the popular items are searched via flooding. In PISCES [109], a just-in-time indexing scheme that can be dynamically tuned to follow query patterns was proposed to facilitate query processing in a peer-to-peer based data management system.

View materialization shares some similar principles with the partial indexing technique. As it is expensive to materialize the whole dataset, a small portion of data is therefore selectively materialized. [23] and [111] discuss how to adaptively materialize the views in multi-dimensional databases and data warehouse systems. Cost models were proposed in [20] and [39] to automatically select views for materialization. In [94], the adaptive view materialization strategy

is applied to reduce the overhead of stream feeding systems. The proposed *TI* adopts a similar design philosophy with the above work. In the *TI*, only data that are deemed essential for the queries are indexed in real-time, while the remaining data are processed in bulk and batch mode.

2.5 Summary

In this chapter, we first reviewed existing works on the large scale data management as a background of this thesis. We then reviewed the related works on multi-way join, real-time aggregation and real-time search query processing in existing large scale systems. In summary, the following limitations exist in these works:

1. For large scale multi-way join query processing, only rule based query optimizer has been implemented. A cost-based query optimizer can significantly improve the performance of multi-way join. In addition, the existing plan iteration algorithms for cost-based optimization were only designed for the centralized DBMSs. A more adequate algorithm can further improve the effective of the cost based optimizer and reduce the execution time of the query.

2. In existing large scan query processing, the OLTP query and OLAP query are usually processed in separate systems. The data stored in OLTP module are periodically exported to OLAP module, and the freshness of the OLAP results become an issue. Though the idea of supporting real-time query processing has been studied in traditional DBMSs, it is still a difficult problem in distributed environment. The distributed streaming systems such as S4 tries to return timely results to the users, but they assume that the new tuples are only appended to the data. Our work considers the scenario where the existing data might be updated.

3. Real-time search has become an important requirement of microblogging systems, but the existing ranking scheme offered by the microblogging vendors such as Twitter cannot return meaningful results. The search results are only sorted by the uploading time of each microblog. It would helpful if the ranking scheme considers the relationships between the microblogging users, and the reply/retweet relationships between the mi-

croblogs. Furthermore, it is also important to efficiently index the microblogs and rank the search results in a distributed environment.

In the rest of this thesis, we will first discuss the system overview of our proposed microblogging data management system, ART. We then show how ART addresses the above three challenges respectively.

CHAPTER 3

System Overview

With the development of social networking, the amount of data in the web is growing exponentially. Taking microblogging as an example, there are more than 115 million number of active twitter users every month, and million of tweets are published every day. The valuable information contained in these data and the challenges to manage these data have attracted many researchers' interest on designing the "big data" systems for microblogging. In this thesis, we propose ART, a full-functional, scalable and efficient microblogging data management system. It is capable of processing major queries required by a microblogging system and is optimized for three types of queries (multi-way join, aggregation and real-time search). In this chapter, we will discuss the design philosophy and architecture of ART in detail.

3.1 Design Philosophy of ART

ART is designed to support the following features:

1. Functionality. First, ART must be a full-functional system that is able to process all the fundamental queries required by a microblogging system. In a microblogging system, there are mainly two groups of queries: (1) the user queries such as the OLTP queries (update, insert, delete) and real-time search query; (2) the data analysis queries (including offline analytics and real-time analytics) that are issued by the system administrators. We

design ART to support the above two groups of queries so that it can be directly used as a back-end microblogging data management system without further extension.

2. Modularity. As ART is required to support various queries, it is not feasible to implement all these functionalities within one module. Based on the query types, we divide a ART into three modules. The first one is the OLTP module that is responsible for processing the OLTP queries issued by the users. To enable real-time analytics, the latest updates caused by user actions must be reflected in the result of the real-time analysis queries. Thus, the real-time analysis queries are also handled by this module. The second one is the offline analytics module, which processes the analysis queries on the data that are periodically loaded from the OLTP module. The third module is real-time search module that maintains a real-time inverted index to serve the real-time search query. To ensure that the processing logic inside each module is independent of the implementations of other modules, the higher level modules can only load data from the lower level system through the data loading API.

3. Scalability. The most important requirement of ART is to scale up as the data volume increases. To ensure that ART has high scalability and to minimize the efforts on implementing ART, we extend the existing "big data" systems (such as Hadoop, HBase and Hive) to implement the modules of ART. These stable systems have already been widely used to provide scalable service, and ART can inherit the scalability feature of these systems as well.

4. Efficiency. In addition to the above features, we also optimize each module of ART so that they are more efficient than exiting systems. Specifically, the offline analytics module is optimized to improve the performance of multi-way join query, and the real-time analytics module is optimized to efficiently process the real-time aggregation query. For the real-time search query, ART offers a better ranking scheme than exiting method within an acceptable response time.

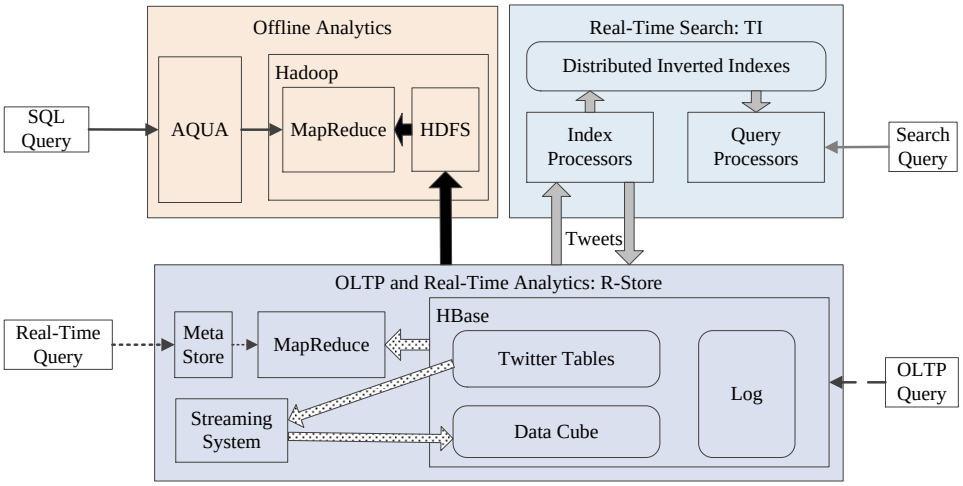


Figure 3.1: Architecture of ART

3.2 System Architecture

Figure 3.1 shows the system architecture of ART, which consists of three major modules.

1. Offline Analytics Module. The offline analytics module is responsible for analyzing the twitter data in a batch mode. In this module, the SQL queries are submitted to AQUA and are automatically translated into a sequence of MapReduce jobs based on the proposed cost-based optimizer. These jobs are submitted to Hadoop, the most popular open-sourced MapReduce implementation, for execution. Although Hadoop can be deployed on different storage systems, the released Hadoop package includes HDFS as the default storage system. In ART, the Twitter data stored in the low level storage module are periodically exported to HDFS, and the MapReduce jobs translated from AQUA are executed to analyze these data offline.

2. OLTP and Real-Time Analytics Module. R-Store is the low level distributed storage system implemented by extending HBase. The OLTP transactions caused by the user actions, such as publishing a tweet and updating status, are submitted to HBase directly. The Twitter tables stored in HBase are then updated accordingly. In addition, R-Store is designed to support real-time data analytics so that the latest updates of the data can be considered during the processing of the data analysis

queries. Specifically, we enable the real-time aggregation query processing through the following modifications: (1) We embed a streaming module in R-Store, which maintains a real-time data cube by the update streams from HBase. This streaming module also periodically materializes the real-time data cube into HBase. (2) The real-time aggregation queries, which are written by our defined data cube operators, are submitted to the MetaStore inside R-Store. The MetaStore translates these data cube operators into MapReduce jobs. (3) Instead of loading the data from HDFS, these MapReduce jobs directly scan the data stored in HBase in order to return the real-time results.

3. Real-Time Search Module. TI is the distributed real-time search engine for microblogs (tweets). The tweets are streamed from R-Store to the index processors inside TI, and the index processors index the tweets that have high probability to be searched in real-time. The tweets that are not indexed immediately are written to a log in R-Store and will be indexed in batch later. The query processors will use the distributed inverted indexes to serve the real-time search queries.

In general, as shown in Figure 3.1, the system can handle four types of queries, and thus four query and data flows exist in the system.

1. OLTP query. The OLTP queries are directly submitted to HBase (the dashed line arrow), and the twitter data are updated accordingly.

2. Offline analysis query. As shown by the black line arrows in the figure, the offline analysis queries written by a SQL-like language (including the multi-way join queries) are issued to AQUA, which translates the queries into MapReduce jobs and submits the jobs to Hadoop for execution. These queries analyze the data exported from R-Store to HDFS (the black rectangle arrows show the data flow of SQL queries).

3. Real-time query. The real-time queries are submitted to R-Store and translated to MapReduce jobs as well (the dotted line arrow). Different from the SQL queries, these MapReduce jobs directly process the data stored in HBase based on our implemented scan interface (the dotted rectangle arrow shows the data flow inside R-Store).

4. Search query. The search queries are submitted to TI (the grey line arrow), which updates the distributed inverted index based on the tweet streams transmitted from R-Store (the grey rectangle arrow).

In this chapter, we have presented the design philosophy and the system architecture of ART, which consists of three modules: offline analytics, OLTP and real-time analytics, and real-time search modules. In the rest of this thesis, we will discuss the details of these modules (since we didn't modify MapReduce framework itself, for the offline analytics module we only focus on AQUA, the cost-based query optimizer on top of MapReduce).

CHAPTER 4

AQUA: Cost-based Query Optimization on MapReduce

MapReduce has been widely recognized as an efficient tool for large-scale data analytics. It achieves high performance by exploiting parallelism among processing nodes while providing a simple interface for upper-layer applications. Some microbloging vendors, including Twitter, Google and Facebook, have enhanced their data management systems by integrating MapReduce into the systems. In ART, we also adopt MapReduce as the parallel processing engine to handle the offline data analysis workloads. However, existing MapReduce-based query processing systems, such as Hive, fall short of the query optimization competency of conventional database systems. Given an SQL query, Hive translates the query into a set of MapReduce jobs sentence by sentence. This design assumes that the user can optimize his query before submitting it to the system. Unfortunately, manual query optimization is time consuming and difficult, even to an experienced database user or administrator. In ART, to improve the performance of multi-way join queries, we propose a query optimization scheme for MapReduce-based processing systems. Specifically, we embed into Hive a query optimizer which is designed to generate an efficient query plan based on our proposed cost model. Experiments carried out on our in-house cluster confirm the effectiveness of our query optimizer.

4.1 Introduction

MapReduce [44] has been widely used as an large scale data processing platform. It achieves high performance by exploiting parallelism among a set of nodes. Massively Parallel Processing (MPP) data warehouse systems, such as Aster [10] and Greenplum [12], have recently integrated MapReduce into their systems. Experiments in [49] show that combining MapReduce and data warehouse systems produces better performance. Besides efficiency, MapReduce simplifies the deployment of MPP systems by providing two user-friendly interfaces: *map* and *reduce*. Applications implemented through the extension of the framework are naturally parallelizable and fault-tolerant.

To build applications on MapReduce, users must transform and code them as customized *map* and *reduce* functions. One major weakness of MapReduce is its lack of high-level declarative languages. In comparison, SQL, which is supported by most DBMSs, hides implementation details (e.g., access method and plan optimization), thereby simplifying application programming. Recently, some high-level languages have been proposed for MapReduce, such as Pig [83] and Hive [101]. These languages resemble SQL in many ways and are thus familiar to database users. Given a query, they automatically transform the query into a set of MapReduce jobs. Compared to the original MapReduce system, such systems are more suited for MPP data warehousing applications. Users can leverage them to process their data without having to model their application as a sequence of MapReduce operators.

Although the syntax and grammar of these systems are similar to SQL, such systems interpret declarative queries procedurally and strictly follow the processing logic specified by users in generating the corresponding map and reduce operations [9, 83]. For example, consider the following Hive query for the TPC-H [14] schema:

```
SELECT avg(quantity), avg(totalprice), nationkey
FROM (
    SELECT temp.quantity, temp.totalprice, c.nationkey
    FROM (
        SELECT l.quantity, o.totalprice, o.custkey
        FROM lineitem l JOIN orders o
        ON (l.orderkey=o.orderkey)
```

```
    ) temp JOIN customer c ON (temp.custkey=c.custkey)
) finaltable GROUP BY nationkey
```

There are three candidate query plans: P_1, P_2 and P_3. P_1 is the default plan of Hive, and it translates the query into three MapReduce jobs. The first job processes $temp = lineitem \bowtie orders$; the second job handles $finaltable = temp \bowtie customer$; and the third job computes the aggregation results for table $finaltable$. P_1 is an inefficient plan, as its first job generates a large intermediate table $temp$ [1], which will be written back to HDFS and read by the second job. To avoid high I/O costs, P_2 changes the orders of jobs. Its first job performs $customer \bowtie orders$ and the join operation involving table $lineitem$ is delayed to the second job. The third job of P_2 is similar to P_1's last job, where the aggregation result is computed. Unlike P_1 and P_2, P_3, applies the replicated hash join scheme [18] and only one MapReduce job is required to process $lineitem \bowtie orders \bowtie customer$. It reduces the overhead of initializing MapReduce jobs. However, it incurs more shuffling costs, as data need to be replicated among the reducers. Therefore, depending on the data distribution, P_3 may be superior to P_1 and P_2.

As has been well recognized in conventional query processing, good plans can indeed improve query performance by orders of magnitude. Current systems, such as Pig [83] and Hive [101], require users to translate SQL queries into their languages manually. The translation process, in fact, defines a specific query plan. However, users may not have sufficient knowledge to provide a good plan. Therefore, as in conventional database systems, a query optimizer is needed to produce near-optimal query execution plans. In this thesis, we propose *AQUA* (Automatic QUery Analyzer), a query optimization method designed for MapReduce-based MPP systems. Based on our experience of query processing in Hive, we find that the performance bottleneck of a MapReduce-based system is the cost of saving intermediate results. In MapReduce systems, to provide fine-grained fault tolerance, the results of each job are flushed back to the DFS (Distributed File System) as a backup. The consecutive job reads results of the previous job to continue with the processing. The I/O cost of DFS is significantly higher than that of the local storage system as network cost is incurred and multiple replicas are usually kept. An efficient MapReduce

[1] This is because *lineitem* and *orders* are the two largest tables in TPC-H. Also, each tuple of *orders* can join with four tuples of *lineitem*.

query plan should therefore avoid generating too many intermediate results.

To address the above requirement, *AQUA* adopts a two-phase query optimizer. In phase 1, the user's query is parsed into a join graph, based on which we adaptively group the join operators. Each group may contain more than one join operator, which will be evaluated by a single MapReduce job. In this way, the total number of MapReduce jobs and the intermediate results that need to be written back to DFS are reduced. In phase 2, the intermediate results of groups are joined together to generate the final query results. We examine all plausibly good plans and select the one that minimizes processing cost. The second phase is similar to a conventional cost-based query optimizer in DBMS.

To facilitate our cost estimation, we design a cost model to analyze relational operators in MapReduce jobs. Just as in traditional query optimization, the system maintains statistics about the underlying database to enable the optimizer to estimate the cost of various query plans. After a plan is selected, the expression tree is changed adaptively and translated into a set of MapReduce jobs.

We believe that ours is the first work that systematically explores how query optimization can be seamlessly embedded into a MapReduce system. The specific contributions of AQUA include:

1. Design and implementation of an efficient and novel optimizer tailored for the MapReduce framework. The optimizer identifies and exploits a variety of characteristics of the MapReduce framework to improve query performance.

2. An adaptive replicated join scheme to reduce I/O cost and MapReduce initialization cost. Based on the cost estimation, join operators are organized into several groups and one MapReduce job is created for each group.

3. A heuristics plan generator to reduce the cost of query optimization. The heuristics generator avoids plans that are obviously bad as early as possible and adopts shared scan to improve the performance.

4. Extensive experiments on our in-house cluster show that AQUA produces more efficient query plans than Hive [101] and Pig [83].

We note that while our implementation is currently based on Hive, our approach can be applied to other MapReduce-based MPP systems [17, 38] as well.

The rest of this chapter is organized as follows: Section 4.2 formalizes the optimization problem and discusses two join algorithms. The details of our query optimizer is presented in Section 4.3. In Section 4.4, we introduce our cost-model, designed for the relational operators in MapReduce. We evaluate the performance of our proposed approach in Section 4.5. We summarize this chapter in Section 4.6.

4.2 Background

4.2.1 Join Algorithms in MapReduce

The default join algorithms in Hive are map-side join and symmetric hash join. Suppose we are processing $T_i \bowtie_{T_i.k_1=T_j.k_2} T_j$, map-side join can be applied if 1) T_i or T_j is small in size and can be fully cached in memory; or 2) T_i and T_j are co-partitioned by k_1 and k_2. In the first case, the *mappers* fully load the small table (suppose it is T_i) into memory and scan the other table T_j. For every incoming tuple of T_j, we perform an in-memory hash join with T_i. After the whole table has been scanned, we get the complete join results. In the second case, each *mapper* loads a co-partition of T_i and T_j and performs a local symmetric hash join. As the tuples that can be joined together reside in the same co-partition, the *mappers* can process the join individually.

If map-side join cannot be applied, the distributed symmetric hash join is used instead. In particular, a hash function h is defined for all *mappers* and *reducers*. In the *map* phase, each *mapper* reads a data chunk of either T_i or T_j. And it generates keys as $h(T_i.k_1)$ or $h(T_j.k_2)$. In this way, all joinable tuples are shuffled to the same *reducer*, where an in-memory hash join is used to generate the final results.

In default join algorithms, one MapReduce job is created for a specific join operator. This strategy may incur high I/O costs for queries involving multiple joins, as the intermediate join results are written back to the DFS and subsequently read out by the next job. To reduce the I/O costs, in [19], a replicated join algorithm is proposed. Given a query $Q = T_1 \bowtie T_2 \bowtie ... \bowtie T_n$, let $\mathcal{A}$ denote the set of join attributes. Namely, if $T_i \bowtie_{T_i.a_x=T_j.a_y} T_j$, $a_x \in \mathcal{A} \wedge a_y \in \mathcal{A}$.

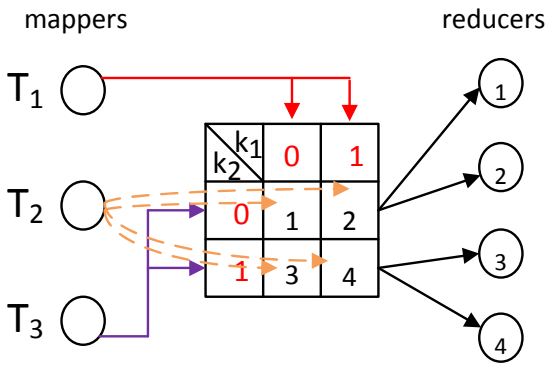


Figure 4.1: Replicated Join

In that case, we consider the two attributes, a_x and a_y, as equivalent attribute and only keep one copy in $\mathcal{A}$. In the replicated join algorithm, we create n types of *mappers*, one for each table. In particular, type-i *mappers* scan table T_i and shuffle the data to *reducers* adaptively.

Suppose we have m *reducers* and $|\mathcal{A}| = k$. To enable replicated join, we set $m = c_1 \times c_2 \times ... \times c_k$, where c_x is an integer, denoting the number of reducers for attribute a_x in $\mathcal{A}$. In *mappers*, we generate a set of composite keys for each tuple. The composite key follows the format of $< v_1, v_2, ..., v_k >$, where v_x is generated for attribute a_x in $\mathcal{A}$. The composite keys are generated in the *partition* function of the *map* phase.

Algorithm 4.1 shows the details of partition function in replicated join. In line 1, we initialize the key set to contain one random key. And then, we iterate all join attributes in $\mathcal{A}$. Suppose the next attribute is a_i. If a_i is an attribute of the tuple t, we set the ith values in current keys to $hash(t.a_i)\%c_i$ (line 4 to 7), where *hash* is a predefined hash function. Otherwise, for each existing key, we extend it to c_i composite keys by varying the ith values from 0 to c_i-1 (line 9 to 15). When all join attributes are processed, we use the key set to shuffle the tuple to multiple *reducers*.

The value of c_i affects the performance of replicated join. For an attribute of a large table, we need to assign more *reducers*, as more tuples need to be processed. In [19], a sophisticated model is applied to estimate the optimal assignment of *reducers*. In this chapter, to reduce the overhead of query optimization, a heuristic approach is adopted. Suppose attribute a_x belongs to table T_i, we define function $f(a_x)$ to return the size of T_i. Given two attributes

a_x and a_y, we assign c_x and c_y *reducers* for them respectively, where $\frac{c_x}{c_y} = \frac{f(a_x)}{f(a_y)}$. Namely, the number of reducers for an attribute is proportional to the size of the corresponding table.

Algorithm 4.1: Partition Function of Replicated Join

```
   input: AttributeSet A, Tuple t
 1 KeySet S ← initial();
 2 for i = 0 to |A|-1 do
 3     Attribute a_i ← A.nextAttribute();
 4     if a_i is an attribute of t then
 5         for j = 0 to S.size-1 do
 6             Key key_j ← S.nextKey();
 7             key_j.v_i ← hash(t.a_i)%c_i;
 8     else
 9         KeySet S' ← initial();
10         for j = 0 to S.size-1 do
11             Key newkey ← S.nextKey();
12             newkey.v_i ← x;
13             S'.add(newkey);
```

Figure 4.1 shows an example of processing query $T_1 \bowtie_{T_1.k_1=T_2.k_1} T_2 \bowtie_{T_2.k_2=T_3.k_2} T_3$. Suppose we have three *mappers* and four *reducers*. Each *mapper* responds for scanning a data chunk of a specific table. We have two join attributes, k_1 and k_2. Suppose $c_1 = c_2 = 2$. Given a value of k_1 or k_2, the predefined hash function will map it to 0 or 1. For a tuple t of T_1, we will generate two composite keys, $< hash(t.k_1)\%2, 0 >$ and $< hash(t.k_1)\%2, 1 >$. Similarly, we also generate two composite keys, $< 0, hash(t'.k_2)\%2 >$ and $< 1, hash(t'.k_2)\%2 >$, for a tuple t' of T_3. However, only one composite key $(< hash(t''.k_1), hash(t''.k_2) >)$ is created for a tuple t'' of T_2, as T_2 contains both join attributes. In this way, each tuple of T_1 or T_3 will be shuffled to two *reducers*. And all *reducers* can process their local joins individually.

Compared to the default join algorithms in Hive, the replicated join algorithm reduces the I/O costs by avoiding writing intermediate results to the DFS (in our implementation, we use HDFS). But it also incurs more shuffling costs by forwarding a tuple to multiple *reducers*. In our optimizer, join operators are grouped adaptively and one replicated join job is generated for each group.

4.2.2 Query Optimization in MapReduce

The intuition of *AQUA* is to adjust a query plan to improve the performance of large-scale data analysis jobs in MapReduce. In *AQUA*, we use *Query Plan* to denote a sequence of MapReduce jobs. These jobs are used to process a single SQL-like query.

Definition Query Plan
Given a query Q in SQL-like format, the query plan is a set of MapReduce jobs $P = \{j_0, j_1, ..., j_{k-1}\}$. j_i *is submitted to the processing engine after* j_{i-1} *completes. And after* j_{k-1} *is processed, the final results of Q are cached in the DFS.*

Given a query, different query plans may use different numbers of MapReduce jobs. To measure the efficiency of a query plan, we define the cost of a query plan as the sum of all its jobs' costs. Let $C(P)$ and $C(j_i)$ denote the costs of plan P and job j_i, respectively. We have:

$$C(P) = \sum_{i=0}^{k-1} C(j_i)$$

Definition Query Optimization
Given a query Q, the query optimization problem is to find a query plan with least cost. Namely, the optimizer needs to return a sequence of MapReduce jobs $\{j_0, j_1, ..., j_{k-1}\}$, *where* $\sum_{i=0}^{k-1} C(j_i)$ *is minimized among all valid plans.*

To improve the accuracy of estimation, some pre-computed histograms are built and maintained in the DFS (HDFS in our implementation). We propose a cost model to estimate the efficiency of a query plan and build an optimizer on top of Hive to select a near-optimal plan.

4.3 Query Optimization

AQUA's optimizer is a two-phase optimizer. In the first phase, the optimizer partitions the tables into join groups. Each join group is processed by a single MapReduce job. In the second phase, the optimizer searches for the best plan

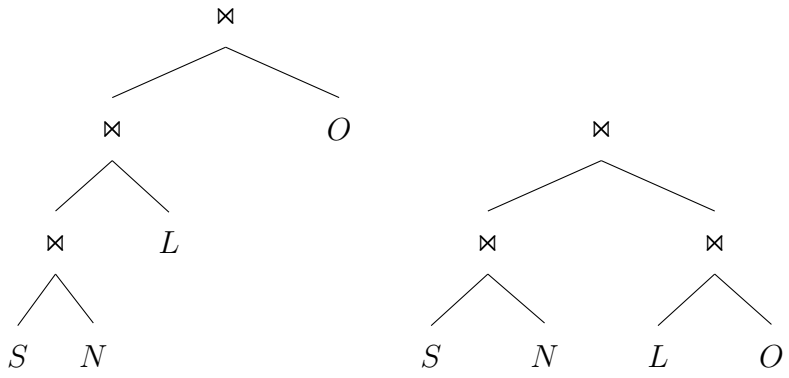
to generate the final results by combining the join groups. In this section, we present the details of our query optimizer and the cost model is discussed in the next section.

4.3.1 Plan Iteration Algorithm

As mentioned earlier, phase 1 partitions the join tables into groups, and decides whether the replicated join should be applied to the sub-groups. Each sub-group has two different implementations: the first implementation is a sequence of euqi-join (symmetric hash join) operators, which can be represented by a binary join tree; the second one is the replicated join algorithm. To select a better implementation for each sub-group, both the cost of binary join tree (implemented by symmetric hash join) and replicated join are estimated. However, for the former implementation, a plan iteration algorithm is required to iterate the possible plans (join trees) for the sub-group and select the one with lowest cost. After that, the cost of this best binary join tree is compared with the cost of the replicated join, and the implementation with fewer cost is assigned to the sub-group.

In phase 2, this plan iteration algorithm is applied to the join tree (each tree node represents a sub-group generated in phase 1), and the final hybrid join tree with lowest cost is outputted by our query optimization algorithm. In this section, we discuss the plan generation algorithm first, since it will be used in both phase 1 and phase 2.

In our plan generation algorithm, we consider both left-deep and bushy plans. As a matter of fact, in [48], Franklin et. al show the bushy plan is always the best plan in the distributed environment, and bushy plans are also used in parallel database systems [37]. We observe that MapReduce systems, by design, are more amenable to bushy query plan optimization. However, iterating all query plans incur too much overhead. Therefore, a heuristic approach is employed to prune the search space. The intuition here is similar to the query optimizer in conventional databases, namely, avoiding bad plans instead of searching for the optimal one. In the following discussion, we show the general ideas of how to iterate query plans and how to prune the search space.



(a) Left Deep Plan (b) Bushy Plan

Figure 4.2: Join Plans

Left-Deep VS Bushy Plans

As the plan space is extremely large for a complex query, most relational database systems only consider the left-deep plan in query optimization [32]. This strategy works well in many real applications. However, it may lead to an inferior plan for MapReduce-based query processing. This is because a MapReduce job needs to materialize the internal results of sub-queries.

In a conventional DBMS, the left-deep plan is preferred because it simplifies pipeline processing as at least one data source is the raw table. In Figure 4.2(a), after a result is produced for $S \bowtie N$, it is pushed to the next operator to join with L. In contrast, data sources in the last join of bushy plan (Figure 4.2(b)) are both internal results. Without fully materializing the internal results, it is difficult to provide the correct results.

A significant difference between MapReduce-based query processing and the traditional query processing is that a MapReduce job will materialize its outputs in the DFS for fault tolerance (In [41], MapReduce is extended to support pipelining between the *mappers* and *reducers*. However, it significantly complicates the failure recovery mechanism and provides marginal performance improvement for batch-based processing). For example, in the left-deep plan, a MapReduce job is used to perform $S \bowtie N$, and the results are written back to HDFS after the job is done. Then, a second job is initiated to join the results of the first job with L. After the second job is done, the results of $S \bowtie N \bowtie L$ are written back to HDFS. Namely, the internal results are written

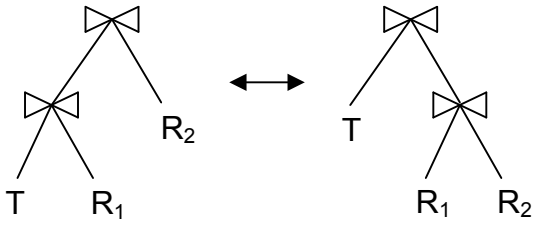


Figure 4.3: Basic Tree Transformation

back to HDFS in the previous MapReduce job and will be directly retrieved from HDFS in the subsequent job. As a matter of fact, HDFS I/O dominates the cost of processing a query. If a large number of results are generated by the intermediate MapReduce jobs, the cost of the plan would be high due to the high HDFS I/Os.

In the left-deep plan, we need to write and read the results of $S \bowtie N$ and $S \bowtie N \bowtie L$, while in the bushy plan, we need to write and read the results of $S \bowtie N$ and $L \bowtie O$. In most cases, we can determine which plan is better by comparing the sizes of $S \bowtie N \bowtie L$ and $L \bowtie O$.

Pruning of Optimization Space

We apply a recursive algorithm to iterate all possible query plans. Figure 4.3 shows two basic plan variants. Suppose T represents a sub-plan. The left plan denotes a left-deep plan while the right plan is a right-deep plan. Actually, if $T = R_3 \bowtie R_4$, the right plan becomes a bushy plan. The recursive algorithm works from the bottom to the top. It first iterates all possible sub-plans, and then for each sub-plan, it tries the left-deep and right-deep combinations.

In our query optimizer, we also support bushy plans and this results in a larger optimization space. Therefore, we apply heuristics to reduce the search space and prune inefficient plans as early as possible. The idea of the pruning approach is summarized as follows:

1. We do not generate equivalent sub-plans. For example, plan $R_1 \bowtie R_2$ is equivalent to plan $R_2 \bowtie R_1$ and plan $R_1 \bowtie (R_2 \bowtie R_3)$ is equivalent to plan $(R_2 \bowtie R_3) \bowtie R_1$. For equivalent sub-plans, we only select one to expand in our recursive algorithm.

2. We prune inefficient plans as early as possible. For example, if $R_1 \bowtie R_2$

generates significantly more (e.g., an order of magnitude more) results than $R_2 \bowtie R_3$, we remove $(R_1 \bowtie R_2) \bowtie R_3$ from the sub-plan set. This can be done by a rough estimation based on the corresponding histograms. In this way, the less effective sub-plan will not appear in the final query plan.

3. We avoid the "low-utility plan". The performance gain of MapReduce comes mainly from parallelism. However, some query plans contradict this principle. As an example, plan $((lineitem \bowtie orders) \bowtie customer) \bowtie nation$ is not a good plan because *customer* joins *nation* on *nationkey*, and there are in total 25 distinct *nationkey* in the TPC-H schema. If we have more than 25 reducers available, the above plan cannot fully exploit them. We call such a plan a "low-utility plan". Low-utility plans inevitably incur significant performance penalty. Therefore, the query optimizer needs to avoid such plans. When building histograms, we also record the number of unique values in each bucket, and based on which, we can estimate the maximal number of usable reducers.

Query Plan Iteration Algorithms

Algorithm 4.2 shows the pseudo code of our query plan generator. The query plan generator tries to transform the expression tree of the query to generate all possible plans. The input parameter is the root node of the expression tree. If the expression tree node has left child or right child, we first try to generate variants of the subtrees (line 2-5). Then, for each pair of variants, we generate an expression tree, which denotes a possible plan (line 8). The plan denoted by the expression tree is then pruned by the heuristic algorithm. To iterate all possible plans, the basic transformation in Figure 4.3 is performed for the tree (line 12). The operators in the left and right sub-trees may be exchanged with each other, which results in a new tree. And the variants will be added to the result (line 13-20). After Algorithm 4.3 returns, we apply the histograms to estimate the pruning selectivity of each plan, and this selectivity will be used to estimate the cost of that plan. At last, the most optimized one will be selected by the algorithm.

Algorithm 4.3 shows the basic idea of the heuristic pruning algorithm. First, we check whether the generated plan is actually equivalent to an existing one

Algorithm 4.2: Iterative Generator

```
   input: ExpressionTreeNode curOp
 1 Vector result = NULL;
 2 if currentOp.leftchild≠ NULL then
 3 |   Array l_variant = IterativeGenerator(curOp.leftchild);
 4 if currentOp.rightchild≠ NULL then
 5 |   Array r_variant= IterativeGenerator(curOp.rightchild);
 6 for i=0 to l_variant.size() do
 7 |   for j=0 to r_variant.size() do
 8 |   |   ExpressionTree tree = NewTree(curOp, l_variant.get(i),
   |   |   r_variant.get(j));
 9 |   |   if HeuristicPruning(tree) then
10 |   |   |   continue;
11 |   |   result.add(tree);
12 |   |   tree=basicTransformation(tree);
13 |   |   Array l_variant'= IterativeGenerator(tree.root.leftchild);
14 |   |   Array r_variant'= IterativeGenerator(tree.root.rightchild);
15 |   |   for x=0 to l_variant'.size() do
16 |   |   |   for y=0 to r_variant'.size() do
17 |   |   |   |   ExpressionTree tree' = NewTree(tree.root,
   |   |   |   |   l_variant'.get(x), r_variant'.get(y));
18 |   |   |   |   if !HeuristicPruning(tree) then
19 |   |   |   |   |   result.add(tree');
20 return result;
```

Algorithm 4.3: Heuristic Pruning

```
   input: ExpressionTree tree
 1 if tree is equivalent to an existing plan then
 2 |   return true;
 3 else
 4 |   if tree.root is an operator that cannot exploit all reducers then
 5 |   |   if tree have equivalent transformation then
 6 |   |   |   return true;
 7 |   else
 8 |   |   R=tree.root.getLeftTable();
 9 |   |   S=tree.root.getRightTable();
10 |   |   size=estimatedSizeOf(R ⋈ S);
11 |   |   size1=estimatedMinSizeOf(R, getJoinableTable(R)-{S});
12 |   |   size2=estimatedMinSizeOf(S, getJoinableTable(S)-{R});
13 |   |   if size/size1 > θ or size/size2 > θ then
14 |   |   |   return true;
15 return false;
```

(line 1 and 2). Then, if the root operator cannot exploit all possible reducers, which means that this candidate plan has a poor resource utilization of the MapReduce cluster, we discard the plan (line 4-6). Finally, we estimate the size of intermediate results of the root operator (line 8-14). The function $estimateSizeOf$ estimates the size of intermediate join result for R and S (the result of this function is assigned to the variable $size$). Then, the $getJoinableTable()$ function returns the tables that can be joined with the sub-tree R in the query. The $estimatedMinSize$ checks the alternative plans by joining R with any of the table in $getJoinableTable(R) - S$, and returns the one with minimal intermediate size. If $size$ is far larger than $size1$ or $size2$ (line 13), we just prune the current plan as it has high probability to generate a join tree with high cost. θ is a predefined threshold, which controls the tradeoff between optimization cost and accuracy.

4.3.2 Phase 1: Selecting Join Strategy

As mentioned before, the replicated join may lead to a better performance by reducing I/O costs in HDFS. But given a query involving multiple joins, the optimizer needs to figure out when and how to use the replicated join. In [19], all joins are grouped together and a single MapReduce job is used to process

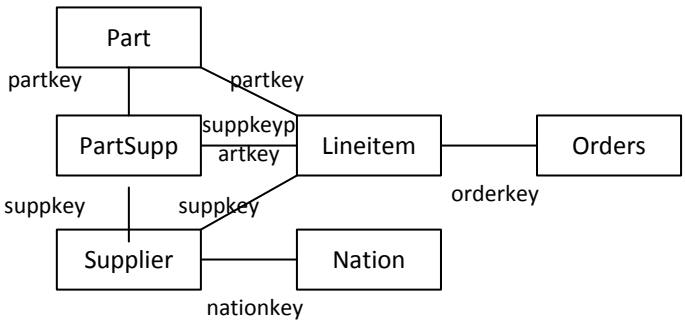


Figure 4.4: Joining Graph For TPC-H Q9

the query. This is not always the optimal solution, as replicated join increases the shuffling cost. In our optimizer, an adaptive join approach is proposed. To simplify the discussion, we define a joining graph for queries.

Definition Joining Graph
Given a query Q, its joining graph is defined as $G_Q = (V, E)$, where

- If table T_i is involved in Q, we have a node n_i in V that denotes the table.
- If $T_i \bowtie_{T_i.k=T_j.k} T_j$ is a join operation in Q, we create an undirected edge $e = (n_i, n_j)$ and e's label is set as k.

Figure 4.4 shows the joining graph for TPC-H [14] Q9, where 6 tables are involved. The edge (*PartSupp*, *Lineitem*) is labeled as "*PartKey*, *SuppKey*", as the join is performed on two attributes.

Each possible join strategy can be represented as a covering set of the graph, which is defined as:

Definition Covering Set of Joining Graph
Given a joining graph $G_Q = (V, E)$, a covering set ($\mathcal{S}$) of that graph is a set of sub-graphs, satisfying:

- $\forall G_i \in \mathcal{S}$, G_i is a sub-graph of G_Q. Namely, given a node n_x in G_i and an edge e_y in G_i, $n_x \in V$ and $e_y \in E$.
- $\forall G_i \in \mathcal{S}$, if n_x and n_y are two nodes of G_i, there must be a path in G_i that connects n_x with n_y.
- $G_Q.V = \bigcup_{\forall G_i \in \mathcal{S}} G_i.V$.

- $\forall G_i, G_j \in \mathcal{S} \rightarrow G_i.V \cap G_j.V = \emptyset$. Namely, subgraphs do not share a common node.

Based on the definition, all the nodes in the joining graph are included in the covering set, while only a portion of edges are selected. The remaining edges, in fact, define the join operations between sub-graphs in the covering set. There is a special covering set $\mathcal{S}_0$, where $\forall G_i \in \mathcal{S}_0$, $|G_i.V| = 1$ (we use $|A|$ to denote the number of elements in a set A) and $G_i.E = \emptyset$. $\mathcal{S}_0$ is used as the initial state of our query optimization.

For a sub-graph G_i in the covering set $\mathcal{S}$, depending on its node number, we have the following join strategies. If $|G_i.V| = 1$, no join is defined. If $|G_i.V| = 2$, the default symmetric hash join is used. Otherwise, if $|G_i.V| > 2$, we adaptively adopt the replicated join or symmetric hash join for G_i. When $|G_i.V| > 2$, we define the cost saving as:

$$C_s(G_i) = C_{hjoin}(G_i) - C_{rjoin}(G_i)$$

where $C_{rjoin}(G_i)$ denotes the cost of replicated join for G_i and $C_{hjoin}(G_i)$ is the estimated costs of using symmetric hash join. The plan iteration algorithm discussed in Section 4.3.1 is applied to find the optimal join plan using symmetric hash join. If $|G_i.V| <= 2$, the cost saving is defined as 0. The intuition is to select the plan with maximal cost savings.

In fact, we can iterate all possible covering sets by adaptively linking the sub-graphs.

Definition Graph Linking
Given two sub-graph G_i and G_j of G_Q, let $e = (n_x, n_y)$ be an edge in G_Q, satisfying $n_x \in G_i.V$ and $n_y \in G_j.V$. We can link G_i by G_j via e. The result is a new sub-graph G_{ij}, where $G_{ij}.V = G_i.V \cup G_j.V$ and $G_{ij}.E = G_i.E \cup G_j.E \cup \{e\}$.

By linking two graphs, we generate a new graph. For any two nodes in the graph, there is a path connecting the nodes. Algorithm 4.4 shows how to iterate all possible covering sets by linking graphs. The special covering set $\mathcal{S}_0$ is used as the initial state (line 1). Then, we iterate all possible combinations of picking i edges from the joining graph (line 4). For a specific combination, we can generate a joining plan, *temp*, which is initialized as $\mathcal{S}_0$. The selected edges are used to link sub-graphs in *temp* (line 7-13). Given a node and a plan, function

Algorithm 4.4: JoinPlans

```
   input: QueryGraph G_Q
1  S_0 ← createInitialState(G_Q);
2  PlanSet S_P ← ∅;
3  for i=1 to |G_Q.V| do
4      EdgeSets S_E ← getAllCombination(G_Q.E, i);
5      for ∀E ∈ S_E do
6          Plan temp ← S_0;
7          for ∀ edges e ∈ E do
8              Graph G_i ←getGraph(e.start, temp);
9              Graph G_j ←getGraph(e.end, temp);
10             if G_i ≠ G_j then
11                 Graph G_new ← link(G_i, G_j, e);
12                 temp.remove(G_i), temp.remove(G_j);
13                 temp.add(G_new);
14         S_P.add(temp);
15 return optimal plan in S_P;
```

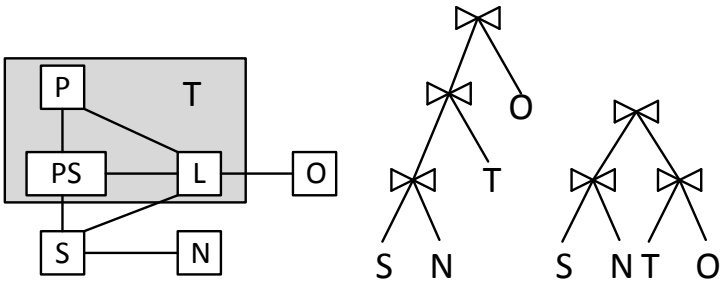
getGraph returns the subgraph containing the node. The resulting covering set is stored as a candidate plan (line 14). After all plans are generated, the one with maximal savings is selected as our join plan (line 15). Algorithm 4.4 searches for all possible plans. Therefore, the complexity is estimated as

$$cost = \sum_{i=1}^{C} \binom{C}{i} = 2^C - 1 \tag{4.1}$$

where $C = |G_Q.E|$. In most cases, only a few tables participate in a join and hence, C is a small value. We show the cost of query optimization in the experiments.

4.3.3 Phase 2: Generating Optimal Query Plan

In phase 1, the optimizer selectively groups some nodes into sub-graphs and generates a single MapReduce job to process each sub-graph. Figure 4.5(a) shows a possible result for Figure 4.4. To simplify the notation, we use L, O, N, P, PS, S to represent table *Lineitem*, *Orders*, *Nation*, *Part*, *PartSupp* and *Supplier*, respectively. In Figure 4.5(a), P, PS and L are put into a MapReduce job and we use T to denote the intermediate results. We need to



(a) Joining Strategy (b) Possible Plans

Figure 4.5: Plan Selection

join T with the remaining tables to generate the query results. Figure 4.5(b) lists two possible query plans, which have significantly different processing costs. Suppose the optimal covering set generated in phase 1 is $\mathcal{S}$, the optimizer needs to find an efficient query plan in phase 2 to join the sub-graphs in $\mathcal{S}$. For each G_i in $\mathcal{S}$, G_i denotes an input table in phase 2. If $|G_i.V| = 1$, the input table is a raw table. Otherwise, the input table is an intermediate result of a MapReduce job. Then, the query plan iteration algorithm discussed in Section 4.3.1 is applied to the join tree generated by phase 1, and the final hybrid plan (with both symmetric hash join and replicated join) is generated.

4.3.4 Query Plan Refinement

After a plan is selected as the execution plan, it is further refined by our optimizer to reduce the processing cost. Two approaches are applied in this stage, sharing table scans and submitting concurrent MapReduce jobs.

Sharing Table Scan in Map Phase

An inter-query sharing framework is proposed in [81], where queries with the same MapReduce jobs are grouped and processed together. In this work, three levels of sharing is possible: (1) Sharing scans only: the queries have the same source table while the lters and the aggregation attributes are different; (2) Sharing map output: the queries have the same source table and the same aggregation attributes (including both the grouping keys and the aggregated columns), while the lters are different; and (3) Sharing map functions: the main purpose of the map function here is to lter the tuples based on the predicates.

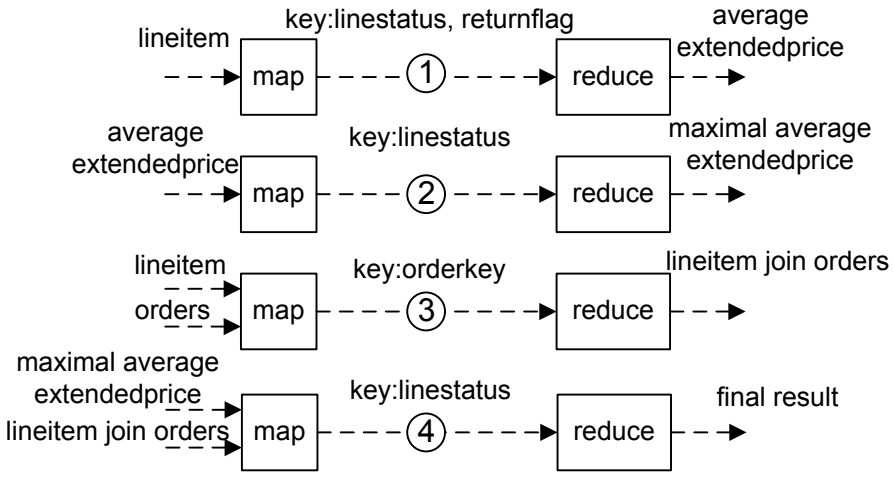


Figure 4.6: MapReduce Jobs of Query q_0

If the queries have the same source table and the same lters, the map functions can be shared between these queries.

The queries evaluated in MRShare are mainly aggregation queries, each of which is implemented by one MapReduce job. Different from this work, our work tries to exploit the possibility of sharing data inside one recursive query, which is implemented by several MapReduce jobs.

Considering the following query for the TPC-D schema:

q_0: SELECT l_0.extendedprice, o_0.shippriority
FROM lineitem as l_0, orders as o_0
WHERE l_0.orderkey = o_0.orderkey and l_0.extendedprice >
(SELECT max(avg(l_1.extendedprice))
FROM lineitem as l_1
WHERE l_0.linestatus = l_1.linestatus
GROUP BY l_1.returnflag)

lineitem appears in both the outer query and the inner subquery. The subquery is correlated with the outer query. In real systems, such queries are not uncommon. In the TPC-D benchmark, more than 60% queries contain at least one table with multiple instances.

Lacking an index, MapReduce scans the whole dataset when processing queries. Figure 4.6 shows the MapReduce jobs for q_0. In Figure 4.6, table *lineitem* is scanned twice, once for computing the average *extendedprice* and

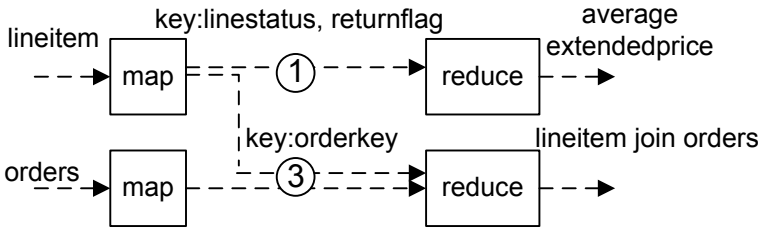


Figure 4.7: Shared Table Scan in Query q_0

another for joining with table *orders*. In the corresponding MapReduce jobs, mappers perform the same I/O operations, namely, loading tuples of *lineitem* from HDFS. If the results of the last scan can be reused, we avoid repeatedly reading the same table. Therefore, we propose a shared-scan approach to reduce I/O cost in consecutive MapReduce jobs.

The shared-scan approach generates all required key-value pairs in the first MapReduce job, which can be loaded by the subsequent jobs from HDFS. For q_0, the first MapReduce scans table *lineitem* and applies the composite key $(linestatus, returnflag)$ to generate the average *extendedprice*. To share the table scan, in the map phase, we also generate key-value pairs for the third MapReduce job. Namely, two key-value pairs, $((linestatus, returnflag), t)$ and $(orderkey, t)$, are created for each tuple t of *lineitem*. *((linestatus, returnflag), t)* is sent to the reducers for computing the average *extendedprice* while $(orderkey, t)$ is cached as a temporary file in HDFS. In the third MapReduce job, the mappers only scan table *orders* and the reducers load key-value pairs of *lineitem* from HDFS. Figure 4.7 shows the idea of sharing table scan in query q_0. In this way, we avoid repeatedly scanning table *lineitem*.

The same strategy can be applied to multiple queries, if they are being processed concurrently and share some common expressions. For example, many TPC-H queries have the sub-expression $lineitem \bowtie orders$. By sharing the common results between queries, we can significantly reduce the I/O costs. In our future work, we will examine how to combine multi-query optimizations into our system. Specifically, when sharing sub-query results is possible, a new query plan can be generated to exploit the features.

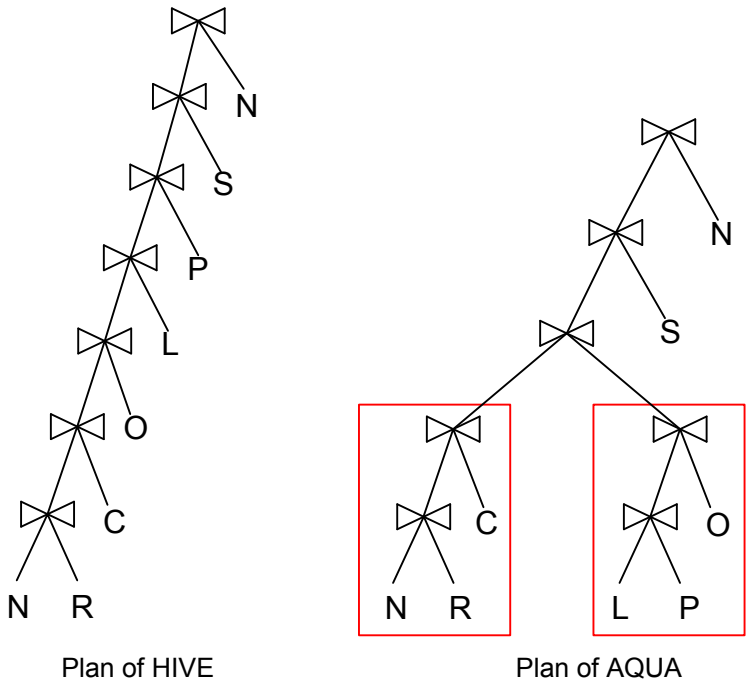


Figure 4.8: Optimized Plan for TPC-H Q8

4.3.5 An Optimization Example

In this section, we show a concrete example of how *AQUA*'s optimizer works. For comparison purpose, we use the query plans in Hive's Benchmark [61] as our baseline. Note that plans in [61] are not the default plans of Hive. Instead, they have been manually optimized to avoid ineffective plans.

In Figure 4.8, we show two possible plans for TPC-H Q8. The left plan is given by [61] and is a left-deep plan. It starts by joining the smallest tables to avoid high I/O costs. However, as each job can only perform a two-way join, it generates 8 MapReduce jobs (7 for joins and 1 for aggregation). Based on the observation of [63], the initialization cost of MapReduce job cannot be ignored and will increase as more nodes are involved. By transforming the query into 8 jobs, the left plan incurs a significant initialization cost and hence, is not cost-effective. The right plan is the plan adopted by *AQUA*. It generates 5 MapReduce jobs, among which two jobs are created for the replicated joins (e.g. $N \bowtie R \bowtie C$ and $L \bowtie P \bowtie O$), two jobs are used to do the two-way joins and one is used to compute the aggregation results. Compared to the left plan, *AQUA*'s plan has the following advantages:

- *AQUA* reduces the number of MapReduce jobs by using replicated joins.

- *AQUA* avoids generating large volumes of intermediate results by adopting replicated joins and considering bushy plans.
- *AQUA* adjusts the join sequences by using a cost-based optimizer.

The above advantages of *AQUA* are further verified by our experiments. A significant performance boost is observed for various types of queries.

4.3.6 Implementation Details

In our system, the plan is represented as an expression tree. The expression tree is forwarded to Hive's analyzer, which applies the metadata of tables to translate the tree into a set of MapReduce jobs. Those jobs (their java classes) are serialized into an XML file, which can be submitted to the process engine for processing.

Shared table scan is implemented by modifying the MapReduce jobs generated by Hive. First, we modify the job description of the first MapReduce job by replacing its key-value pairs with composite key-value pairs. Second, two new operators are implemented for Hive. One is designed for mappers to write back key-value pairs to HDFS and the other one is used in reducers to load key-value pairs from HDFS. Those operators are serialized and embedded into the original job description. When shared scan is applied, the cost model is modified with the inclusion of the cost of writing back key-value pairs to HDFS.

4.4 Cost Model

To evaluate the performance of a specific plan, we propose a cost model tailored for the MapReduce framework. For efficiency, the cost model applies some precomputed histograms to estimate the selectivity of predicates and joins. Before we present the details of our cost model, we first discuss how to efficiently build histograms in MapReduce framework.

4.4.1 Building Histogram

Given a table T, a special MapReduce job is submitted to build histograms for all its columns. Suppose $a_0, a_1, ..., a_{n-1}$ are columns of table T and $[l_i, u_i]$ is a_i's domain. We build an equal-width histogram for each column. Namely, we split

$[l_i, u_i]$ into K cells, and in each cell, we record the number of tuples within the cell and assume the data follow uniform distribution.

One naive approach to build a histogram is to apply n MapReduce jobs, one for each column. In the map phase, we scan the table and partition tuples according to their values in a specific column. In the reduce phase, each reducer generates a cell for the column's histogram. The cells are then inserted into HDFS. The query optimizer can ask HDFS to retrieve the whole histogram of the column. Although simple, the naive approach repeatedly scans a table in multiple MapReduce jobs, which actually can be avoided. In our approach, a single MapReduce job is used to build histograms for all columns within a table.

To build histograms on all the columns of a table using a single MapReduce job, we generate a *composite key* for each tuple in the map phase. Suppose we build a histogram with K equal-width buckets for column a_i. Let the domain of a_i be $[l_i, u_i]$. The jth bucket covers the range $[l_i + \frac{j(u_i-l_i)}{K}, l_i + \frac{(j+1)(u_i-l_i)}{K}]$. In the map phase, we generate a composite key for each tuple. Key-value pairs follow the format of $< (columnID, bucketID), 1 >$, where $columnID$ is the unique ID of the column and $bucketID$ is the bucket ID of the corresponding value. When comparing two keys, we first compare their $columnID$s and then the $bucketID$. Therefore, if the size of T is m, mappers actually generate $n \times m$ key-value pairs, where n is the number of columns involved in histogram building. To reduce shuffling cost, pre-aggregation is performed in the map phase. The *partition* function in the map phase is implemented as mapping data within the same bucket to the same reducer. We customize the *combiner* function to aggregate key-value pairs within the same bucket. In this way, each mapper only generates at most one key-value pair for a bucket, which reduces shuffling cost.

In the reduce phase, we classify key-value pairs by their $columnID$ and combine the results from multiple mappers. In the end, the metadata of a histogram bucket (table name, column name, bucket range and bucket value) are written back to HDFS. To efficiently locate a histogram, histograms are maintained as a directory tree in HDFS. The histogram for column a_i of table T is stored in "$/user/hive/histogram/T/a_i$".

Algorithm 4.5 and 4.6 illustrate the presudo code of building histograms. In Algorithm 4.5, we scan a table stored in HDFS. The tuples of the table are

stored as strings. Hence, we need to parse the string into individual attributes (line 1). For each attribute, we generate a key-value pair by using the attribute ID and its corresponding bucket ID as the composite key (lines 2-6). Given a value $data[i]$ of ith column, suppose $low[i]$ and $up[i]$ denote the column's domain, function $getBucketID$ returns the histogram bucket ID that the value falls in. In the reduce phase, we first retrieve the column ID and the bucket ID from the key (lines 1 and 2 in Algorithm 4.6). Then, the statistics from multiple mappers are combined together (lines 3 and 4). When the reduce phase completes, the histograms are written back to HDFS. Multiple reducers may write statistics about the same bucket. A file lock is applied to guarantee consistency. To reduce shuffling cost, before key-value pairs are shuffled to reducers, pre-aggregation is performed with the use of the same reduce function defined in Algorithm 4.6.

Algorithm 4.5: `map(Object` *key*`, Text` *value*`, Context` *context*`)`

```
1 //value: serialized string of a tuple
2 Object[] data = parse(value);
3 for i=0 to data.length do
4     if need to build histograms for column i then
5         int bucketID = getBucketID(i, data[i], low[i], up[i]);
6         CompositeKey newKey = new CompositeKey(i, bucketID);
7         context.collect(newkey, 1);
```

Algorithm 4.6: `reduce(Key` *key*`, Iterable` *values*`, Context` *context*`)`

```
1 int id = key.first();
2 int bucketID = key.second();
3 for IntWritable val : values do
4     histogram[id][bucketID]+= val; //combining the values from
      mappers
```

In current implementation, we build equal-width histograms for each column individually. Though simple, the histograms can provide good enough estimations for us to avoid obviously bad plans. Database systems, often use more complex histograms (e.g., V-optimal, maxdiff [86]) that provide better selectivity estimation. To build more sophisticated histograms, we can extend

Table 4.1: Parameters

Parameter	Definition
r_l	cost ratio of local disk reads
w_l	cost ratio of local disk writes
r_h	cost ratio of HDFS reads
w_h	cost ratio of HDFS writes
μ	cost ratio of Network I/O
ν	cost ratio of CPU computation
b	size of mapper's memory buffer
d	size of data chunk in HDFS
$\|T\|$	number of tuples in table T
$f(T)$	size of T's tuple (in bytes)
$g(T, S)$	join selectivity of table T and S

$getBucketID$ in the map phase and rewrite the combining algorithm in reducers. For example, to support MaxDiff histograms, two MapReduce jobs are generated. In first job, we partition the values into buckets of equal-length. If the final histogram composes of k buckets, in first job, we will generate ck buckets, where c is a constant. In this way, we generate more buckets than necessary. In second job, all buckets are sent to the same reducer for combining. The reducer applies a local MaxDiff algorithm to combine the small buckets into larger ones. When only k buckets left, the process terminates and the histogram is written back to HDFS. Techniques to map algorithms to construct such histograms to the MapReduce framework will be a significant deviation from the main contribution of this paper and hence are relegated to future work.

4.4.2 Evaluating Cost of MapReduce Job

After a query plan is transformed into a set of MapReduce jobs, we assume these MapReduce jobs are processed by the same set of nodes. In the cost model, we consider two types of costs: I/O costs (including local disk I/O and network I/O) and CPU costs. The total cost of processing a MapReduce job is used as the metric. Note that the cost model does not provide an accurate estimation. Instead, the approximate approach is applied to simplify computation. The intuition is to avoid bad plans instead of searching for the optimal one. Table 4.1 shows the parameters used in the analysis.

Basically, there are two types of MapReduce jobs: map-only jobs and map-reduce jobs. For single table *select* and *map-side join*, Hive creates a map-only job. For *join* and *aggregation* operations, a map-reduce job is generated. We handle them differently in the cost model.

Map-Only Jobs

In Hive, single table scan and map-side join are transformed into map-only jobs. These jobs can be processed by each mapper individually. Therefore, we do not need to consider the cost incurred during the reduce phase.

For a select query, if it only retrieves data from a single table T and does not perform aggregations, it can be processed by map-only jobs. To handle the select query, all tuples of table T are retrieved from HDFS, which incurs $|T|f(T)r_h$ cost. The predicates defined in the query are used as a filter to prune unqualified tuples. Only the necessary columns are output as results. Suppose the selectivity of the ith filter is α_i and there are k filters for table T, we use α ($\alpha = \prod_{i=1}^{k} \alpha_i$) to denote the accumulative selectivity. Let the projection selectivity be β (after ruling out unnecessary columns, the tuple size is reduced to $\beta \times 100\%$ of its original size). It costs $\alpha\beta|T|f(T)w_h$ I/O to write the results back to HDFS. α is estimated by histograms while β is computed based on the metadata of the table. For each input tuple, it is compared with k filters. Hence, the expected number of comparisons is

$$p(T,k) = \sum_{i=1}^{k}(i \times (1-\alpha_i) \times (\prod_{j=1}^{i-1} \alpha_j)) + k \times \prod_{i=1}^{k} \alpha_i \tag{4.2}$$

In summary, a single table select query incurs a cost of

$$C_{select} = |T|f(T)r_h + \alpha\beta|T|f(T)w_h + \nu p(T,k)|T| \tag{4.3}$$

In Hive, a map-side join can be used in two cases: 1) one table can be fully buffered in memory; and 2) both tables are partitioned by the join attribute. For example, if both *Lineitem* and *Orders* are partitioned by *orderkey*, we can apply the map-side join to process $Lineitem \bowtie_{Lineitem.orderkey=Orders.orderkey} Orders$. Suppose two tables T and S participate in a map-side join. If T can be fully buffered in memory, T will be read $n = \frac{|S|f(S)}{d}$ times, where n is the

number of mappers. In this case, the cost of the map-side join is estimated as:

$$\begin{aligned} C_{memory\_join} &= (\frac{|S|f(S)|T|f(T)}{d} + |S|f(S))r_h + \\ &\quad \alpha\alpha' g(T,S)|T||S|(\beta f(T) + \beta' f(S))w_h + \\ &\quad \nu(\alpha\alpha'|T||S| + p(T,k)|T| + p(S,k')|S|) \end{aligned} \tag{4.4}$$

where α, β, α' and β' denote the accumulative filter selectivity and projection selectivity of T and S respectively, and k and k' represent the number of predicates for T and S respectively. The first term gives the I/O cost of reading table T and S. The second term estimates the result size and the cost of writing back results to HDFS. The last term calculates CPU cost (composed of join cost and filter cost). On the other hand, if neither T nor S can be buffered in memory and both of them are partitioned based on the join attribute, we simply replace the first term of Equation 4.4 to $(|T|f(T) + |S|f(S))r_h$.

Map-Reduce Job

To process join or aggregations, a full MapReduce job is created in Hive. Compared to a map-only job, a map-reduce job is more costly as it triggers sort operations at both the map and reduce sides, and it shuffles data files between mappers and reducers.

Given an equal-join query $T \bowtie S$, suppose neither T nor S can be buffered in memory, and at least one table is not partitioned by the join attribute, a map-reduce job is established to process the query. In the map phase, the data of two tables are loaded from HDFS, which incurs $(|T|f(T)+|S|f(S))r_h$ cost. The input tuples are pruned via corresponding filters. If the tuple passes the filter, a key-value pair is generated and buffered in memory. We estimate the CPU cost to be $\nu(p(T,k)|T|+p(S,k')|S|)$, where k and k' are the numbers of predicates for table T and table S, respectively. Let α, β, α' and β' denote the accumulative filter selectivity and projection selectivity of T and S, respectively. Suppose the size of the key is about δ bytes. The sizes of key-value pairs are estimated as $\beta f(T)+\delta$ and $\beta' f(S)+\delta$ for table T and S, respectively. The total numbers of key-value pairs generated for T and S are $\alpha|T|$ and $\alpha'|S|$, respectively. When the memory buffer is full, the mapper applies quick-sort algorithms to sort the key-value pairs and writes them as a local file. After all key-value pairs have been generated, the local files are merged together. Suppose the size of the

memory buffer is b, there will be $x = \frac{b}{\beta f(T)+\delta}$ key-value pairs for T in the buffer, and the total size of key-value pairs of T is $y = \alpha|T|(\beta f(T) + \delta)$. We estimate the cost of sorting and merging for table T as:

$$C(T)_{sort} = \nu\alpha|T| \log_2 x + yw_l + y(w_l + r_l) \tag{4.5}$$

where the first term represents the quick-sort cost in the memory buffer, the second term denotes the I/O cost of flushing data from buffer to disk, and the last term is the I/O cost of merge-sort. Actually, mappers do not perform full merge-sort as each mapper only reads a data chunk. $C(T)_{sort}$ actually includes the sort cost in the reducer part. In the same way, we can estimate the sort cost of table S, $C(S)_{sort}$. Therefore, the total cost in the map phase is:

$$\begin{aligned} C_{map} &= (|T|f(T) + |S|f(S))r_h + \nu(p(T,k)|T| \\ &\quad + p(S,k')|S|) + C(T)_{sort} + C(S)_{sort} \end{aligned} \tag{4.6}$$

When the mapping phase completes, the reducers will pull data files from the mappers. The network cost is computed as

$$C_{shuffle} = \mu(\alpha|T|(\beta f(T) + \delta) + \alpha'|S|(\beta' f(S) + \delta)) \tag{4.7}$$

After that, a multi-way merge-sort is applied. As sorting cost has already been computed in the map phase, we do not consider it in the reduce phase. For tuples of the same key, an in-memory join is performed, and $z = \alpha\alpha'|T||S|g(T,S)$ results are generated. Each result refers to a comparison operation of the in-memory join. Therefore, the CPU cost of the in-memory join is estimated as $z\nu$. Finally, all the results are written back to HDFS, which incurs $z(\beta f(T) + \beta' f(S))w_h$ cost. In summary, the total cost in the reduce phase is:

$$C_{reduce} = C_{shuffle} + z(\nu + (\beta f(T) + \beta' f(S))w_h) \tag{4.8}$$

The above analysis is based on a two-way join. For a replicated join involving k tables $(T_1, T_2, ..., T_k)$, we can perform a similar estimation as Equation 4.6 and 4.8. The only difference is the shuffling cost. Suppose the tables are joined on an attribute set $\mathcal{A}$, where $|\mathcal{A}| = n$, and we have m reducers. We use c_x to

denote the number of reducers for attribute a_x. Therefore, we have

$$m = \Pi_{x=1}^{n} c_x \tag{4.9}$$

As mentioned before, to improve the performance, the number of required reducers is set to be proportional to the size of corresponding table. If a_x is an attribute of table T_i, we use $f(a_x)$ to denote the size of T_i. Therefore, we compute c_x as

$$c_x = \frac{a_x \delta}{\Pi_{i=1}^{n} f(a_i)} \tag{4.10}$$

After combining Equation 4.9 and 4.10, we can estimate the value of δ and the number of required reducers for each attribute.

For table T_i, if it contains a join attribute set $\mathcal{A}'$ ($\mathcal{A}' \in \mathcal{A}$) , we need to replicate its data to r_i reducers, where

$$r_i = \Pi_{\forall a_x \notin \mathcal{A}' \wedge a_x \in \mathcal{A}} c_x \tag{4.11}$$

Therefore, the shuffling cost is computed as:

$$C'_{shuffle} = \mu \sum_{i=1}^{k} (\alpha_i r_i |T_i| (\beta_i f(T_i) + \delta_i)) \tag{4.12}$$

where α_i, β_i and δ_i denote the accumulative filter selectivity, projection selectivity and the size of keys of table T_i, respectively.

Compared to join, aggregation is much more similar to a map-only job. In the map phase, we scan the corresponding table and use the "group by" attributes as the key. In the reduce phase, aggregations are computed for each key. For table T, the map phase incurs a cost of

$$\begin{aligned} C_{map} &= |T| f(T) r_h + p(T,k)|T|\nu + \\ &\quad \alpha |T| \nu \log_2 x + y w_l + y(w_l + r_l) \end{aligned} \tag{4.13}$$

where x and y are defined as in Equation 4.5, and the cost of the reduce phase is estimated as:

$$C_{reduce} = \mu\alpha|T|(\beta f(T) + \delta) + \alpha|T|\nu + \gamma h w_h \tag{4.14}$$

where $\alpha|T|\nu$ denotes the CPU cost of aggregations, γ denotes the number of keys (groups) and h is the size (in bytes) of the result tuple.

4.5 Experimental Evaluation

Table 4.2: Cluster Settings

Parameter	Value
Size of Data Chunk	512M
Reducers per Node	1
Maximal Concurrent Mappers	2
Maximal Memory	4G
Replication Factor	3
Default Node Number	50
Data per Node	4G
θ	∞

We evaluate the effectiveness of *AQUA* on our in-house cluster, Awan, which contains 144 cluster nodes. The nodes are connected via three high-speed switches. Each node is equipped with Intel X3430 2.4 GHz processor, 8 GB of memory, 2x500GB SATA disks, gigabit ethernet, and operates CentOS 5.5. The cluster nodes are evenly divided into three racks. We evaluate the performance of AQUA on two data sets: TPC-H data and a real Twitter data set. The Twitter data set consist of 5 tables (*User*, *Tweet*, *UserGraph*, *TweetGraph* and *Location*) and the size of the data set is around 100G. For the experiments, 50 nodes of Awan are reserved and each node stores around 4G data. Since the maximal number of tables for our evaluated queries is 8 (TPCH Q8), which is quite small, we set θ to ∞ in order to generate query plans with higher quality. When θ is small, our query optimization algorithm tends to trade the quality of plans for the performance of query optimization. The detail configuration of the cluster is listed in Table 4.2.

We run some simple read and write jobs in the cluster to test I/O performance. Specifically, in our cost model, we set the cost ratio of Table 4.1 as follows: local read (r_l) = 1, local write (w_l) =1.2, HDFS read (r_h)=1.2, HDFS write (w_h)=2 and network I/O (μ)=1.2. CPU ratio (ν) is set to 0 in these experiments as most TPC-H queries are I/O intensive jobs .

Table 4.3: List of Selected TPCH Queries

Query ID	Joined Tables of the Query
Q3	$customer \bowtie orders \bowtie lineitem$
Q5	$customer \bowtie orders \bowtie lineitem \bowtie supplier \bowtie nation \bowtie region$
Q8	$part \bowtie supplier \bowtie nation \bowtie lineitem \bowtie orders \bowtie customer \bowtie nation \bowtie region$
Q9	$part \bowtie supplier \bowtie lineitem \bowtie partsupp \bowtie orders \bowtie nation$
Q10	$customer \bowtie orders \bowtie lineitem \bowtie nation$

For comparison purposes, we list the performances of three plans: $HIVE-MO$ (Hive-Manually Optimized) denotes the plans adopted by Hive's Benchmark [61], where all queries have been manually optimized for better performance; $AQUA$ represents the best plan generated by our query optimizer; and $HIVE-UO$ (Hive-Unoptimized) is the worst plan based on our cost model. We test all the TPC-H queries and list the results of query Q3, Q5, Q8, Q9 and Q10 (which are listed in table 4.3).

These queries provide the representative results. The rest of the queries either show a similar performance or are too simple to optimize, such as Q1 and Q6. Each query is run 10 times and we compute the average performance. In addition, we also evaluate the multi-way join query on the Twitter data set:

$$
\begin{aligned}
&SELECT\ count(*)\\
&FROM\quad Tweets\ T, Users\ U, Location\ L, UserGraph\ UG\\
&WHERE\ T.coord = L.coord\\
&AND\quad T.uid = UG.uid\\
&AND\quad UG.fid = U.uid\\
&AND\quad L.state = \text{“}state1\text{”}\\
&AND\quad U.state = \text{“}state2\text{”}
\end{aligned}
$$

The above query is named as QT, which computes the number of tweets published in "state1" by the users who follow a user in "state2". It can show how close the relationship between these two states are. We evaluate two instances

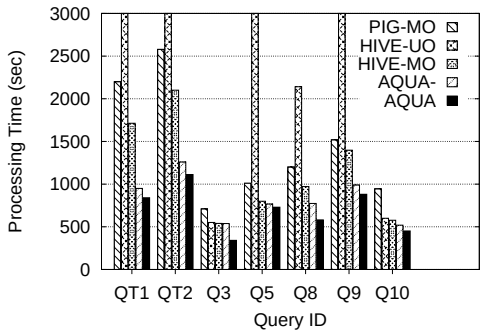


Figure 4.9: Query Performance

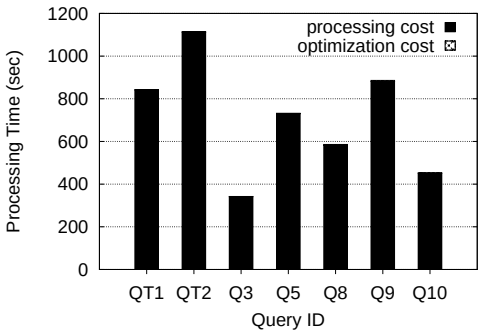


Figure 4.10: Optimization Cost

of this query, QT1 ("state1" = "WA" and "state2" = "NY") and QT2 ("state1" = "CA" and "state2" = "NY") in the experimental section.

4.5.1 Effect of Query Optimization

Figure 4.9 lists the overall performance of selected queries. In this figure, we also show the performance of Pig [83], which is denoted by $PIG-MO$. In our settings, Pig translates the join queries into MapReduce jobs using the same plans as $HIVE-MO$. We find that $PIG-MO$ performs worse than $HIVE-MO$ for all queries. Therefore, in the remaining experiments, we omit the results of $PIG-MO$. $AQUA-$ is a simplified version of AQUA. The final join tree selected by $AQUA-$ is implemented by a sequence of symmetric hash joins. We can see how replicated join improves the performance of the query by comparing $AQUA$ with $AQUA-$.

In all cases, $AQUA$ performs the best, which shows the effectiveness of our query optimization. For simple queries such as Q3 and Q10, AQUA generates two MapReduce jobs. One job performs the replicated join to process all the join operations, while the other job is used to do the "group by" and aggregations. For Q5, $HIVE-UO$ results in an "out of memory" exception for Hive, but before it triggers the exception, its running time is much longer than that of other schemes. In fact, $HIVE-UO$ generates some bad plans that cannot exploit all processing nodes (see section 4.2.2). For Q8, Q9 and QT1 and QT2, $AQUA$ performs significantly better than $HIVE-MO$, because both queries are complex (involving eight and six tables, respectively). In that case, it is difficult to manually optimize the query plan, while our query optimizer is able

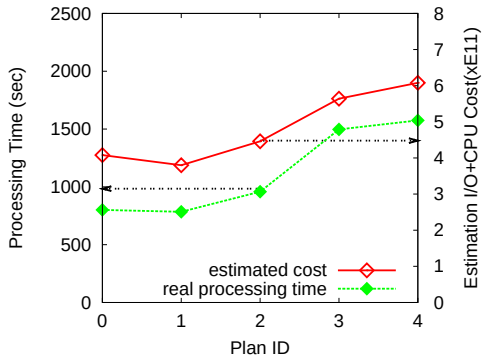


Figure 4.11: Accuracy of Optimizer

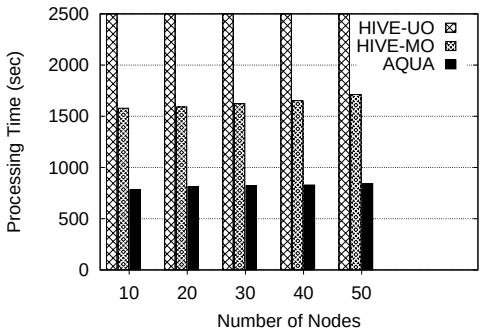


Figure 4.12: Twitter Query (QT1)

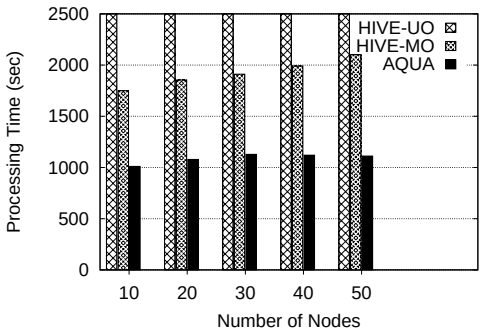


Figure 4.13: Twitter Query (QT2)

to deliver its superior performance. The execution time of QT2 is longer than QT1 because of two reasons: there are more users in "CA" than "WA" and the users in "CA" are more connected with the users in "NY" compared to the users in "WA". Figure 4.10 shows the cost of query optimization. For all the queries, our optimizer can complete its plan selection within seconds. Compared to the query processing cost, optimization cost is negligible. As a result, we can hardly recognize the white rectangle in the figure.

Figure 4.11 shows the accuracy of our query optimizer. We pick the first five query plans output by our optimizer for Q5 and show the plan's estimated cost and processing time. The optimizer employs a cost model to evaluate the costs of relational operators in the MapReduce framework, which considers both I/O cost and network cost. The estimated cost is used to predict the efficiency of a query plan and the optimizer selects the plan with minimal estimated cost to execute a query. In Figure 4.11, we observe that when a plan has a

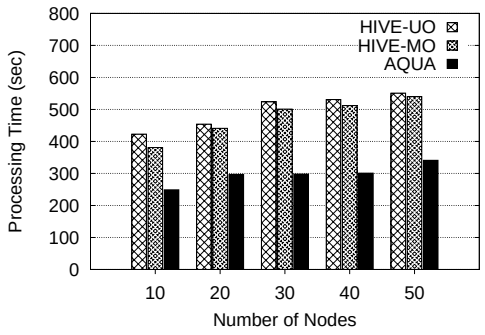


Figure 4.14: TPC-H Q3

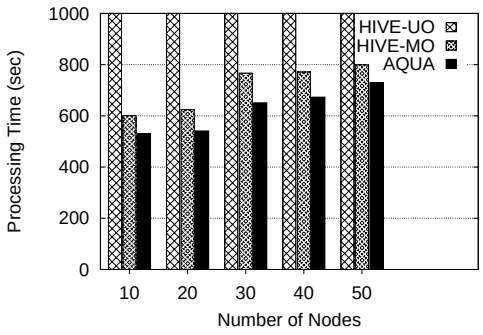


Figure 4.15: TPC-H Q5

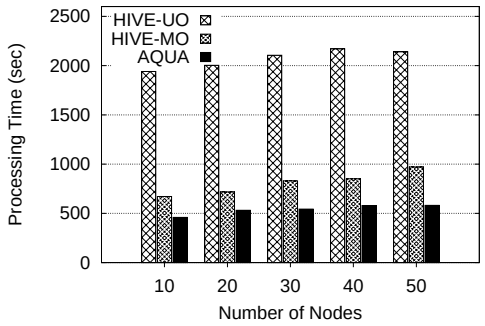


Figure 4.16: TPC-H Q8

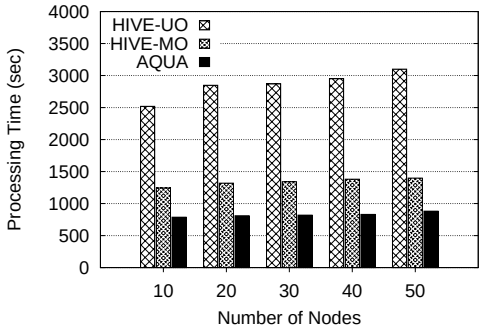


Figure 4.17: TPC-H Q9

higher estimated cost, it always requires more processing time, which verifies the accuracy of our optimizer.

4.5.2 Effect of Scalability

In this experiment, we evaluate the scalability of different schemes. In Figures 4.12, 4.13, 4.14, 4.15, 4.16, 4.17 and 4.18, the number of nodes varies from 10 to 50, and correspondingly, the total size of the data increases from 40G to 200G. The figures respectively show the performance of the twitter query (QT1 and QT2), Q3, Q5, Q8, Q9 and Q10.

In our experiments, AQUA and $HIVE-MO$ show linear scalability for all queries. But $HIVE-UO$ results in an "out of memory" exception for TPC-H Q5. This is caused by a plan that shuffles most intermedia results to a few *reducers*. When data size keeps increasing, the memory will become insufficient for some *reducers* eventually. Therefore, selecting good plans is

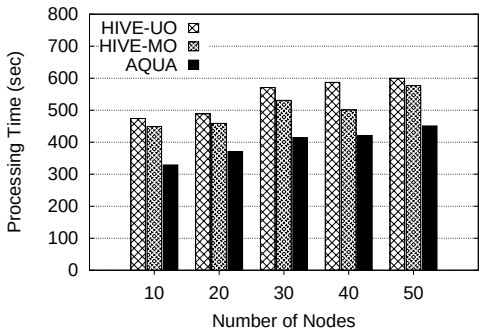


Figure 4.18: TPC-H Q10

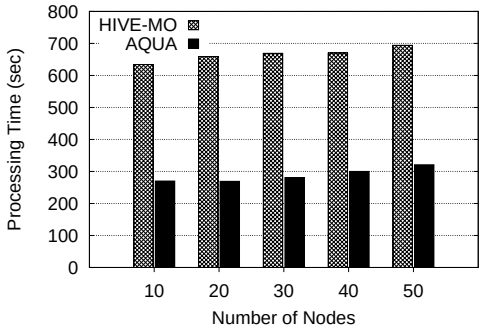


Figure 4.19: Performance of Shared Scan

extremely important for large-scale datasets.

$AQUA$ performs better than $HIVE - MO$ for different reasons. For Q3 and Q10, as mentioned before, $AQUA$ generates a single job to process all join operations. This strategy avoids repeatedly writing and reading data from HDFS. For Q5, $AQUA$ adopts a similar plan as $HIVE - MO$, except that it processes $N \bowtie R \bowtie S$ in a single job. However, as *nation* and *region* are two smallest tables in TPC-H, $AQUA$ achieves less improvement by applying the replicated join. The biggest performance gap is observed in Q8 and Q9. For these two queries, $AQUA$'s plans are quite different from those of $HIVE-MO$. $AQUA$ generates two replicated joins for each query and adopts the bushy plans to combine the results. Compared to $HIVE-MO$, the space of candidate plans in $AQUA$ is extended to include more possible plans. Therefore, $AQUA$ can perform better than $HIVE - MO$.

Figure 4.19 shows the effect of shared scan approach. We use Q17 in TPC-H as an example.

$$
\begin{aligned}
&select\ sum(l\_extendedprice)\ /\ 7.0\ as\ avg\_yearly \\
&from\ lineitem,\ part \\
&where\ p\_partkey\ =\ l\_partkey\ and\ p\_brand\ =\ '[BRAND]' \\
&\qquad and\ p\_container\ =\ '[CONTAINER]'\ and\ l\_quantity\ <\ (\\
&\qquad select\ 0.2\ *\ avg(l\_quantity)\ from\ lineitem \\
&\qquad where\ l\_partkey\ =\ p\_partkey);
\end{aligned}
$$

Q17 accesses *lineitem* in the outer query and the inner nested query. By

applying shared scan strategy, we only need to scan *lineitem* once, which can greatly reduce the I/O costs and hence improve the performance.

4.6 Summary

In this chapter, we have presented the design and implementation of *AQUA*, the core part of the offline analysis module in ART. Given an SQL-like query, *AQUA* generates a sequence of MapReduce jobs, which minimizes the cost query processing. *AQUA* adopts a two-phase optimization scheme. In the first phase, join operators are organized into various groups and one MapReduce job is generated for each group. In the second phase, a cost-based scheme is employed to search for an optimized plan that combines the results of different join groups. To reduce the search space, *AQUA* applies the features of the MapReduce framework to prune the search space. In particular, we consider both the left-deep and bushy plans. Also, we avoid generating a plan that under-utilizes computing resources. We evaluate our approach by running multi-way join queries on both TPC-H data and a real Twitter data set on our in-house cluster. The result verifies the effectiveness of our proposed query optimizer.

This work is published as a full paper in *the ACM Symposium on Cloud Computing* (SOCC) 2011 [108].

CHAPTER 5

R-Store: A Scalable Distributed System for Supporting Real-Time Analytics

It is widely recognized that OLTP and OLAP queries have different data access patterns, processing needs and requirements. Hence, the OLTP queries and OLAP queries are typically handled by two different systems, and the data are periodically extracted from the OLTP system, transformed and loaded into the OLAP system for data analysis. In microblogging companies, with the awareness of the ability of big data in providing enterprises useful insights from vast amounts of data, effective and timely decisions derived from real-time analytics are important. It is therefore desirable to provide real-time OLAP querying support, where OLAP queries read the latest data while OLTP queries create the new versions.

In Chapter 4, we have introduced the offline analysis module of ART, which assumes that the data are unchanged after they are extracted-transformed-loaded (ETL) from the OLTP module. In this chapter, we discuss R-Store, the distributed storage module of ART. In addition to processing the OLTP queries, R-Store is specifically designed and implemented to enable real-time analytics. R-Store maintains both the real-time twitter data and the data cube on these data. When real-time data are updated, they are streamed to a streaming MapReduce, namely Hstreaming, for updating the cube on incremental basis. Based on the metadata stored in the storage system, either the data cube

or OLTP database or both are used by the MapReduce jobs for aggregation queries. We propose techniques to efficiently scan the real-time data in the storage system, and design an adaptive algorithm to process the real-time query based on our proposed cost model. The main objectives are to ensure the freshness of answers and low processing latency. The experiments conducted on the TPC-H data and the Twitter data demonstrate the effectiveness and efficiency of our approach.

5.1 Introduction

Database systems implemented for large scale data processing are typically classified into two categories: OLTP systems and OLAP systems. The data stored in OLTP systems are periodically exported to OLAP systems for processing. In recent years, MapReduce [44] framework has been widely used as a large scale OLAP system because of its scalability. However, most of these only focus on how the OLAP queries are efficiently processed. The issue of freshness of the OLAP results has not been addressed.

In this chapter, we try to address the problem of large scale real-time query processing using MapReduce framework. Specifically, we focus on a subset of the OLAP query: the real-time aggregation (RTA) query. The RTA is defined as follows: a real-time aggregation (RTA) query accesses, for each key, the latest value preceding the submission time of the query [52]. Specifically, we propose and design a scalable distributed system called R-Store, in which the storage system supports multi-versioning, and each version is associated with a timestamp. Each aggregation query operates on the version of data that exists at the time it is submitted whereas each OLTP transaction creates a new version. R-store uses the MapReduce framework where the mappers of the aggregation query directly access the real-time data stored in the storage system. The storage system is implemented by extending HBase [3]. HBase supports the `HBaseScan` operation that takes a timestamp as input and returns the version of the data with the largest timestamp before the scan operation. Though this can be used to offer consistent data to RTA queries, simply using this default scan operation for querying the data stored in HBase is inefficient due to the following reasons:

1. HBase only stores a fixed number of versions for each key, and automati-

cally removes the versions that exceed this cap by its default compaction policy. To support real-time querying, this number has to be set to infinity in case the old versions are removed during the running of an RTA query. However, this will lead to continuously increasing of the data size, and waste too much space to store the unused data.

2. For each RTA query, the entire HBase table has to be scanned and shuffled to the mappers, which is a very costly process.

To facilitate efficient processing of RTA queries, we periodically materialize the real-time data into a data cube and implement an `IncrementalScan` operation in HBase to avoid the shuffling of the entire HBase table to MapReduce during real-time querying. To the best of our knowledge, this is the first work that proposes a scalable RTA distributed system based on MapReduce framework. In summary, the contributions of this paper are as follows:

1. We propose a scalable distributed system framework called R-Store, for performing RTA query processing. R-Store evaluates an RTA query by transforming it into a MapReduce job, which is run on our modified HBase (in remaining of this paper, we name it as HBase-R in order to differentiate it from HBase), to obtain the real-time data.

2. We propose an efficient storage model for caching the data cube result. The data cube is treated as historical data, while the data updated after the refresh time of the data cube are real-time data. We also propose a more efficient scan operation in the storage model for obtaining the real-time data.

3. We integrate streaming MapReduce into our system, which maintains a real-time data cube in the reducers, and periodically materializes the data cube. This data cube update method is much faster than the data cube re-computation method, and in turn accelerates the processing of RTA query since fewer real-time data are scanned during the query execution.

4. We design an algorithm to efficiently process the RTA queries, which takes both the historical data cube and the real-time table as input. We also propose a cost model that guides the adaptive processing of RTA.

5. We perform an extensive experimental study on a cluster with more than one hundred nodes, which confirms the effectiveness of the cost model, and the efficiency and scalability of R-Store.

The remaining of the chapter is organized as follows. We first present the architecture, design and implementations of R-Store in Section 5.2 and 5.3. In Section 5.4, we discuss the processing of real-time aggregation queries. We evaluate the performance of R-Store in Section 5.5 and summarize this chapter in Section 5.6.

5.2 R-Store Architecture and Design

In this section, we present the architecture of R-Store, the design philosophy of the storage system, and how the data cube is maintained.

5.2.1 R-Store Architecture

Figure 5.1 illustrates the architecture of R-Store. The system consists of four components: a distributed key/value store, a streaming system for maintaining the real-time data cube, a MapReduce system for processing large scale OLAP queries, and a MetaStore for storing some global variables and configurations.

The OLTP queries are submitted directly to the key/value store, while the aggregation queries are processed by the MapReduce system. The simplest method of supporting RTA for MapReduce is to scan the whole real-time table and obtain the latest version before the submission time of the aggregation query for every key/value pair (`FullScan` operation), as the input of the MapReduce job. The key/value store has to support multi-version concurrency control in case the OLTP queries and aggregation queries are blocked by each other. However, this method is not efficient because obtaining one version for each key/value pair is a costly operation in large scale distributed systems. Note that in real applications, such as social networks, the updates usually follow a Zipf distribution, and within a time interval, only a small portion of keys are updated in the table. Based on this observation, we try to accelerate aggregation queries by materializing the real-time table into a data cube. When an aggregation query is submitted to the system, it first connects to *MetaStore* to acquire the timestamp of the query for consistency. The

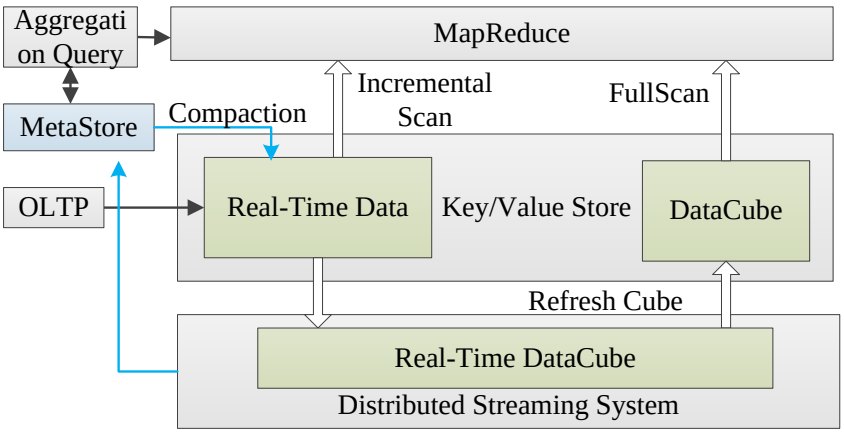


Figure 5.1: Architecture of R-Store

statistics stored in *MetaStore* are also used to optimize the query based on our proposed cost model (Section 5.4.3). After the optimization by the cost model, the aggregation query can be transformed to a MapReduce job that takes as input both the historical data in the data cube and the real-time data in the key/value store. To efficiently access real-time data, the key/value store is designed to support incremental scan (Section 5.2.2). The real-time data is scanned by the `IncrementalScan` operation, while the data cube is scanned by the `FullScan` operation. The `IncrementalScan` operation only shuffles the key/value pairs that are updated after the last building of the data cube, and thus is much faster than `FullScan` because fewer data are shuffled.

The data cube is also stored in the distributed key/value store and is periodically refreshed based on the real-time table. The versions of the key/value pairs before the refresh time of the data cube are compacted in order to accelerate the scan time of the real-time table. The performance of refreshing the data cube is crucial to our system because if the data cube is refreshed fast, more data are compacted by our compaction scheme, and fewer real-time data are accessed during the scan operation. In an extreme case where no update is submitted since the data cube refresh, the MapReduce job only needs to scan the data cube. To efficiently refresh the data cube, the updates applied to the key/value store are streamed to the streaming system, and a real-time data cube is maintained in the local storage of the streaming system. The real-time data cube is periodically materialized to the key/value store to refresh the data cube. Based on our experimental results, this method is much faster than the

method of re-computing the data cube, and the throughput of this method is sufficiently high to process the update streams from the key/value store.

Once this refresh process is completed, the timestamp of the latest data cube is sent to *MetaStore*, and the compaction process is invoked to compact the real-time data. The *MetaStore* also stores other global information, including the submission time of each aggregation query, the frequency of materializing the data cube, etc.

5.2.2 Storage Design

The key/value store must support multi-version concurrency control techniques to ensure that the aggregation query and the OLTP query do not block each other. In addition, our storage design considers many other features including efficient file scan operations, compaction scheme and load balancing, which we discuss below.

Full and Incremental Scans

To handle aggregation queries and to build the data cube, a scan operation needs to be implemented in the key/value store. Two types of scan operations are required, which are used in different scenarios:

- **FullScan(T_i)**. For each key/value of the table, the `FullScan` operation takes a timestamp T_i as input, and returns the latest version of the value before T_i. The data returned by this operation can be used to create or re-compute the data cube.

- **IncrementalScan(T_1, T_2)**. This operation takes two timestamps T_1 and T_2 ($T_1 < T_2$) as input, and returns two versions for the keys updated after T_1. The first version is the latest value before T_2, and the second version is the latest value before T_1. If a new key is inserted (not updated) into the store after T_1, only one version is returned. This operation can be used in the RTA query processing algorithm. During the query processing, T_1 is set to the querying time, while T_2 is set to the data cube refresh time. By combining the data returned by `IncrementalScan` and the data cube, we can efficiently re-construct the most real-time data, and using these data to process the RTA query.

Global and Local Compactions

Since each key may have several versions, the scan operations read more than one version of the data to obtain the required versions, incurring unnecessary I/O cost. To reduce the number of stored versions for each key and to improve the scan performance, the data are automatically compacted. We provide two forms of compaction:

- **Global Compaction**. The global compaction process is launched immediately following each data cube refresh. For the same key, all the versions inserted before the data cube refresh are merged into one version. We call this version V_{DC}, which is consistent with the data cube and is used in updating the data cube.
- **Local Compaction**. The local compaction process is invoked on each node. At first, the submission time of the current running scan process (T_{scan}) on the *region* is acquired by the compaction process. For each key, the latest version of the data before T_{scan} is accessed by this scan process. Thus, the local compaction only compacts the older versions that will not be accessed by any scan process. Furthermore, V_{DC} is not changed during this compaction so that the new data cube can be computed correctly when the data cube is refreshed.

Load Balancing

In most applications, some key ranges might be updated more frequently than others, causing skewed load on the nodes. In addition, since the update operation inserts a new version of the key into the node instead of replacing the old version, there is a skew on the amount of data on the nodes. This affects the performance of the scan process on those nodes. The solution is to split heavily updated ranges in the key/value store and move some data to other nodes.

5.2.3 Data Cube Maintenance

To improve the performance of RTA queries, a data cube is maintained in the key/value store. A data cube could be either a full data cube, an Iceberg cube, or a closed cube. The selection of the best suitable data cube depends on the applications, which is not the focus of this work. To make it general, we only

consider the full data cube, which consists of a lattice of cuboids. There are two approaches to refresh the data cube:

Re-Computation. To re-compute the data cube, the `FullScan` operation is used. It takes a timestamp T_i as input, and returns the latest version of the value before T_i for all the keys stored in this *region*. Each mapper of the MapReduce job takes the results of the `FullScan` operation on one *region* as input. For each cuboid, a key/value pair is generated. The map output key is the combination of the dimension attributes for the cuboid, while the map output value is the numeric value. The reducers compute the aggregation value for each cell of each cuboid, and output the result to the key/value store.

Incremental Update. The second approach is performed in two steps: propagation step and update step. The propagation step computes $\triangle DC$ (change of the data cube) from $\triangle T$ (change of the table), and the update step updates data cube based on $\triangle DC$. However, not all data cubes can be incrementally updated. The incremental update only works for self-maintainable aggregate functions [78] (the new cell value can be computed from the old cell value and the updated tuples) such as SUM, COUNT, and the algebraic functions derived from them.

In R-Store, the re-computation approach is used to build the first data cube, while the incremental update approach is adopted to maintain a real-time data cube in the stream processing module. The streaming system updates its data cube with the update streams coming from the key/value store, and periodically materializes the data cube into the storage system. As the updating of the data cube consists of two phases, which can be processed by the MapReduce processing logic in nature, a streaming version of MapReduce is used as the stream processing module of R-Store.

5.3 R-Store Implementations

In this section, we present the implementations of R-Store. Specifically, we show how we implement our storage system, namely HBase-R, on top of HBase to fulfill the design philosophy discussed in Section 5.2.2.

5.3.1 Implementations of HBase-R

HBase [3] is an open source distributed key/value store. A table stored in HBase is partitioned to several *regions*, which are assigned to a certain nodes, and each node runs a *region* server to manage *regions* and serve the transactions. Inside a region, the data of the same column family (a group of columns) are stored in the same structure, which is called *store*. A *store* has an in-memory structure, *memstore*, and several in-disk files, *storefiles*. When a new version of data is about to be inserted into this *store*, it is first inserted into the *memstore* and appended to the write ahead logs. Once the size of the *memstore* reaches its upper bound, the data in the *memstore* are transferred to a *storefile*. The store files are sorted in inverse chronological order. Inside the *memstore* or *storefile*, the data are sorted by keys, and the versions for each key are sorted in inverse chronological order. HBase only supports the `FullScan` operation, so we designed and implemented `IncrementalScan` in HBase-R.

IncrementalScan

For a *store* in a *region*, by accessing the same key across the *storefiles* and *memstore* in parallel, the `IncrementalScan` operation scans the keys in ascending order. For each key, the version with the larger timestamp is scanned earlier. For all the versions of a key, the algorithm checks the timestamp of each version and returns the required two versions. If the key has only one version, which means the operation on the key is an insertion, the `IncrementalScan` only returns that version for the key.

For real-time queries and data cube update, scanning the key/value pairs in HBase-R is the most costly step. It is, therefore, important to improve the performance of `IncrementalScan`. For this purpose, we propose an adaptive incremental scan algorithm.

First, we maintain an in-memory structure to estimate $d(T)$, the number of distinct keys updated since the last refresh of the data cube. Estimating $d(T)$ in a data stream has been well studied [65]. A straightforward method is to keep all the keys in memory, and, for each key, to maintain a bit value to indicate whether or not it has been updated. However, this method requires a considerable amount of memory to store the keys. In HBase-R, the size of a *region* is configured before the data are inserted. Thus, the num-

ber of keys for a *region* has an upper bound (M), which can be estimated by $SizeOfRegion/SizeOfKeyValue$. Since each *region* usually stores a range of consecutive keys, a hash function $h(key)$ can be used to map a key to a value between 0 and $M-1$, and a bit array of size M, *DistinctKeys*, is maintained in memory to indicate whether or not a key has been updated. Using this bit array, to compute the number of updated values on a node with even one billion distinct keys, only 128 MB of memory are required.

To improve the performance of `IncrementalScan`, the above data structure is used in the adaptive incremental scan algorithm (Algorithm 5.1). When an `IncrementalScan` request is sent to a *region* server, the first parameter (T_1) is always set to the refresh time of the current data cube (T_{DC}), and the second parameter (T_2) equals to the submission time of the query (T_Q). Instead of scanning all the key/value pairs before T_Q, the key/value pairs in *memstore* are scanned first. Note that in *memstore*, there might be several versions for a key, and only the newest version is cached in *kvMap* (line 1). The number of key/values updated after T_{DC} but not in *memstore* is then computed (line 7), and the random read cost of these key/values is estimated. If this cost is smaller than the cost of scanning all the data between T_{DC} and T_Q, the *storefile* index is used to directly read the values for these keys (lines 8 to 14). In this way, the latest versions for the updated keys are obtained. Then, by simply scanning the key/values before T_{DC}, the latest versions before T_{DC} for the updated keys are returned to the client. Since the cost of scanning *memstore* (in-memory structure) is much lower than the cost of scanning *storefile*), when $d(T)$ is large, the adaptive incremental scan is almost the same as the default *IncrementalScan*. In contrast, when $d(T)$ is small, this adaptive scan strategy incurs fewer I/O operations.

Compaction

HBase's default compaction process combines all the *storefiles* into one file and retains only one version for each key. If R-Store simply inherits HBase's default compaction process, the version of the data which is consistent with the latest data cube will be lost, and the most real-time data cube cannot be re-constructed to process the RTA query or data cube slice query. Thus, in HBase-R, we implemented two different compaction schemes. The global compaction in HBase-R is similar to HBase's default, but with a different triggering

Algorithm 5.1: Adaptive IncrementalScan

```
input: Timestamp T_DC, Timestamp T_Q, boolean[] DistinctKeys, int
       NumDistinctKeys
1  kvMap ← new HashMap<Key, Value>();
2  for KeyValue kv ∈ MemStore do
3      if kvMap.contain(kv.key) then
4          continue;
5      else
6          kvMap.put(kv.key, kv.value);
7  NumKeysNotInMemory ← NumDistinctKeys - kvMap.size();
8  if CostOfRandom × NumKeysNotInMemory <
   CostOfScan × NumOfUpdatedKeyValues then
9      for key updated but not in kvMap do
10         kv ← randomRead(key);
11         kvMap.put(kv.key, kv.value);
12     for each kv before T_DC do
13         if kvMap.exist(kv.key) then
14             send kvMap(kv.key) and kv;
15 else
16     delete kvMap;
17     invoke the default IncrementalScan(T_DC, T_Q)
```

condition. In addition, it always keeps one latest version before the data cube refresh time for each key. The local compaction only compacts the data that are earlier than a certain timestamp. To ensure that the compaction process does not block the scan processes, the compacted data are stored in different files, instead of directly replacing the un-compacted data. The files that contain the old versions are replaced by the compacted files when they are not accessed by any scan process. Since the compaction process competes with aggregation queries for CPU and I/O resources, there is a tradeoff between the frequency of the compaction and the performance of the whole system. We define a threshold so that the local compaction process is triggered when $(numberOfTuples)/(numberOfDistinctKeys)$ exceeds this threshold.

Load Balancing

HBase has its default *region* size, which is 256MB. If the size of the data for a *region* is larger than this size, it is automatically split to two sub-*regions*, which are distributed to other nodes. In HBase's default setting, only a fixed number of versions for a key are stored. Once the number of versions for all the keys in this *region* reaches the maximum number, the size of the *region* would not change regardless of the frequency of key updates in this *region*. This requires users to manually split the hot *region*. In contrast, in R-Store, we do not strictly remove the old versions of the updated keys once the number of versions exceeds HBase's default setting. We wait until the size of frequently updated *region* reaches its upper bound, and the split happens automatically.

5.3.2 Real-Time Data Cube Maintenance

R-Store adopts HStreaming for maintaining the real-time data cube (note that other streaming MapReduce systems can also be used in R-Store). Each mapper of HStreaming is responsible for processing the updates within a range of keys. The map function of the data cube update algorithm is shown in Algorithm 5.2. When an update for a key arrives, the old value for this key is retrieved from the local storage if exists. To efficiently retrieve the old value, a clustered index is built for the key/values, and the frequently updated keys are cached in memory. In reality, the updates are usually on a small range of keys, and the old value of the updates have a high probability to be directly retrieved from the cache. If the key is new (thus, does not exist in local storage), for each cuboid, one key/value pair is generated and shuffled to the reducers. The map output key is the combination of the dimension attributes, and the map output value is the numeric value. If the key of the update exists in local storage and the updated key/value pair falls into the same cell for a cuboid, one key/value pair is shuffled to the reducer, and the numerical value is equal to the value change. Otherwise, two key/value pairs are generated, one is the new value with a tag "+", and the other is the old value with a tag "-".

The reduce function is invoked at a time interval w_r specified by the user. For example, if the time interval is set to one second, the reducers will cache the incoming intermediate data within the past second, and apply the reduce function to them. Another time interval, w_{cube}, defines how frequently the data

Algorithm 5.2: Map Function for Incremental Update

```
   input: KeyValue kv
 1 oldkv = retrieveFromLocal(kv.key);
 2 if oldkv == null then
 3     for cuboid in data cube do
 4         CuboidK ← extractCuboidKey(cuboid, kv.value);
 5         CuboidV ← extractCuboidValue(kv.value);
 6         CuboidV.setTag("+");
 7         Emit(CuboidK, CuboidV);
 8     insertToLocal(kv);
 9 else
10     oldCuboidV ← extractCuboidValue(oldkv.value);
11     oldCuboidV.setTag("-");
12     newCuboidV ← extractCuboidValue(kv.value);
13     newValue.setTag("+");
14     for cuboid in data cube do
15         oldCuboidK ← extractCuboidKey(cuboid, oldkv.value);
16         newCuboidK ← extractCuboidKey(cuboid, kv.value);
17         if oldCuboidK == newCuobidK then
18             newCuboidV.set(computeChangeOfCell
               (oldCuboidV,newCuboidV));
19             Emit(newCuboidK, newCuboidV);
20         else
21             Emit(oldCuboidK, oldCuboidV);
22             Emit(newCuboidK, newCuboidV);
23     updateToLocal(kv);
```

cube is materialized. The reduce function to incrementally update the data cube is shown in Algorithm 5.3. A reducer merges the local data cube (DC) with the intermediate key/value pairs that it receives from mappers (which is a cell in a cuboid) if these are due to an update before next cube refresh time (T_{DC}). Otherwise, it stores these key/value pairs in $\triangle DC'$. When the timestamps of the incoming updates on all mappers are larger or equal to T_{DC}, the data cube refresh process is invoked, which writes the local data cube to HBase-R (different cuboids are written to separate HBase-R tables). The incoming cells during this refresh process are still written to $\triangle DC'$ since their timestamps are no less than T_{DC}. When this refresh process is completed, T_{DC} is incrementally changed, and DC is merged with $\triangle DC'$. In streaming system,

Algorithm 5.3: Reduce Function for Incremental Update

```
   input: Key key, List<Value> vlist, Context context
1  i ← 0, sum ← 0;
2  for Value v in vlist do
3      if v.timestamp < T_DC then
4          MergeWith(key, v, DC)
5      else
6          MergeWith(key, v, △DC′)
```

to deal with fault tolerance, the accumulated states of the stream computation have to be checkpointed periodically. The data streams after the checkpointing time are stored in logs and will be used during the recovering process. In R-Store, the data cube materialized to key/value store is indeed a checkpointing of the real-time data cube. Since the key/value pairs after the last data cube refresh are still stored in the storage (even though some intermediate versions of the key/value pairs might be removed by the local compaction process, the necessary versions for building the next data cube are still there), the real-time data cube maintenance process can be recovered using the data cube and the real-time table without extra efforts of checkpointing.

5.3.3 Data Flow of R-Store

Figure 5.2 illustrates the data flow between HBase-R, HStreaming and MapReduce in R-Store. Each HBase-R *region* server handles several *regions*. Some of these *regions* belong to the real-time table, while the others belong to the data cube. An OLTP query is submitted to one of the *region* servers, and stored in *memstore* of the *region* it belongs to. If the size of the *memstore* reaches its upper bound, the data are written into HDFS as a *storefile*. Once the update is written to HBase-R, it is streamed to a mapper in HStreaming based on the key of this update. In the mappers of HStreaming, the change of a cell for each cuboid is computed and shuffled to reducers. On the reduce side, the real-time data cube is updated and cached in local disk. At time interval, HStreaming materializes its local data cube into HBase-R and notifies *MetaStore* with the timestamp of the latest data cube. The compaction process is then launched to compact the versions of data before data cube is refreshed.

When an aggregation query arrives, it acquires a timestamp from the *Meta-*

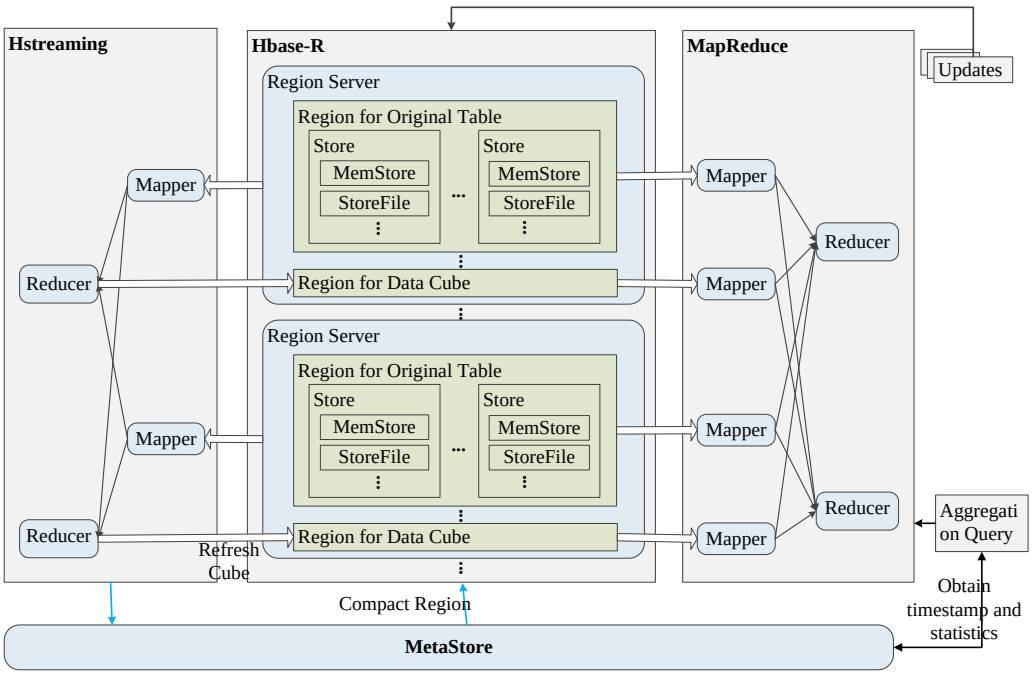


Figure 5.2: Data Flow of R-Store

Store, together with the statistics of the real-time table stored in HBase-R. It is then transformed to a MapReduce job based on the data statistics, and submitted to the system. Each mapper starts a scan operation over its input *region* belonging to either the real-time table or the data cube. At the end of the job, the results of aggregation query are stored in HBase-R.

5.4 Real-Time Aggregation Query Processing

Section 5.2 to 5.3 described in detail the architecture and implementation of R-Store. In this section, we discuss how the RTA queries are processed. In R-Store, if the input of the MapReduce job is only the data cube, the performance of the scan phase on the map side is maximized, but the result might be stale. To maximize the freshness of the OLAP query, all the updated key/value pairs before the submission time of the query must be considered. Thus, not only the data cube, but also the real-time table must be scanned.

Suppose the creation time of the data cube is T_{DC} and the submission time of the query is T_Q. For each updated key after T_{DC}, `IncrementalScan` running on the real-time table returns both the old version before T_{DC} and the latest

Algorithm 5.4: Map Function for IncreQuerying Algorithm

```
   input: KeyValueList kvlist, Context context
 1 key ← null, value ← null;
 2 if kvlist.size == 1 then
 3 |   key ← extractKey(kvlist[0].key);
 4 |   if key is not filtered then
 5 |   |   value ← kvlist[0].value;
 6 |   |   value.setTag("Q");
 7 |   |   Emit(key, value);
 8 else
 9 |   key ← extractKey(kvlist[0].value);
10 |   if key is not filtered then
11 |   |   value ← extractValue(kvlist[0].value);
12 |   |   value.setTag("+");
13 |   |   context.write(key, value);
14 |   |   value ← extractValue(kvlist[1].value);
15 |   |   value.setTag("−");
16 |   |   Emit(key, value);
```

version before T_Q, if its two parameters are set to T_{DC} and T_Q respectively. By merging these two versions with the numeric values of each cuboid, the latest cubiod value can be computed on demand, and the freshness of the RTA query can be satisfied. In the following subsection, we present the query processing algorithm (called *IncreQuerying*) making use of the `IncrementalScan` operation.

5.4.1 Querying Incrementally-Maintained Cube

We implement *MultiTableInputFormat* so that each MapReduce job can scan the data of multiple tables, and the scan operation of each table can be configured as either full scan or incremental scan. Using this input format, the MapReduce job for *IncreQuerying* can access two types of input tables: one is the cuboid table for which a full scan is performed, and the other is the real-time table over which the incremental scan is used.

Map. Algorithm 5.4 describes the map function. The mappers filter the cell and the real-time tuple based on the filtering condition. The cells and tuples that will be aggregated are assigned the same partition key and shuffled to the

same reducer. The output value for the cell is the selected numeric value, while the output value for the real-time tuple is the original value, which will be used to re-compute the numeric value. The value is attached with a tag "Q", "-" or "+" to indicate whether it is the cell value of a cuboid, the old value of a key/value pair, or the new value, respectively. This phase is similar to the map phase of incrementally updating the data cube, except that a filtering process is added, and the partition key could be different from the dimension attributes of the data cube.

Reduce. The reduce function calculates the new value of each cell based on the old cell value, the change of the cell and the aggregation function. The cell key of the reduce function is different from that of Algorithm 5.3. For example, for the TPC-H *part* table, to compute a rectangular subset of the cube ($mfgr$ = "Manufacturer#13"), the key of the reduce function is the combination of the attributes (*brand* to *container*) after removing $mfgr$.

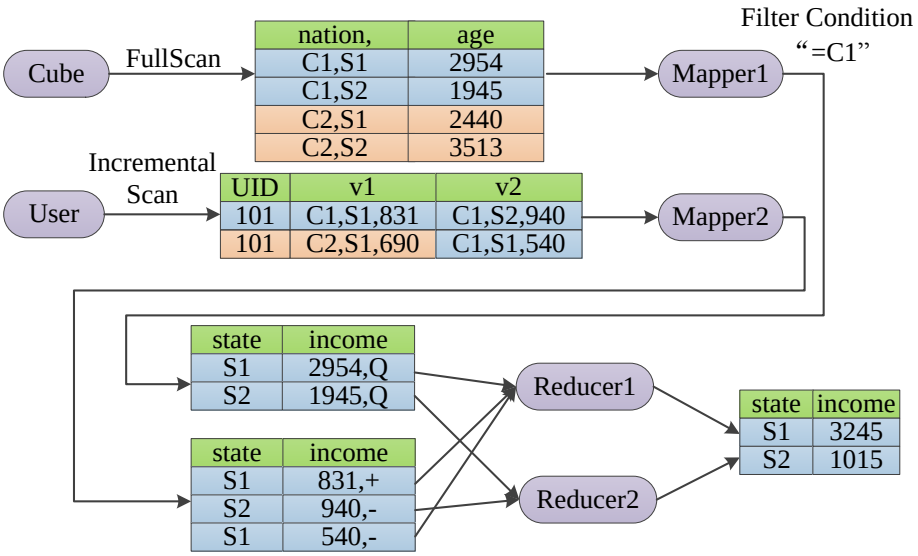


Figure 5.3: Data Flow of IncreQuerying

Figure 5.3 shows the data flow of *IncreQuerying* alogrithm for an RTA query on a two-dimensional cuboid (*mfgr*,*brand*). The query computes the summation of *price* for each brand produced by "M1". To ensure the freshness of the results, all the data of the queried table and the cuboid are scanned to process the real-time query. Note that the row key of the stored data cuboid is the combination of the dimension attributes. Therefore, if the filtering condition

Table 5.1: Data Cube Operations

Operator	**Parameters**
`addFilter`	attribute name, function, value
`addGroupBy`	group-by attribute
`setAggregationFunc`	aggregation function name
`setNumericAttribute`	numeric attribute name

Algorithm 5.5: Example Data Cube Query

```
input: DataCube cub
1 cub.addFilter("mfgr", "=", "Manufacturer#1");
2 cub.addFilter("brand", "=", "Brand#13");
3 cub.addGroupBy("type");
4 cub.setNumericAttribute("retailprice");
5 cub.setAggregateFunc("sum");
6 cub.setOutputTable("resultTable");
7 SubmitQuery(cub);
```

contains some attributes that could form a prefix of the row key, such as "Manufacturer#1" and "Brand#13", the range scan function of HBase-R can be used to avoid scanning the entire data cube. The min key for the range scan is "Manufacturer#1,Brand#13", and the max key is "Manufacturer#1,Brand#14".

To relieve users from having to merge the real-time data and the historic data cube, we define new data cube operators and automatically translate these operators into a MapReduce job. The processing of the real-time data is transparently encapsulated into the operators shown in Table 5.1. Algorithm 5.5 shows an example that computes the summation of the *retailprice* for all the parts with "Brand#13" produced by "Manufacturer#1", grouped by *type*.

5.4.2 Correctness of Query Results

When an aggregation query is submitted to the system, a timestamp T_Q is acquired for this query from the *MetaStore*. To guarantee correctness, if the query needs to scan a table several times, the scan process on each node always returns the data before time T_Q. However, in a distributed system, although clocks can be synchronized to a certain extent, there might still be some difference between the clocks of different nodes. If the current timestamp T_k on a certain node k is smaller than T_Q, the next scan process on this node would

return some data between T_k and T_Q, which leads to an inconsistent state. To avoid this inconsistency, if the timestamp T_Q is larger than T_k, the scan process is blocked for a while until T_Q is equal to or smaller than T_k. Since clock synchronization can achieve one millisecond accuracy in local area networks under ideal conditions, the delay of the scan process can be ignored compared to the processing time.

5.4.3 Cost Model

The *IncreQuerying* algorithm discussed above is not always better. Since the `IncrementalScan` scans not only the real-time table, but also the data cube, it can incur a higher cost. In addition, it shuffles two versions for each updated key to MapReduce. When there are fewer OLTP transactions or the OLTP transactions access a small range of keys, *IncreQuerying* algorithm is better because `IncrementalScan` only transfers a small amount of data to the mappers. An alternative implementation of real-time querying is similar to re-computing the data cube: a `FullScan` operation is used to return one version for each key/value pair regardless of whether or not it has been updated. When the updates are uniformly distributed across all the keys, this baseline implementation could be more efficient. To be able to select a more efficient approach, we propose a cost model. Table 5.2 shows the parameters of the cost model. The most important one is $s(T)$, which is the percentage of the keys that are updated after refreshing the data cube: $s(T) = d(T)/|T|$.

Cost Analysis of IncreQuerying Algorithm

First, we estimate the cost of the scan phase on the map side. The scan phase consists of two parts: scanning the local data on each HBase-R node (`FullScan` or adaptive `IncrementalScan` discussed in Section 5.2.2) and shuffling these data to mappers. The `FullScan` scans all the *storefiles* of the real-time table, while the adaptive `IncrementalScan` scans fewer *storefiles* when $d(T)$ is small and the *memstore* has enough number of keys. However, whether the adaptive `IncrementalScan` is activated depends on the status of each HBase-R node and cannot be easily estimated. Thus, we assume that the cost of reading the local data on each HBase-R node are the same for `FullScan` and `IncrementalScan`. The difference is in the number of

Table 5.2: Parameters

Parameter	**Definition**
$\|T\|$	number of tuples in table T
$d\|T\|$	number of distinct keys updated
$f(T)$	size of the tuple in table T
$s(T)$	percentage of the keys that are updated since the last data cube refresh
$\|C\|$	number of cells in the selected cuboid
$d(C)$	size of dimension attributes of cuboid
$n(C)$	size of numeric attribute of cuboid
$\|Q\|$	number of tuples in the query result
$s(Q)$	filtering selectivity of the query
$d(Q)$	size of query result key
$n(Q)$	size of query result value
sh_{HBase}	cost ratio of shuffling from HBase-R
w_{HBase}	cost ratio of HBase-R writes
sh_{MR}	cost ratio of shuffling in MapReduce
c_L	cost ratio of local I/Os
m_T	number of mappers for table T
m_C	number of mappers for the cuboid
B	block size

tuples transferred from HBase-R to mappers. Thus, we base our analysis on the network transfer cost. At first, the mappers scan both the real-time data and the data cube. The cost of shuffling the real-time data and data cube to mappers is:

$$C_{scan-R} = sh_{HBase} \times 2|T| \times f(T) \times s(T)$$

while the cost of scanning the data cube is:

$$C_{scan-C} = sh_{HBase} \times |C| \times (d(C) + n(C))$$

After the scan phase, the real-time data and the data cube are sorted. The size of the map output for these two types of data is:

$$S_{MO-R} = 2 \times (s(Q) \times |T| \times s(T)/m_T) \times (d(Q) + n(Q))$$
$$S_{MO-C} = (|C|/m_C) \times s(Q) \times (d(Q) + n(Q))$$

and the cost of external sorting the map output is:

$$\begin{aligned} C_{sort-map-R} =& m_T \times 2c_L \times \\ & ((S_{MO-R} \times log_B(S_{MO-R}/(B+1))) \\ C_{sort-map-C} =& m_C \times 2c_L \times \\ & ((S_{MO-C} \times log_B(S_{MO-C}/(B+1))) \end{aligned}$$

The data on all the mappers are shuffled to the reducers after the mapper completes. The cost of shuffling is:

$$\begin{aligned} C_{shuffling} =& sh_{MR} \times s(Q) \times \\ & ((2 \times |T| \times s(T) + |C|) \times (d(Q) + n(Q)) \end{aligned}$$

In the reduce phase, the cost of sort merging process is:

$$\begin{aligned} C_{reduce-merge} =& 2c_L \times s(Q) \times \\ & (2 \times |T| \times s(T) + |C|) \times (d(Q) + n(Q)) \end{aligned}$$

and the cost of writing the data into HDFS is:

$$C_{reduce-write} = w_{HBase} \times |Q| \times (d(Q) + n(Q))$$

The cost of baseline algorithm can be analyzed in a similar way. Based on the cost model discussed above, the more efficient approach is dynamically selected when a real-time query is submitted.

5.5 Evaluation

In this section, we evaluate the R-Store on our in-house cluster of 144 nodes. The cluster settings are shown in Table 5.3. Each node is equipped with Intel X3430 2.4 GHz processor, 8 GB of memory, 2x500 GB SATA disks, each of which is connected by a gigabit ethernet and running CentOS 5.5. The cluster nodes are evenly placed onto three racks. We adopt the Twitter data for the experiments. However, we only have one copy of the twitter data, while we need online transactions that update the existing keys. Therefore, we write our own

Table 5.3: Cluster Settings

Parameter	Value
CPU	Intel X3430 2.4GHz
Memory	8GB
Disk	2x500 GB SATA disks
Default Node Number	100
Data per HBase Node	4.8G(original) + 2.4G(update)
Number of keys per HBaseNode	40 million

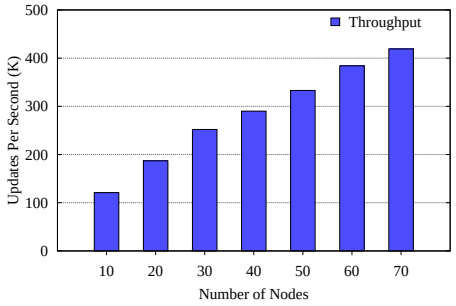


Figure 5.4: Throughput of Real-Time Data Cube Maintenance

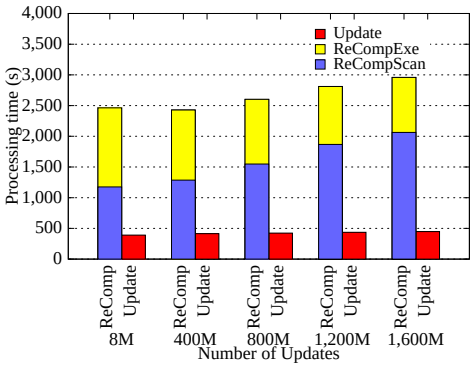


Figure 5.5: Performance of Data Cube Refresh

scripts to simulate the updating of the *User* table in Figure 1.2. The scripts can update the information of a UID based on either a uniform distribution or Zipf distribution. In addition, for the scalability experiments, to ensure that the size of data on each node are roughly the same, we need to adjust the total number UIDs as the number of nodes increases. However, the *User* table contains only 80 millions of users, which is not enough for the scalability experiments. Thus, our data generation script takes a *scale* parameter as input and generates $200,000 \times scale$ distinct users based on the original 80 millions of users.

5.5.1 Performance of Maintaining Data Cube

In this experiment, we first measure the throughput of our real-time data cube maintenance algorithm to ensure that it has sufficiently high processing capacity to handle the update streams from HBase-R. As can be seen in Figure 5.4, when HStreaming is configured with 10 nodes, the algorithm can process more than 100K updates per second, which is even higher than the throughput of HBase-R

with 40 nodes (the throughput of HBase-R will be discussed in Section 5.5.3).

We compare the two methods for refreshing the data cube: re-computation and incremental update. We deploy the system on 100 nodes, with 40 nodes for MapReduce, 40 nodes for HBase-R, and 20 nodes for HStreaming. The scale factor of the Twitter data is set to 8000, so that there are 1,600,000,000 UIDs for *User* table. On each HBase-R node, there are 4.8GB data. The data cube is built after the *User* table is loaded into HBase-R.

Figure 5.5 shows the processing time of the two methods. The distribution of updated keys follows a Zipf distribution. We adjust the factor of the Zipf distribution so that about 1% keys are updated, while the number of updates is increased from 8 million to 1,600 million. Since HBase-R does not remove the previous version of the data, 0.024 GB to 4.8 GB of new data are inserted into each HBase-R node. The processing time of re-computation has two parts: the blue rectangle (ReCompScan) is the scan time of the real-time table, and the yellow rectangle (ReCompExe) is the execution time of the MapReduce job after the scan phase. As the number of updates increases, the data stored on each HBase-R node increases as well. Thus, more data are scanned at the HBase-R side for the re-computation approach, and the running time of the scan phase for re-computation is increased over time. However, as illustrated in Figure 5.5, the running time of the ReCompExe decreases as the number of updates increases, which is counterintuitive. We expected that the execution time of the MapReduce job should remain the same in different settings as they process the same number of key/value pairs. The reason for the decrease in ReCompExe is that ReCompScan and ReCompExe are pipelined. The more time ReCompScan takes, the more these two phases overlap, reducing the time ReCompScan takes.

In contrast, the processing time of incremental update consists of only one part (the red rectangle): the time it takes to write data cube into HBase-R. This is because our real-time data cube maintenance algorithm is fast enough to update the real-time data cube with the data streams from HBase-R. Thus the latency of periodically refreshing the data cube in HBase-R equals to the time of writing the real-time data cube into HBase-R. This time is related to the the size of the data cube and does not change as the number of updates increases.

We also evaluate the scalability of R-Store. In this experiment, the number

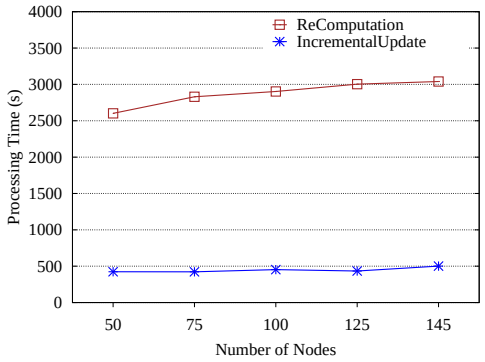


Figure 5.6: Scalability

of nodes and the data size increase with the same ratio. The percentage of updates is set to 1% for different scalability settings. As can be seen in Figure 5.6, the running time of both re-computation (the brown line) and incremental update (blue line) do not change much as the number of nodes increase, which demonstrates the scalability of R-Store.

5.5.2 Performance of Real-Time Querying

In this experiment, we investigate the performance of real-time querying. First, we compare the *IncreQuerying* algorithm, which optimizes the real-time query using the data cube, with the *Baseline* algorithm implemented with the `FullScan` operation. The cluster settings are the same as those of Figure 5.5, except that we fix the number of updates to 8,000 million and vary the percentage of the keys updated.

Figure 5.7 shows the processing time of both algorithms for a typical data cube slice query:

$$SELECT\ avg(income)\ FROM\ users$$
$$WHERE\ country\ =\ \text{“}USA\text{”}$$
$$GROUPBY\ state, gender, age$$

The processing time of the *Baseline* algorithm consists of two parts: the black rectangle (ReCompScan) is the time to scan the real-time table, and the yellow rectangle (ReCompExe) is the execution time of the MapReduce job after the scan phase. In contrast, the processing time of *IncreQuerying* consists

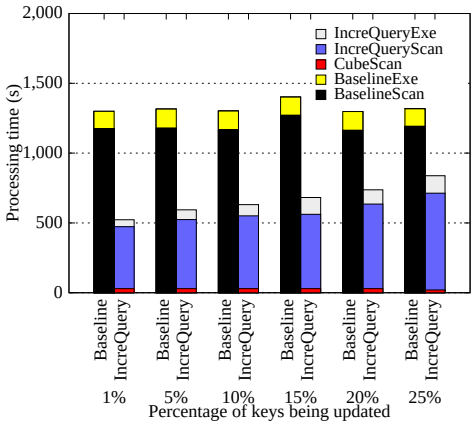


Figure 5.7: Data Cube Slice Query on Twitter Data

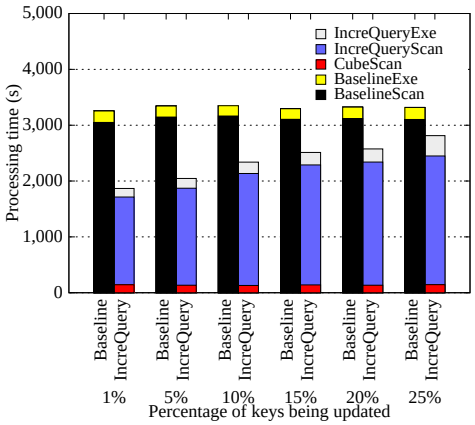


Figure 5.8: Data Cube Slice Query on TPCH data

of three parts: the red rectangle (CubeScan) is the time to scan the data cube, the blue rectangle (UpdateScan) is the time to scan the *part* table in HBase-R, and the grey rectangle (UpdateExe) is the execution time of the MapReduce job after the scan phase.

When only a small range of keys are updated, *IncreQuerying* performs much better than *Baseline*. It outperforms the *Baseline* approach for two reasons: (1) by using adaptive incremental scan, it scans fewer data in HBase-R and shuffles fewer data to MapReduce; (2) its MapReduce job processes fewer data than that of re-computation. However, as the percentage of updated keys increases, more data are shuffled from HBase-R to MapReduce. Thus, both the scan time and the execution time increase. In contrast, for *Baseline*, since the `FullScan` always shuffles one version for each key to MapReduce, the amount of data shuffled from HBase-R is constant. As a result, the running time of *Baseline* is almost constant. Due to the existence of the filtering condition on attribute *mft*, most tuples of the table are filtered, and fewer data are sorted and shuffled during the execution of the MapReduce job. As a result, the difference between the execution times is not so significant. In general, *IncreQuerying* algorithm outperforms *Baseline* algorithm when the percentage of keys being updated is low.

In addition to the above query, we also evaluate the *IncreQuerying* algorithm on TPCH data, a standard benchmark for data warehousing. Figure 5.8 shows

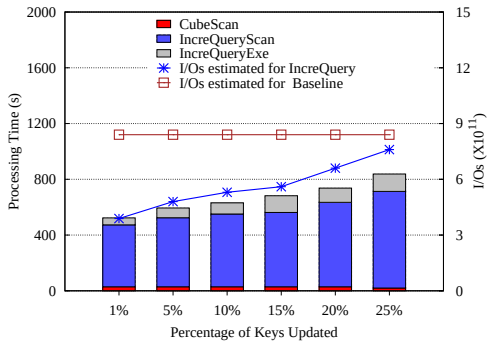


Figure 5.9: Accuracy of Cost Model

the result of a data cube slice query with the same experimental settings.

$$SELECT\ sum(prices)\ FROM\ part$$
$$WHERE\ mft\ =\ \text{“}Manufacture\#1\text{”}$$
$$GROUPBY\ brand, type, size, container$$

Since there are more dimension attributes in the table *lineitem* for the TPCH-Q1, more intermediate keys will be generated during the execution of the query. Thus, the TPCH-Q1 query runs slower than the data cube slice query on twitter data.

To select the better querying method among the two, we use the cost model (Section 5.4.3) to estimate the number of I/Os. Figure 5.9 shows the running time of *IncreQuerying*, and the I/Os estimated for both *Baseline* and *IncreQuerying* algorithms. The y-axis on the left is the processing time of the query, while the y-axis on the right is the estimated I/Os. The estimated number of I/Os for *IncreQuerying* (the blue line) increases linearly with almost the same slope (the histogram) as the processing time of the query, while the estimated number of I/Os for the Baseline (the brown line) is constant, which is around 2.52×10^{11}. This result hence verifies the accuracy of our cost model.

Compared to querying only the data cube, RTA queries require two additional steps, which incur additional cost: scanning the real-time data from HBase-R, and merging the real-time data with the data cube on demand in MapReduce. On each HBase-R node, the key/values are stored in *storefile* format. Though only one or two versions of the same key are returned to MapReduce, HBase-R has to scan all the *storefiles* of the *part* table.

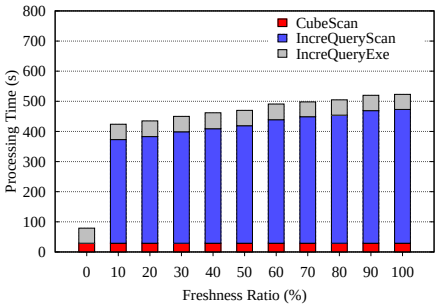


Figure 5.10: Performance vs. Freshness

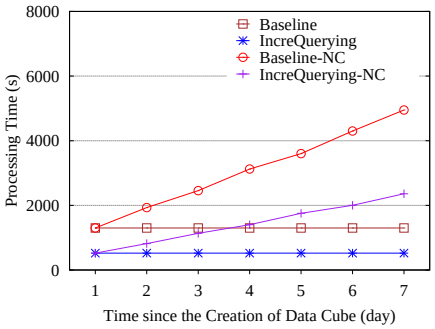


Figure 5.11: Effectiveness of Compaction

Since the *memstore* is materialized to HDFS when it is full, these files are sorted by time. Thus, instead of scanning all the *storefiles* and *memstore* between T_{DC} and T_Q, only the *storefiles* between T_{DC} and a user specified timestamp T_i ($T_i < T_Q$) are scanned. The value of T_i decides the freshness of the result. There is a tradeoff between the performance of the query and the freshness of the result: the smaller T_i is, the fewer real-time data are scanned. Figure 5.10 shows the query processing time with different freshness ratios, which is defined as the percentage of the real-time data we have to scan for the query. In this experiment, $|User| = 1600$ million, and 800 million updates on 1% distinct keys are submitted to HBase-R. When the freshness ratio is 0, the input of the query is only the data cube. Thus, the cost of scanning the real-time data is 0. When the freshness ratio increases to 10%, the cost of scanning the real-time data is around 1500 seconds because the cost of scanning the real-time table dominates the aggregation query. As the freshness ratio increases, the running time of *IncreQuerying* method increases slightly, which is due to two reasons: (1) the data before T_{DC} still need to be scanned; and (2) the amount of data shuffled to mappers are roughly the same with different ratios.

Figure 5.11 depicts the effectiveness of our compaction scheme. In this experiment, we measure the processing time of the data cube slice query when the compaction scheme is applied (*Baseline* and *IncreQuerying*) and when it is not (*Baseline-NC* and *IncreQuerying-NC*). We submit 800 million updates to the server each day, and the percentage of keys updated is fixed to 1%. The data cube is refreshed at the beginning of each day, and the aggregation query is submitted to the server at the end of the day. Since the data are compacted after the data cube refresh, the amount of data stored in the real-time table

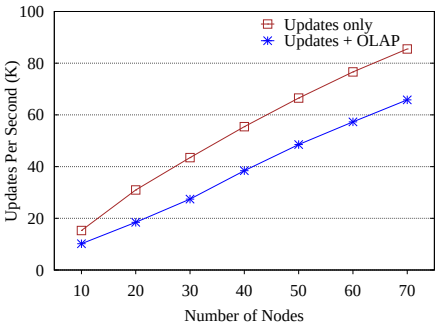


Figure 5.12: Throughput

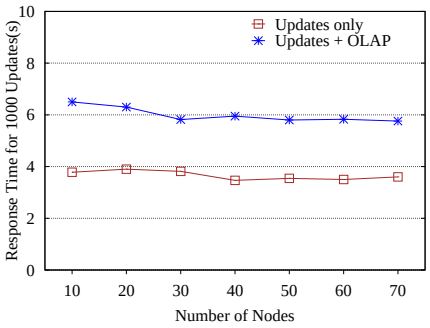


Figure 5.13: Latency

are almost the same at the same time of each day. The processing time of *Baseline* and *IncreQuerying* are thus almost constant. In contrast, when the compaction scheme is turned off, HBase-R stores much more data, and the cost of locally scanning these data becomes larger than the cost of shuffling the data to MapReduce. As a result, the processing time of *Baseline-NC* and *IncreQuerying-NC* increases over time.

5.5.3 Performance of OLTP

In this experiment, we investigate the performance of OLTP queries when aggregation queries are running. The workload is update-only, and the keys being updated are uniformly distributed. We launch ten clients to concurrently submit the updates when the system is deployed on 100 nodes. Each client starts ten threads, each of which submits one million updates (100 updates in batch). Another client is launched to submit the data cube slice query. That is, one aggregation query and approximately 50,000 updates are concurrently processed in R-Store. The system reaches its maximum usage in this setting based on our observation. When the system is deployed on other number of nodes, the number of clients submitting updates is adjusted accordingly.

Figure 5.12 shows the throughput of the system. The throughput increases as the number of nodes increases, which demonstrates the scalability of the system. However, when aggregation queries are running, the update performance is lower than running only OLTP queries. This result is expected, because the aggregation queries compete for resources with the OLTP queries. We also evaluate the latency of updates when the system is approximately fully used. As shown in Figure 5.13, the aggregated response time for 1000 updates are

similar with respect to varying scales.

5.6 Summary

MapReduce is a parallel execution framework, which has been widely adopted due to its scalability and suitability in a large scale distributed environment. However, most existing works only focus on optimizing the aggregation queries and assume that the data scanned by MapReduce are unchanged during the execution of a MapReduce job. In microblogging systems, the real-time results from the most recently updated data are more meaningful for decision making. In this chapter, we propose R-Store for supporting real-time aggregating on MapReduce. R-Store leverages stable technology (HBase and HStreaming) and extends them to achieve high performance and scalability. The storage system of R-Store adopts multi-version concurrency control to support real-time aggregating. To reduce the storage requirement, it periodically materializes the real-time data into a data cube and compacts the historical versions into one version. During query processing, the proposed adaptive incremental scan operation shuffles the real-time data to MapReduce efficiently. The data cube and the newly updated data are combined in MapReduce to return the real-time results. In addition, based on our proposed cost model, the more efficient query processing method is selected. To evaluate the performance of R-Store, we have conducted extensive experimental study using the TPC-H data and the tweet data. The experimental results show that our system can support real-time aggregation queries much more efficiently than the baseline methods. Though the performance of OLTP degrades slightly due to the competition for resources with the aggregation queries, the response time and throughput remain good and acceptable.

This work is published as a full paper in *the IEEE International Conference on Data Engineering* (ICDE) 2014 [73].

CHAPTER 6

TI: An Efficient Indexing System for Real-Time Search on Tweets

In traditional search engines, the inverted index is typically reconstructed on a periodical basis so that search queries can be answered efficiently. The freshness of the search results thus relies on the frequency of index construction. However, Such an indexing method naturally does not support real-time search. To make a blog or tweet searchable as soon as it is published, the index must be updated in real time.

In this chapter, we propose *TI* (Tweet Index), a distributed adaptive indexing system for supporting real-time search. The basic idea of *TI*'s index scheme is to only index tweets that may appear in the search result in real-time. The other tweets are indexed in batch. This strategy significantly reduces the indexing cost and yet still provides the search results with high quality. The processing of the real-time indexing requests is distributed to multiple TI slaves in order to handle the increasing data volume in microbloging systems. We also design a new ranking scheme that considers relationships between the users and tweets. The experimental study using a real Twitter dataset confirms the efficiency of *TI*.

6.1 Introduction

The increasing popularity of social networking systems changes the form of information sharing. Instead of issuing a query to a search engine, the users log into their social networking accounts and retrieve news, URLs and comments shared by their friends. This is in part caused by the failure of conventional search engines in providing real-time search service for social networking systems. For example, it is difficult to search a new blog or tweet uploaded a few minutes ago using a conventional search engine. The problem is further amplified in the microblogging systems such as Twitter due to unprecedented amount of tweets or microblogs being posted each day. For example, Tumblr [15] estimated that there were more than 2 million posts and fifteen thousands new users every day [6]; and based on a latest report from Twitter [8], it handled more than 50 million tweets per day.

Providing real-time search service is very challenging in large-scale microblogging systems, in which thousands of new tweets are published per second. To search the newly uploaded tweets, the data need to be indexed in real time, and the response time of the search query need not be affected much. The objectives are therefore contradictory since maintenance of up-to-date index will cause severe contention for locks on the index pages. Another problem of real-time search is the lack of effective ranking functions. Since the current Twitter search engine sorts the results based on time, and therefore, the latest tweets have the higher rankings. Without proper ranking functions, the search results are meaningless. However, defining a ranking function for real-time search is not trivial, and the function must have the following two desiderata:

1. The ranking function must consider both the timestamp of the data and the similarity between the data and the query. As an example, for a given query submitted to Twitter, we do not want to get tweets posted many weeks ago, even though they may contain the keywords of the query. On the other hand, newer tweets with less information are not preferred either. Hence, the ranking function is composed of two independent factors, time and similarity.

2. The ranking function should be cost-efficient. As we want to support real-time search using a ranking function partially based on time, we have to

compute the rankings during query time. Thus, the computation of the ranking function should not incur high overhead.

In this chapter, we propose *TI* (Tweet Index), a novel indexing system for supporting real-time search in microblogging systems such as Twitter. *TI* is designed based on the observation that most tweets will not appear in the search results. Therefore, we can significantly reduce the indexing cost by delaying indexing less useful tweets. In essence, *TI* classifies the tweets into two types, *distinguished* tweets and *noisy* tweets. *TI* has of two indexing schemes: a real-time indexing scheme for distinguished tweets and a background batch indexing scheme for noisy tweets. Given a new tweet, *TI* analyzes its contents and determines its type. If it is a distinguished tweet, we will index it immediately. Otherwise, it is grouped with other noisy tweets and periodically, the batch indexing scheme is invoked to index all the noisy tweets in one go. The design principle of *TI* is similar in spirit to the partial indexing scheme [96, 93], and it is also related to the view selection problem [20].

In *TI*, the ranking function plays the major role in deciding whether the tweets are distinguished tweets or noisy tweets and in retrieving meaningful answers. We therefore propose a new ranking function by combining the user graph and tweet graph. In social networks, each user can be considered as a node and different nodes are connected together via the friend links. The user graph denotes the relationship among the users. Naturally, a popular user will have more friends and his/her blogs/tweets also attract wider readership. Therefore, the PageRank value for the user graph is calculated to compute the ranking for each user. Besides the user graph, the tweets also form a graph, as some tweets are exchanges between people while some tweets are reply to the other tweets. We group tweets into topics based on their relationships, and we measure the popularity of topic based on its statistics. Finally, our proposed ranking function is composed of the user's PageRank, the popularity of topics, the TF (Term Frequency) and the timestamp. The IDF (Inverse Document Frequency) is not used in *TI*, since the length of a microblog is fairly small and often capped at certain length (e.g. in Twitter, it is capped at 140 characters).

We evaluate *TI* by using a real Twitter dataset collected for a user group within the last three years. The experiments examine the performance of our indexing scheme and the effect on the quality of query results. We also compare our ranking function with the other relevant ranking functions such as

Palanteer [75] and Twitter's default ranking method.

In summary, the contributions of this paper are as follows:

1. We propose TI, a distributed indexing system for supporting real-time search.

2. TI adopts an adaptive indexing scheme to reduce indexing cost. Only the tweets that have high ranking scores are indexed immediately into the real-time index. Others are indexed in the batch mode.

3. The ranking scheme of TI considers the user-relationships, the popular topics, the similarity between tweets and queries and the timestamp.

4. The experimental results on real twitter data show the efficiency and effectiveness of TI.

The rest of the chapter is organized as follows. In Section 6.2, we present the earlier works in social network search and the corresponding database techniques. In Section 6.3, we introduce the overview architecture of *TI*. The details of *TI*'s indexing scheme and ranking function are discussed in Section 6.4 and Section 6.5, respectively. We evaluate the performance of the proposed schemes in Section 6.6. And the chapter is concluded in Section 6.7.

6.2 System Overview

6.2.1 Social Graphs

In order to design an efficient search mechanism for microblogging systems, we first examine the characteristics of social networks.

In social networks, users are connected together by friend links (in Twitter, it's following/follower link). Typically, a popular and famous user will have more friends than an ordinary or low-profile user. Here, we define a user graph $G_u = (U, E)$, where U is set of users in the system and E is the friend links between them.

Apart from the user graph, we have another graph that is induced by the relationship of microblogs or tweets. Figure 6.1 shows a tree structure of tweets, where each node denotes a tweet and the directed edge indicates that one tweet replies to or retweets another tweet. For example, tweet B replies to tweet A

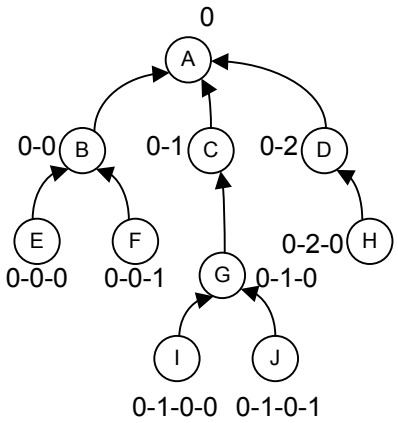


Figure 6.1: Tree Structure of Tweets

and thus A is the parent node of B in the tree. The tweet that does not reply to others becomes the root of the tree. In this paper, we use a tweet tree to represent a discussion topic. When searching, tweets in the same topic can be grouped together and returned. We do not explicitly maintain the tweet tree, as it may incur too much overhead. Instead, we assign each tweet a tree encoding ID, which is similar to the Dewey Order ID [100] in XML search. Given tweet t_i, we sort its child nodes by their timestamps (the time that the tweet is inserted into the system). Suppose the encoding of t_i is "x" and tweet t_j is t_i's kth child, t_j's encoding is "x"+"-"+"j", where + indicates the string concatenation. With the help of tree encoding, we can easily reconstruct the tree structure.

6.2.2 Design of the TI

Figure 6.2 shows the architecture of TI, our distributed indexing system for tweets. TI adopts a master-slave architecture. The TI master node is responsible for partitioning the user graph based on the number of slave nodes. Each slave node has two processes: the index processor is responsible for indexing the incoming tweets for the users in a sub-graph, while the query processor is responsible for processing the search queries.

When a new tweet is published by a user, it is first stored in our distributed storage system, R-Store. The tweet is then shuffled to an index processor on an arbitrary slave node in TI for indexing. If the publisher of the tweet does not belong to the sub-graph on that slave, the tweet is shuffled to the specific

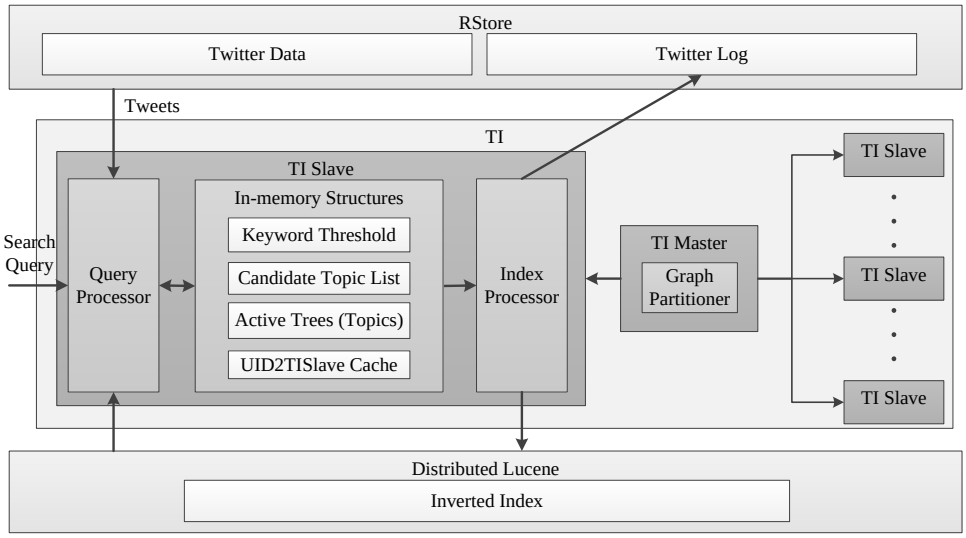


Figure 6.2: Architecture of *TI*

slave by looking up the mappings between the user id and the slave id. We cache the mapping for the active users who frequently publish new tweets in memory, and thus the slave id of a specific user can be obtained quickly. After the tweet is sent to the corresponding TI slave, the index processor inside that TI slave determines whether the tweet should be indexed or not. In general, the following data are maintained in order to support the real-time indexing in TI.

1. Distributed Inverted Index. We maintain a distributed inverted index for the tweet data, which is partitioned by the keywords of tweet data. Given a keyword, the inverted index returns a tweet list, T. T consists of a set of tweet IDs, and tweets in T are sorted by their timestamps (the time when a tweet is inserted into the system). Figure 6.3 shows the index structure of the inverted index. For each record in the index, we keep its tweet ID, TID (inherited from the status ID provided by Twitter), to identify different tweets. Then, for the ranking purpose, we keep the U-PageRank of a tweet (to be defined in Section 6.5), the TF (Term Frequency) value, the tree ID and the timestamp of the tweet. Tree ID is the TID of the root node in a tweet tree. Records of the same keyword are maintained as a list and the latest record is inserted into the head of the list. As a result, the records are sorted by their timestamps in the list.

2. Tweet MetaData. To facilitate our ranking scheme, we also keep the meta-

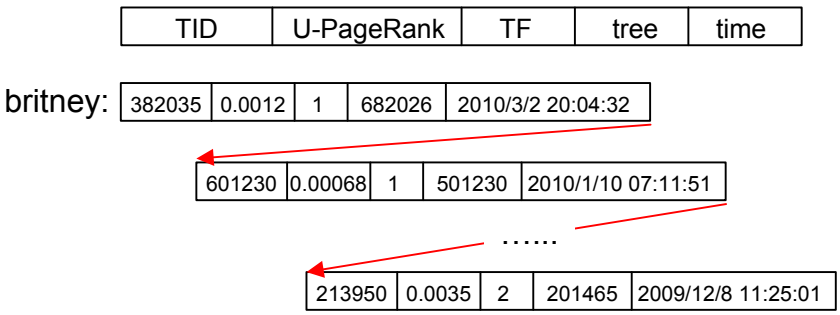


Figure 6.3: Structure of Inverted Index

Table 6.1: Example of Tweet Table

TID	RID	tree	time	count	coding	UID	pointer
26476	76732	25742	...	0	0-0-0	...	null
57380	76732	25742	...	0	0-0-1	...	null
26980	null	26980	...	1	0	...	1022
47806	null	47806	...	0	0	...	1034

data of a tweet. Specifically, we define a tweet table as shown in Table 6.1. Based on a tweet's content, we know whether the tweet replies/re-tweets another tweet. We maintain the ID of the replied tweet as RID, and it can be used to retrieve the parent tweet. If a tweet belongs to an existing tree, we keep the root ID of the tree, which can be obtained from its parent tweet. Otherwise, we create a single node tree by using the tweet itself as the root. We also keep the timestamp of each tweet and the *count* attribute denotes the number of tweets that reply to this tweet. To enable efficient reconstruction of the tree, the encoding of the tree node is stored with each tweet. The author ID UID of a tweet is defined as the foreign key in the tweet table. Finally, if a tweet is not indexed and written back to the log file, we keep a pointer to its offset in the log file.

Besides the tweet table, *TI* keeps a log file for recording the unindexed tweets. *TI* selectively indexes the inserted tweets, the distinguished tweets. The noisy tweets are appended to the log file and periodically, a background batch indexing process will scan the log file to index the noisy tweets.

3. In-Memory Structures. To facilitate the fast index maintenance and search query processing, we keep some useful information in the memory,

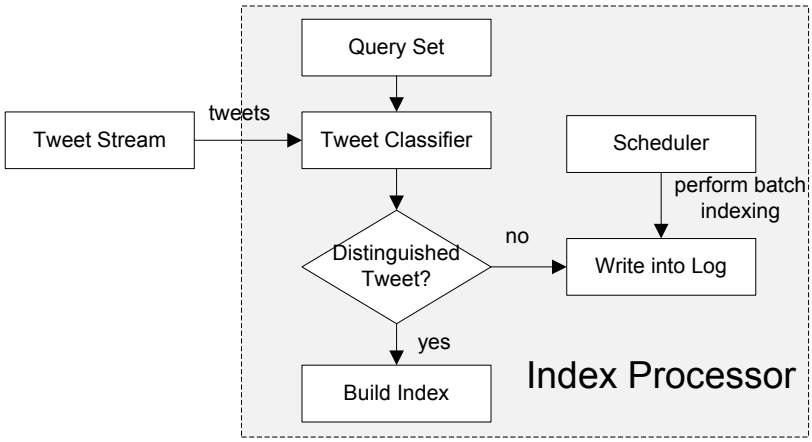


Figure 6.4: Data Flow of Index Processor

such as keyword threshold, candidate topic list and active trees (topics). Keyword threshold records the statistics of recent popular queries. The candidate topic list maintains the information about recent topics, while the active trees (topics) represents the hotly discussed topics. We assume that the users only reply or retweet to the people he follows, and thus in each TI slave, we only store the topics discussed by the users of the sub-graph handled by that slave.

Based on above information, we can quickly classify a tweet as a distinguished or noisy tweet and adopt different indexing scheme accordingly.

A search request is submitted to an arbitrary query processor in TI. The query processor first retrieves the TID lists for the keywords from the distributed inverted index. It then obtains the necessary information (e.g., the active trees that contains the TIDs) from other slaves, and re-ranks the TIDs based on these infomation. At last, it retrieves the tweets for the top-k TIDs and returns the results to the user.

6.3 Content-based Indexing Scheme

The basic idea of the *TI*'s indexing scheme is indexing the tweets based on their contents and their rankings with respect to past queries. Intuitively, it streams a new tweet into an existing set of popular queries, and based on its ranking, determines if it should be indexed in real-time or in batch periodically. Figure 6.4 shows the data flow in *TI*'s index processor. In this section, we present

how we classify the tweets and apply the adaptive tweet indexing strategy. The details of ranking function $\mathcal{F}$ will be discussed in the next section.

6.3.1 Tweet Classification

The first challenge in the design of TI's indexing strategy is on the measurement of the importance of a tweet. Limited by its size, a tweet itself does not provide too much information. Therefore, we apply a query-based classification approach. We assume that users are only interested in the top-K results. This assumption can easily be verified by the statistics of search engines [57] where 62% of the users click a result in the first page and more than 90% of the users do not browse beyond the third page of the results.

Formally, the problem can be stated as follows.

Definition *Tweet Classification*
Given a tweet t and a user's query set $\mathcal{Q}$, t is said to be a distinguished tweet, if $\exists q_i \in \mathcal{Q}$ and t is a top-K result for q_i based on the ranking function $\mathcal{F}$. Otherwise, t is a noisy tweet.

To answer top-K queries in query set $\mathcal{Q}$, we just need to index the distinguished tweets, while the noisy tweets can be indexed periodically. In this way, we avoid high real-time update costs.

Obviously, for a different query set $\mathcal{Q}$, the classification result will be different. Ideally, when all possible queries are considered, the classification will provide an accurate result for every query. However, the maintenance cost may neutralize the benefit of partial indexing. Fortunately, it has been confirmed that, like any social phenomenon, the search engine queries[22] and social networking queries [99] do in fact follow the well known Zipf's distribution. In other words, the top 20% queries represent 80% of the user requests. Therefore, only popular queries are maintained in $\mathcal{Q}$ to reduce maintenance cost. In particular, suppose the nth query appears with a probability of

$$p(n) = \frac{\beta}{n^{\alpha}} \tag{6.1}$$

where α and β are parameters that describe the Zipf's distribution. α determines how skew the distribution is. The larger α is, the more skew the distribution is. In many cases, α is set to a value close to 1. β is a constant

value, which is used to normalize the zipf's distribution in order to ensure that summation of $p(n)$ equals to 1. Let s be the number of submitted queries per second. The expected time interval of the nth query is

$$t(n) = \frac{1}{p(n)s} \tag{6.2}$$

That is, after $t(n)$ seconds, the nth query will be submitted to the system with high probability. Suppose we perform our batch indexing every t' seconds. We will keep the nth query in $\mathcal{Q}$, only if $t(n) < t'$. The intuition of this strategy is that for infrequent queries, we do not need to update the index frequently.

To estimate the query distribution, we keep a query log in disks. When a new unseen query arrives at the system, we assume it is an infrequent query and do not insert it into $\mathcal{Q}$. $\mathcal{Q}$ is updated at the next batch indexing process. We search the query log to build a query histogram and extrapolate the distribution using Zipf's law. Based on Equation 6.2, popular queries are inserted into $\mathcal{Q}$.

After having defined the classification problem, a naive method can be designed directly from the definition. Suppose the tweet set is $\mathcal{T}$. Given a query $q_i \in \mathcal{Q}$, we use $\mathcal{F}(q_i, t_j)$ to denote the rank of a tweet $t_j \in \mathcal{T}$. To simplify the discussion, we define dominant set as:

Definition *Dominant Set*
Given a tweet t, a query q and a tweet set $\mathcal{T}$, t's dominant set in relation to q is defined as the tweets that have higher ranks than t, namely

$$ds(q, t) = \{t_i | t_i \in \mathcal{T} \wedge \mathcal{F}(q, t_i) > \mathcal{F}(q, t)\}$$

A straight forward approach would compute t's dominant set for all queries in $\mathcal{Q}$. Algorithm 6.1 illustrates the idea. If there exists a query q_i satisfying $|ds(q_i, t)| < K$, we classify t as a distinguished tweet (line 3-4). Otherwise, it is a noisy tweet. Algorithm 6.1 suffers from two performance problems. First, to compute the dominant set, we need a full scan of the tweet set. Second, given a tweet t, we test it against every query in $\mathcal{Q}$. To address the above two problems, two optimization approaches are proposed respectively. The first optimization approach is based on the Zipf's distribution of the natural language and our theoretical analysis, which will be shown in the rest of this section. The second one is a typical space-for-time optimization: we reduce the time of discovering

the candidate query for a tweet by maintaining an in-memory matrix index.

Algorithm 6.1: NaiveClassifier(Tweet t, QuerySet $\mathcal{Q}$)

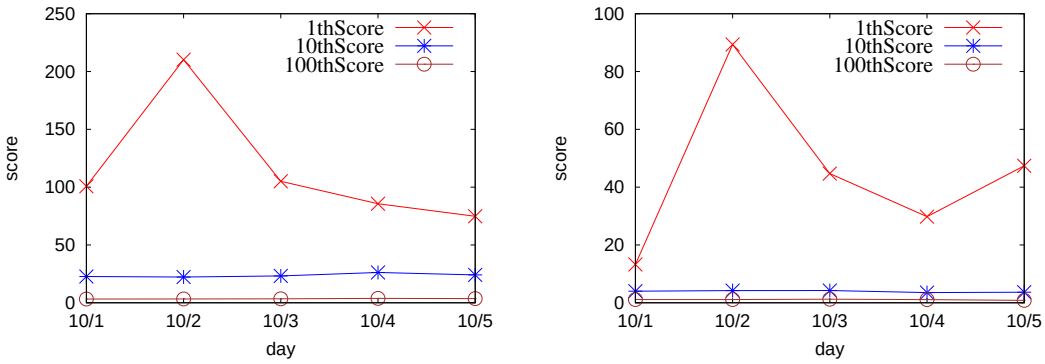
```
for ∀q_i ∈ Q do
    ds(q_i, t)=getDominantSet(Q, t);
    if ds(q_i, t).size< K then
        return distinguished tweet;
return noisy tweet;
```

Optimization 1: Top-K Threshold

The first optimization is to employ the query statistics to speed up the dominant set computation. Figure 6.5 shows the statistics of top-K query results in our Twitter dataset. The X-axis denotes the date of the ranking and the Y-axis is the ranking score computed by our ranking function $\mathcal{F}$. The naive approach is invoked to compute the scores of pair (t_i, q_j), where t_i denotes an existing tweet by that specific day and q_j is a query in $\mathcal{Q}$. In Figure 6.5(a) and 6.5(b), we present the results for the query "coupon" and "database" respectively. Other queries share the same property. In particular, in the figures, we compare the scores of the topmost tweet, the top 10th tweet and the 100th tweet (our threshold). We find that although the score of the topmost tweet varies a lot with time, the scores of the top 10th and 100th tweet are quite stable. This is because in natural language, the words follow Zipf's distribution [74], where each word tends to appear in the text with a certain frequency. Given a query, the expected number of hot tweets remains stable over time. We have the following theorem.

Theorem 6.1. *Suppose each keyword appears in the tweets with a fixed probability and the tweets are inserted into the system with a stable rate. If query q_i has m results ($m >> K$), the variance of top-K score for q_i decreases for a larger K.*

Proof. Suppose we have n tweets and there are m tweets ($m > K$) containing the search keyword. We try to estimate the Kth score of m resultant tweets, assuming they are randomly distributed in the tweet dataset. We sort the tweets by their ranks and have a list $\{t_1, t_2, ..., t_n\}$. The Kth tweet appears in



(a) Query "Coupon" (b) Query "Database"

Figure 6.5: Statistics of Keyword Ranking

the position x with probability of

$$p(x) = \frac{\binom{x-1}{K-1}\binom{n-x}{m-K}}{\binom{n}{k}}$$

And the expectation of top-K score is

$$E(K) = \sum_{i=k}^{n} p(i)score(i)$$

where $score(i)$ denotes the score of the ith tweet. The problem can be transformed into an order statistic problem. Based on the estimated bounds in [25], when m is sufficiently large, we get a more closer bound for $E(K)$ for a larger K. □

The above observation motivates our classification scheme. We keep a top-K threshold for each query $q \in \mathcal{Q}$, which is called threshold table T_θ. Given a query q, $T_\theta(q)$ returns the threshold for the top K tweets.

Lemma 6.1. *For a tweet t, if $\mathcal{F}(q_i, t) < T_\theta(q_i)$, the size of t's dominant set is larger than K at the moment.*

Proof. If $\mathcal{F}(q_i, t) < T_\theta(q_i)$, t's score is smaller than current Kth result. Therefore, more than K tweets have higher ranks than t. □

Theorem 6.2. *For a tweet t, if $\mathcal{F}(q_i, t) < T_\theta(q_i)$ for all $q_i \in \mathcal{Q}$ and $\mathcal{F}(q_i, t)$ decreases with time, t is a noisy tweet.*

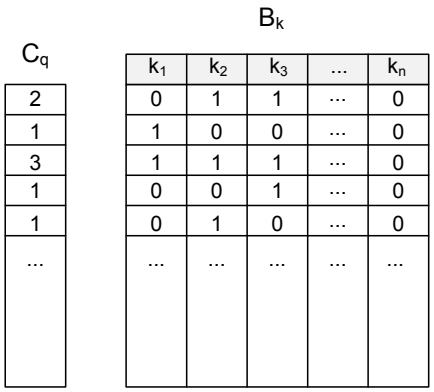


Figure 6.6: Matrix Index

Proof. If $\mathcal{F}(q_i, t)$ decreases with time, the tweet will never be a top-K result for a query. Thus, it is a noisy tweet. □

In Theorem 6.2, we require $\mathcal{F}(q_i, t)$ to be monotonically decreasing with time. In fact, in our ranking function, to catch the hotly discussed topics and discussion trend, $\mathcal{F}(q_i, t)$ may increase for a small number of hot tweets. We shall discuss how to handle such cases in Section 5.2.

T_θ can be constructed and updated by Algorithm 6.2. Initially, T_θ's values are set to 0 for all queries. After a query is processed, we update its threshold based on the query result.

Algorithm 6.2: UpdateThreshold(T_θ, Query q)

```
1 Result R= getTopResult(K, q);
2 if R.size= K then
3 |   Score s = R[K].score;
4 |   T_θ(q) = s;
5 else
6 |   T_θ(q) = 0;
```

Optimization 2: Matrix Index for Queries

As analyzed in optimization 1, instead of computing dominant set for every query, we maintain a top-K threshold T_θ for each of the query. However, for each incoming tweet, we need to compare it with every query in order to find

the queries that have common keywords with this tweet, which is still a time-consuming step. Therefore, our second optimization is to avoid unnecessary comparison between the tweets and the queries. We consider both queries and tweets as a bag of words. To simplify our discussion, we define the candidate query set as follows:

Definition *Candidate Query*
For a tweet $t = \{k_1, k_2, ..., k_n\}$ and a query $q = \{k'_1, k'_2, ..., k'_m\}$, q is a candidate query for t, i.f.f.

$$\forall k_i \in t \rightarrow \exists k'_j \in q \wedge k'_j = k_i$$

Instead of checking every query for an incoming tweet t, we just need to compute $\mathcal{F}(q_i, t)$ for t's candidate queries. To facilitate the discovery of candidate queries, we propose a matrix index.

Figure 6.6 illustrates the index structure. B_k is a $m \times n$ matrix index (m is the size of $\mathcal{Q}$ and n is the number of unique keywords in $\mathcal{Q}$) and C_q is the counter vector for queries. Each row in B_k refers to a query and each column in B_k denotes a keyword. If the jth keyword appears in the ith query, we set $B_k[i][j]$ to 1. Otherwise, it is set to 0. C_q keeps the number of keywords in a query. The ith query has $C_q[i]$ keywords. Given a tweet t, we define its vector as $V_t = (v_1, v_2, ..., v_n)$, where $v_i = 1$ if t contains the ith keyword. Otherwise, $v_i = 0$. To find all candidate queries, we compute an evaluation vector as

$$V_e = V_t \times B_k^T \tag{6.3}$$

where B_k^T is the transpose of B_k. If $V_e[i] = C_q[i]$, then the ith query is a candidate query for tweet t. By applying the matrix index, we transform the discovery process of candidate queries into matrix computation. Because B_k is a sparse matrix, Equation 6.3 can be computed efficiently, which is reflected in our optimized classification algorithm.

Optimized Classifier

Algorithm 6.3 outlines our tweet classification algorithm. It is an evolution from Algorithm 6.1 by combining the two optimization approaches discussed previously. Given a tweet t, we first create a temporary counter for recording the queries that have been processed (line 1). Then we scan each column of

Algorithm 6.3: Classifier(Tweet t, QuerySet $\mathcal{Q}$)

```
1  Array count=0;
2  V_t=getTweetVector(t);
3  for j = 0 to n do
4      if V_t[j] == 1 then
5          for i=0 to m do
6              if B_[i][j] == 1 then
7                  count(j)++;
8                  if count[j] == C_q(j) then
9                      if t's ranking is larger than T_θ(j) then
10                         return distinguished tweet;
11 return noisy tweet;
```

matrix index (line 3-10). Once we detect the keyword is contained in a query (line 6), we will increase the count of the query in the temporary counter. If the counter indicates that all keywords of the queries have been seen (line 8), we will test the tweet's score against the query's threshold (line 8). If larger than the threshold, t is classified as the distinguished tweet.

In Algorithm 6.3, we use a temporary counter to simplify the matrix computation. As an example, in Figure 6.6, suppose a tweet t contains k_1, k_2 and k_3 as the keywords. We will start scanning the columns of the three keywords. By scanning the first column, we know that query q_1 and q_2 contain k_1. And after comparing with the value in counter C_q, we know q_1 is a candidate query, as it only has 1 keyword. Hence, we can compare its threshold with the score of the tweet.

We now discuss the complexity analysis of the above algorithm. Suppose we have m queries and n keywords. We need m bytes for the counter vector C_q and $\frac{nm}{8}$ bytes for the matrix index B_k. The top-K threshold is an array of floats. Therefore, its takes $4m$ bytes. Algorithm 6.3 incurs a storage overhead of

$$S = 5m + \frac{nm}{8} \tag{6.4}$$

As an example, when $m = 100000$ and $n = 5000$, we need approximately 60 MB memory. Suppose the average number of tweet's keywords is x, Algorithm 6.3 scans x columns of B_k. During scanning, instead of testing each bit one by one, we test the whole word. In a W-bit system, the time complexity is $\frac{xm}{W}$.

To further optimize the classification algorithm, we adopt compression technique. For each column in B_k, most bits are 0, as not every query contains the keyword. Therefore, we apply WAH (Word Aligned Hybrid) encoding [106] to compress the index. On average, WAH encoding can reduce the index size by 90%, which significantly reduces the memory overhead of the classification algorithm. We shall further discuss this in the experimental study section.

6.3.2 Implementation of Indexes

For each incoming tweet, we will classify it as a distinguished or noisy tweet, and insert into the index or log file for batch update. We shall present both indexing schemes in this subsection.

Real-Time Indexing

A new tweet that is identified as a distinguished tweet is indexed immediately. The indexing process entails the following steps,

1. If the tweet belongs to an existing tweet tree, we retrieve its parent tweet (1 atomic operation in HBase) to get the root ID and generate the corresponding encoding. Then, we update the *count* number in the parent tweet.

2. The encoding column of the tweet store in HBase is updated correspondingly.

3. Lastly, the tweet is inserted into the inverted index, which incurs a few I/Os depending on the number of keywords in the tweet. This is the dominant component of the indexing cost.

The first step is used to maintain the tree structure of tweets, which may incur one or two database operations. This cost can be saved, if the ranking function does not consider the effect of the tree structure. However, even in our case where the tree structure is used, this is not a major cost. Based on the statistics of [7], less than 23% of the tweets get replies, for which we need to maintain the tree structures. Furthermore, most of the tweets get replies within a relatively short period after posting, and thus, caching the recent tweet records can significantly reduce the cost.

The main overhead of the indexing process is the cost of updating the inverted index. For a given tweet which has n keywords, we need to update n inverted list, one for each keyword. Moreover, to support real-time search, the tweets in the inverted list are sorted by their timestamps. The update and sorting costs dominate the indexing cost.

Batch Indexing

When a noisy tweet is submitted to the microblogging system, instead of indexing it in the inverted index, we append it to the log file. The operation is straight forward, and it incurs one HBase atomic operation. Hence, batch indexing is very efficient compared to the real-time indexing.

Periodically, the batch indexing process scans the log file and indexes the tweets in an offline manner. To reduce the cost of building the inverted index, we build an in-memory inverted index. We maintain an inverted list (a list of document ids and necessary attributes for ranking) for each encountered keyword in memory. If the memory is full, we combine the in-memory inverted index with the disk based index. In this manner, we can significantly reduce the I/Os, as the updates to an inverted list of a keyword can be performed in groups.

6.3.3 Tweet Deletion

In microblogging system, deleting an existing tweet is a common user action. To deal with a tweet deletion, we adopt the standard method that is widely used in search engines [76]. The deletion operation on a tweet is written to a log file in the storage, and the real-time inverted index will be periodically updated based on this deletion log. For a search query, the tid (tweet id) list is retrieved from the real-time inverted index first, and then the content of these tid is retrieved from the storage (R-Store in ART). If a tid does not exist in R-Store anymore, which means that the tweet has been deleted by its owner but the inverted index has not been updated yet, we simply ignore this tid and do not show the tweet in the final search result.

6.4 Ranking Function

In *TI*, the indexing scheme is independent of the ranking function. The user can therefore define different ranking functions. In this section, we propose a computationally efficient and effective ranking function tailored for the social networking systems by exploiting the features of user behaviors. Our proposed ranking function is composed of the user's PageRank, popularity of the topic, the timestamp and the similarity between the query and the tweet.

6.4.1 User's PageRank

To capture the relationships between social networking users, we have a user graph $G_u = (U, E)$ where U denotes all the available users and E describes the links between them. In a system such as Twitter, there are two links defined for a user, the *followers* and *following*. Given a user u, its *followers* is a set of users, who follow u's tweets, while its *following* is another set of users that u currently follows. We use $f(u)$ and $f^{-1}(u)$ to denote the *followers* and *following* set of user u, respectively. For ease of discussion, we define the complete graph as below.

Definition Complete Graph
Graph $G_u = (U, E)$ is a complete graph, i.f.f.
1) $\forall u_i \in U \forall u_j \in f(u) \rightarrow u_j \in U$
2) $\forall u_i \in U \forall u_j \in f^{-1}(u) \rightarrow u_j \in U$

In a complete graph, the *following* link is analogical to the *follower* link. The *follower* graph can be directly constructed by reverse the direction of the *following* graph. Therefore, in the remaining discussion, we only consider the *following* link. We build a matrix M_f to record the *following* links between users. As shown in Figure 6.7, if u_i follows u_j, we set $M_f[i][j]$ to 1. To compute PageRank, we also define a weight vector $V = (w_1, w_2, ..., w_n)$, where w_i is the weight of user u_i. Currently, w_i is set to 1 for all users, by assuming that every user is equally important initially. We then compute the user's PageRank as follows:

$$P_u = V M_f^x \tag{6.5}$$

x keeps increasing, until M_f^x converges. $P_u[i]$ denotes the PageRank value of user u_i. We normalize it as $P_u[i] = \frac{P_u[i]}{\sum_{1 \leq i \leq n} P_u[i]}$.

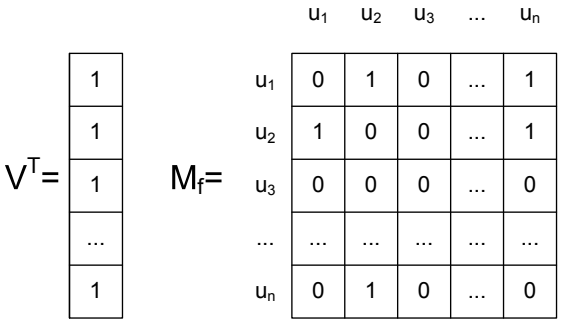


Figure 6.7: Following Matrix

The PageRank values are stored in a user table, which is defined as *(UID, Name, PageRank)*, where UID is the ID of the user. We also have a follower and following table for capturing the friend links. In the ranking function, the tweet inherits the PageRank from its author. In particular, we define the tweet's U-PageRank as

Definition U-PageRank
Suppose the tweet t's author is u, t's U-PageRank is defined as u's PageRank value.

A higher PageRank value indicates that the user has more friends and his tweets are probably more attractive than others. Therefore, we can use U-PageRank to decide whether a tweet is important for the users. In [104], an extended PageRank algorithm is also applied to rank Twitter data.

Computing the user's PageRank is costly. However, the active users in a system tend to be stable over time. Hence, the PageRank is computed in an offline manner. We can periodically, say every ten days, recompute the PageRank values. When a new user joins the system before the next computation, we set its PageRank value to 0.

6.4.2 Popularity of Topics

In Twitter, users retweet tweets of other people to broadcast the tweets to their friends. They also express their own ideas when replying to other's tweets. In *TI*, tweets are grouped into a tree by the retweet/reply links. We define a tweet tree as a discussion topic or thread. To help users retrieve the popular topics, our ranking function is designed to favor the tweet trees with many discussions.

This strategy is also adopted by the news group search [110] and community search [91]. In particular, given a tweet tree $\mathcal{T}$, we define its popularity as:

$$Pop(\mathcal{T}) = \sum_{\forall t_i \in T} t_i.UPageRank \tag{6.6}$$

As a result, the popularity of a tree is equal to the sum of U-PageRank values of all tweets in the tree. For a single node tree, the popularity of the tree is equal to the root's U-PageRank.

The tree's popularity can be computed fairly easily by joining the tweet table and user table. For example, the following query can be used for its computation.

```
SELECT SUM(U.PageRank) as Popularity, tree
FROM tweet T, user U
WHERE T.UID = U.UID
GROUP BY T.tree
```

However, processing such queries is costly, especially for a large-scale Twitter dataset. If we can reduce the number of records that need to be processed, we can effectively speed up the above query.

It is observed that more than 70% of tweets do not get any response (be replied or retweeted) [7]. For a majority of tweets, we do not need to compute the tree popularity, as the single node tree's popularity is equal to the root's U-PageRank, which can be directly obtained from the inverted index. Figure 6.8 verifies our assumption. It shows the changes of popularity values (without normalization). Most tweet trees exhibit the same behavior. When a tweet is published, it probably does not attract the interest of other users right away. As a result, in the first few hours, it has a low popularity. However, if the tweet belongs to a popular topic, the ranking score of this tweet will benefit from the popularity of this topic. The popularity of the corresponding tweet tree increases significantly, until the topic becomes stale some days later. Then, there will be no new tweets in this tree and the popularity remains stable after that.

We call a tweet topic that is being hotly discussed an *Active Tweet Tree*, which is defined as following:

Definition Active Tweet Tree

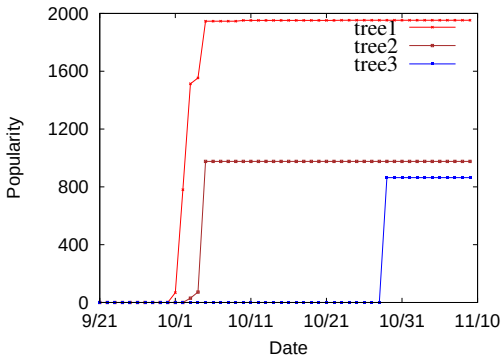


Figure 6.8: Popularity of Topics (computed based on Equation 6.6 by using unnormalized PageRank values)

A tweet tree T is an active tweet tree, if the number of tree nodes keeps on increasing continuously.

For example, in Figure 6.8, tree 1 is an active tweet tree for tweets posted from October 1st to October 3rd. Instead of computing the popularities of all tweet trees, we just compute the popularities of active trees and maintain them in memory. By doing so, we can update the popularities of active trees efficiently when new tweets are submitted. To process the queries, we can look up the popularities kept in memory to rank the tweets.

Algorithm 6.4: isActiveTree(Tweet t)

```
1  ID rid = getRootID(t);
2  if rid is not null then
3      if L_t.containsKey(rid) then
4          L_t(rid).popularity += t.UPageRank;
5          L_t(rid).timestamp = t.timestamp;
6      if t.timestamp-L_c(rid).timestamp> θ then
7          L_c(rid).count = 1
8      else
9          L_c(rid).count++;
10         if L_c(rid).count> γ then
11             L_t.insert(rid, getPopularity(rid), t.timestamp);
12             if some tweets in the tree are not indexed then
13                 create index for the tweets on the fly;
14     L_c(rid).timestamp = t.timestamp;
```

In Algorithm 6.4, we outline the steps entailed in maintaining the active

tree in memory. Initially, all the trees are assumed to be inactive trees. We keep two lists, a candidate tree list L_c and an active tree list L_t, and use hash tables to implement the lists. When a new tweet joins a tweet tree t, we use t's root ID to find its corresponding bucket in L_t and L_c. If t belongs to an active tree, we increase the tree's popularity and reset its timestamp (line 3-5). Otherwise, we retrieve t's record in L_c and compare the timestamp (line 6). If $t.timestamp - L_c(t.rid) > \theta$, we reset the counter to 1 (line 7). Otherwise, we update the timestamp and increase the value of counter by 1 (line 9). If the counter is larger than γ, we promote t as the active tree (line 10). In function $getPopularity(rid)$, we compute the popularity by issuing the query:

SELECT SUM(U.PageRank) as Popularity
FROM tweet T, user U
WHERE T.UID = U.UID AND T.tree= rid

To efficiently process the above query, we build B^+-tree indexes on attribute $T.UID$, $U.UID$ and $T.tree$. Recall that in Theorem 6.2, we require the ranking function to be decreasing with time. But for an active tree, its popularity may increase with time. Therefore, we index all the tweets which are not yet indexed in the active tree (line 12 and 13). This can be done efficiently by following the pointers in the tweet table.

The active tree will be discarded, if it does not obtain any new tweet in more than δ time. In fact, in our ranking function, the popularity of a tree remains steady after a certain time. That is, after δ days, the rank of an inactive tree becomes too small and does not affect the top-K results. In that case, we remove it from L_t. The parameters θ, γ and δ are used to control the accuracy and memory overhead, which can be tuned based on statistics. In our experiment, θ, γ and δ are set to 8 hours, 3 tweets and 10 days respectively.

6.4.3 Time-based Ranking Function

The final part of our ranking function is the similarity between a query q and a tweet t. By using the bag-of-words model, we transform q and t into vectors. Their similarity is estimated as

$$sim(q,t) = \frac{q \times t}{|q||t|} \tag{6.7}$$

The general ranking function combines all the factors and are computed as

$$\mathcal{F}(q,t) = \frac{w_1 \times t.UPageRank + w_2 \times sim(q,t)}{q.timestamp - t.timestamp} + \frac{w_3 \times tree.popularity}{q.timestamp - tree.timestamp} \quad (6.8)$$

where q.timestamp denotes the time when the query is submitted, $tree$.timestamp is the timestamp of the tree that t belongs to (computed as the timestamp of the root node), w_1, w_2 and w_3 are used to normalize the rankings. Currently, w_1, w_2 and w_3 are set to 1, as we treat all factors equally important. If a tweet does not belong to a popular tree, we discard the second term in above formula, as in that case, the popularity should not contribute to its ranking. In our definition, a tweet's ranking is affected by its timestamp. An older tweet is less important than a newly inserted one. When searching, we prefer the latest tweets with high similarity.

6.4.4 Adaptive Index Search

To process a query, the inverted index is employed to retrieve the result tweets based on the scores derived from the ranking function. In our ranking function, the PageRank value, the timestamp and the similarity can be computed based on the information in the inverted index, while the popularity can be obtained by querying the active tree list in memory. Hence, the ranking function is computationally efficient as it does not incur a significant overhead.

Nevertheless, the main problem that affects the search performance is the size of inverted index. Suppose the inverted index for keyword k_i is $\mathcal{I}_i$. The size of $\mathcal{I}_i$ will keep increasing, as more tweets are inserted[1]. To address this problem, we propose an adaptive index searching scheme. The maximal possible score of a tweet at timestamp ts is estimated as:

$$score = \frac{w_1 \times UPageRank_{max} + w_2 + w_3 \times popularity_{max}}{q.timestamp - ts}$$

$UPageRank_{max}$ denotes the maximal user PageRank. We set similarity to 1. And $popularity_{max}$ is estimated by current active tree set. Let S_{tree} de-

[1]In Twitter, only recent tweets can be retrieved and hence, the size of inverted index is reduced

note the active trees that have a timestamp before ts. If no such tree exists, $popularity_{max}$ is set to 0. Otherwise, $popularity_{max}$ equals to the maximal popularity in S_{tree}.

Let $\mathcal{T}_\theta(q)$ be the top-K threshold for query q. Instead of reading the whole inverted index blindly, we iteratively read a block of the index. If the last entry in the block has a timestamp ts and based on the above equation, the maximal score before ts is smaller than $\mathcal{T}_\theta(q)$, we will stop reading the index, since the remaining tweets will not contribute the the search results. This strategy effectively reduces the index search cost.

After the candidate tweets are retrieved from the index, we sort them based on the ranking function. The in-memory sort is efficient, as many tweets have already been pruned by the index searching process. Then we select the top-K results and group them by their tree structures, based on the tree encoding.

6.5 Experimental Evaluation

Table 6.2: Cluster Settings

Parameter	Value
CPU	X3430 2.4 GHz
OS	CentOS 5.5
Memory	8G
Disk	2x500 GB SATA
Default Node Number	25
Data per Node	4G

In this section, we shall evaluate the performance of *TI* indexing scheme and the effectiveness of the propose ranking functions. The cluster settings are shown in Table 6.2. In the experiments, we use a Twitter dataset collected for three years [40] from October 2006 to November 2009. 500 random users are selected from Twitter as the seeds, including politicians, musicians, environmentalists and techies. Following the friend links, more users are discovered and added into the social graph. The total number of involved users is about 465K. For each user, the tweets are crawled every 24 hours. There are more than 25 millions of tweets in the dataset. However, since Twitter does not allow users to crawl their data in a large scale any more, for the scalability

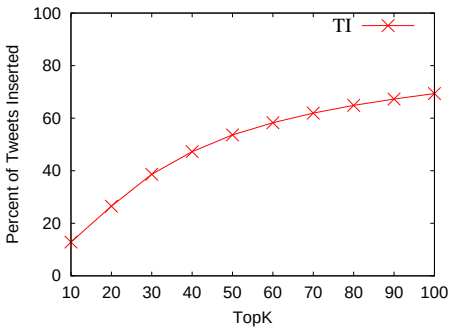


Figure 6.9: Number of Indexed Tweets in Real-Time

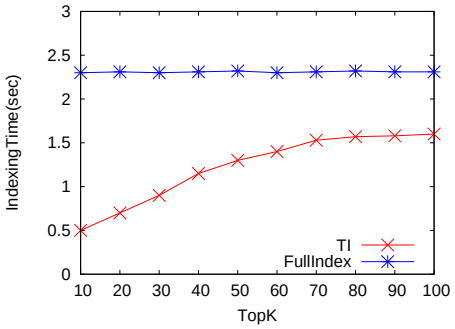


Figure 6.10: Indexing Cost of TI with 5 slaves (per 10,000 tweets)

experiment, we have to generate synthetic data based on this small data set. The size of the data per node is around 4GB.

In the experiments, we start from September 26 2009 and simulate users' behavior for the next ten days. The first five days are used to warm up the system (e.g. building the top-K threshold, learning the popularities of topics). The remaining five days are used to measure the performance. We collect keywords from the first five days' tweets. After removing the keywords in the stop-list and the infrequent words (frequency less than 10), we have less than 5K keywords left. Queries are generated by randomly combining the keywords, and the number of keywords in queries follows Zipf's distribution, where α is set to 1. Approximately, 60% are 1-word queries; 30% are 2-word queries; and 10% are queries with more than two keywords. The queries are submitted to the system at random timestamps, while the tweets are inserted into the system based on their recorded timestamps. The interval for batch indexing is set to one day in this experiment, which is a proper setting based on our observations. When this interval is too large, some important tweets will be missing from the search results. In contrast, if the interval is too small, the batch indexing would affect the performance of the real-time indexing. Each experiment is repeated for ten times and the average result is reported.

6.5.1 Effects of Adaptive Indexing

In the first set of experiments, we study how the adaptive indexing scheme affects the performance. In Figure 6.9, we show the percentage of tweets that are indexed in real-time. When only top-10 results are required, we can prune

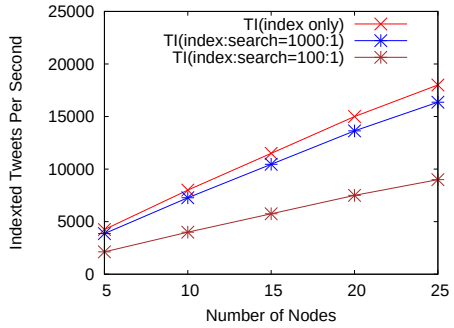


Figure 6.11: Indexing Throughput

more than 80% of tweets (by using batch indexing scheme). As more results are returned to users, more tweets need to be indexed to be searchable. When $K = 100$, about 70% of tweets need to be indexed in real-time. Because only a portion of tweets need to be indexed in real-time, the indexing cost is significantly reduced. Figure 6.10 compares the indexing time of TI and full indexing scheme with 5 TI slaves. In TI, the cost of indexing is proportional to the number of indexed tweets. Therefore, when more tweets are required in the results, TI will incur higher indexing overhead. Figure 6.11 shows the indexing throughput as the number of nodes increases. We evaluate the indexing throughput in three scenarios: (1) only index requests exist in TI; (2) the ratio between the index requests and the search requests is 1000 to 1; (3) the ration between the index requests and the search requests is 100 to 1. As shown in the figure, as the number of nodes increases, the number of tweets that can be indexed in real-time also increases. In addition, since the search requests are more expensive and will compete for computation resources with the index requests, as the ratio between the index requests and the search requests decreases, the indexing throughput decreases as well. We will show the detailed experimental studies of the search queries in Section 6.5.2.

To evaluate whether the adaptive indexing scheme reduces the quality of results, we compute the query accuracy as $\frac{R \bigcap R'}{|R|}$, where R denotes the result set returned by the full indexing scheme (all tweets are inserted in real-time), R' denotes the result set returned by TI, $R \bigcap R'$ represents the number of tweets in both result sets. Figure 6.12 shows the accuracy of TI's results. For comparison, we use two strategies. For the *Constant Threshold*, we do not update the top-K threshold when processing queries. On the contrary, for the

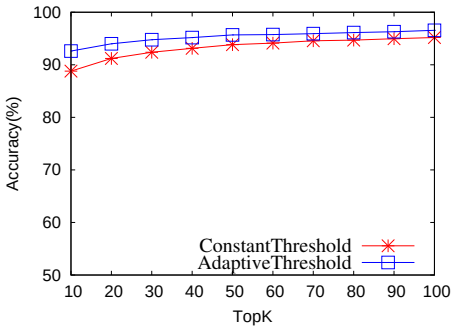


Figure 6.12: Accuracy of Adaptive Indexing

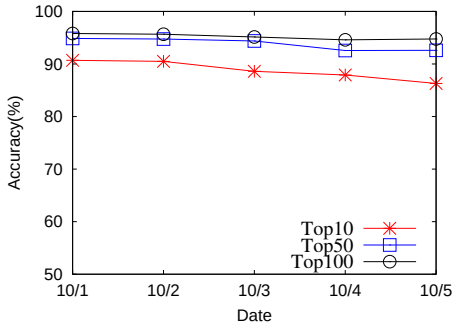


Figure 6.13: Accuracy by Time (constant threshold)

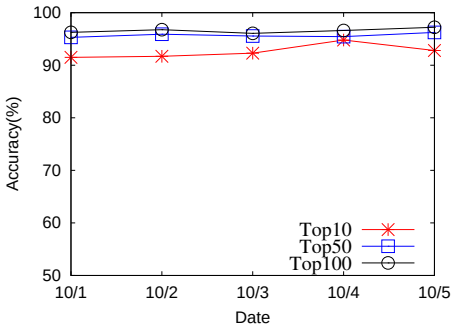


Figure 6.14: Accuracy by Time (adaptive threshold)

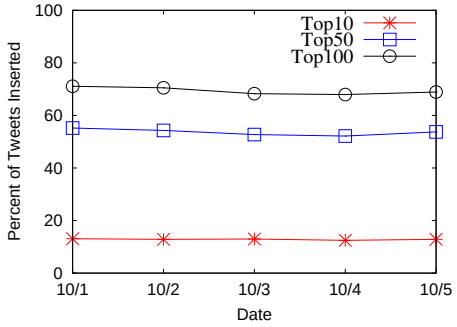


Figure 6.15: Effect of Adaptive Threshold

Adaptive Threshold, we use Algorithm 6.2 to update the threshold adaptively. As shown in Figure 6.12, the accuracy of *Constant Threshold* is just slightly worse than *Adaptive Threshold*. The result verifies our observation made in Figure 6.5, where the top-K threshold remains stable over a period of time. The accuracy of both strategies decreases as K decreases. This can also be observed in Figure 6.5. When K is small, the top-K threshold changes more significantly. An extreme case is when $K = 1$. Thus, the TI may wrongly delay indexing some high ranking tweets. This problem can be fixed by setting a lower bound, e.g. 20, for K. Although user only requests for top 1 result, we always maintain the threshold for top 20 results.

In Figure 6.13 and Figure 6.14, we show the changes of accuracy by dates. The accuracy of *Constant Threshold* degrades, because it never updates its threshold values. However the quality of the results is still acceptable. For the *Adaptive Threshold*, as the threshold is updated by the queries, we always get results with high accuracy. In Figure 6.15, we show the percentage of indexed

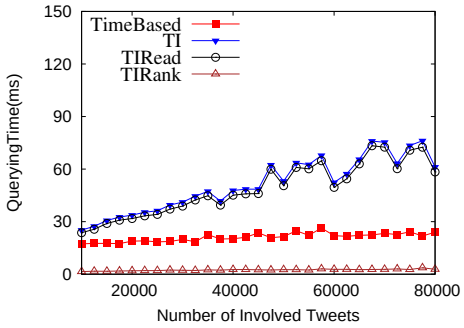


Figure 6.16: Performance of Query Processing (Centralized)

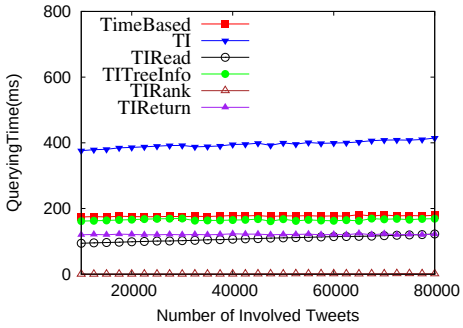


Figure 6.17: Performance of Query Processing (Distributed)

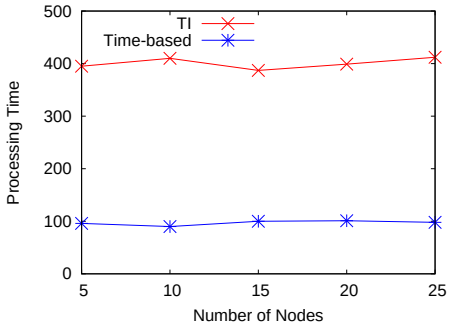


Figure 6.18: Performance of Query Processing

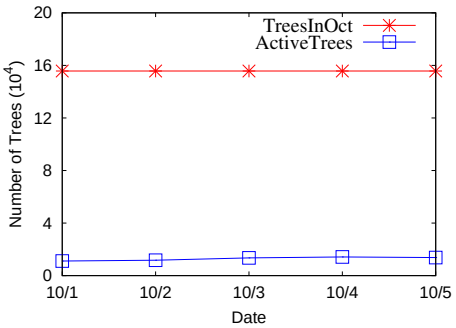


Figure 6.19: Popular Tree in Memory

tweets in *Adaptive Threshold* by dates. We can observe from the figure that the *Adaptive Threshold* scheme does lead to a stable performance, independent of K. As the *Adaptive Threshold* exploits the query results to update its threshold, which is almost free, we will always use *Adaptive Threshold* strategy in the TI indexing scheme.

6.5.2 Query Performance

To provide better search results, TI adopts a sophisticated ranking function. In this experiment, we study whether the ranking function leads to a better query performance. For comparison purposes, we implement a tweet search, which only ranks tweets via their timestamps. Similar ranking strategy seems to have been adopted by Twitter [16] and Google [11]. As we sort the tweets in the inverted index by their timestamps, for a single keyword query, we just need to read the first K entries from the index, which is quite efficient. For a

multi-keyword query, we iteratively read a block of the index for all keywords, and we stop when K results are obtained; Otherwise, more blocks are searched.

Figure 6.16 shows the query performance of the TI and time-based ranking schemes in a centralized mode. TI's costs are decomposed into two parts, the ranking cost $TIRank$ and the index search cost $TIRead$. We group queries by their total number of involved tweets. In Figure 6.16, the X-axis ranges from 0 to 80000, indicating that some popular queries get about 80000 hits in our dataset. Since the size of the inverted index for a keyword k_i is proportional to the number of tweets containing k_i, the index search cost increases as more tweets are involved. This is verified by the results. We have adopted some optimization approaches, such as the adaptive index search outlined in Section 5.4, in order to reduce the cost. As shown in Figure 6.16, $TIRead$ increases linearly with the number of involved tweets. On the contrary, the time-based ranking scheme only retrieves some top tweets, and hence, incurs less overhead. However, it achieves the efficiency by sacrificing the quality of results. Without a reasonable ranking scheme, the query results are less useful.

We also evaluate the performance of TI in a distributed mode with 5 nodes, which is shown in Figure 6.17. In a distributed environment, the main cost is network communicating. In order to reduce the network communicating cost, we modify the ranking scheme in the original centralized TI: for each keyword, instead of retrieving the entire TID list from the distributed index, we only retrieve the top N TIDs, which are ranked by the combination of timestamp and users' PageRank score (TIRead in Figure 6.17, the cost of which increases as the number of tweets involved increases). The query processor then obtains the tree info of each TID from other TI slaves (TITreeInfo), and re-ranks these N TIDs based on the complete ranking function (TIRank). At last, it reads the content of the top k tweets from the distributed key/value stores, HBase, instead of local databases, and return it to users (TIReturn). In contrast, the cost of TimeBased ranking consists of only two part: retrieving the top k TIDs from the distributed lucene and obtaining the tweets for each TID. Figure 6.18 shows the response time of the search queries as the number of nodes increases. As can be seen, the response time does not change much as the number of nodes increases.

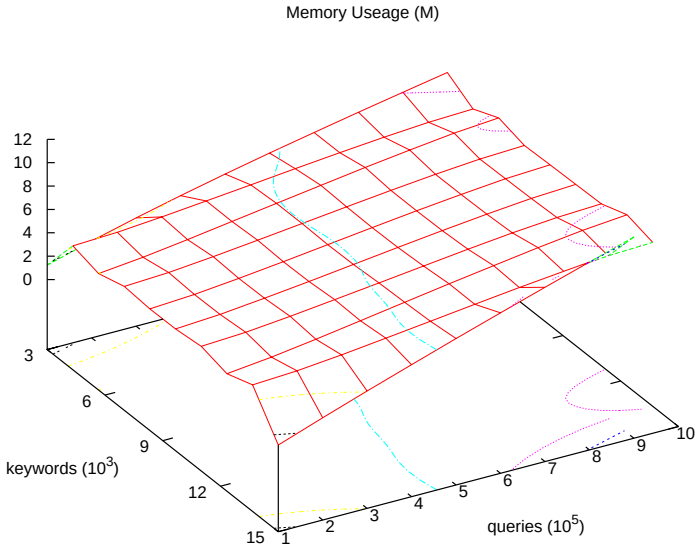


Figure 6.20: Size of In-memory Index

6.5.3 Memory Overhead

In this experiment, we evaluate the memory overhead in each TI slave. We have maintained some memory structures to support adaptive indexing and efficient ranking. Since we partitioned the indexing and retrieval computations to different nodes based on the user graph, the active tree on each TI slave is independent of the active trees on the other slaves. Figure 6.19 shows the total number of active trees. For comparison, we also show the total number of trees generated in October,2009, where less than ten percent of the trees, approximately 13000 trees, are identified as active trees. Moreover, we observe that the number of active trees does not increase with time. In conclusion, the memory requirement is well controlled and is not high.

Another memory structure is the matrix index. Given n keywords and m queries, we need $\frac{nm}{8}$ bytes to maintain the index. To reduce the overhead, we adopt WAH encoding to compress the matrix index. Figure 6.20 shows how the size of in-memory index changes for different n and m. We change the number of keywords from 3000 to 15000 and the number of queries from 100000 to 1 million. The maximum memory usage is only 12 MB, which indicates that the matrix index is very cost-efficient and we can maintain a much larger one for holding more keywords and queries. Another interesting observation is that

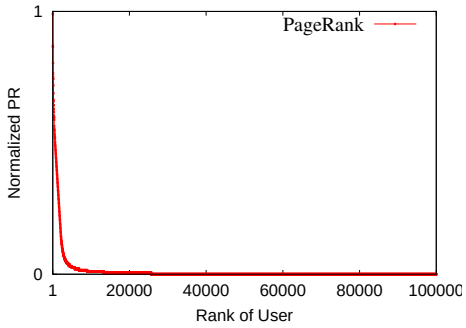


Figure 6.21: Distribution of PageRank

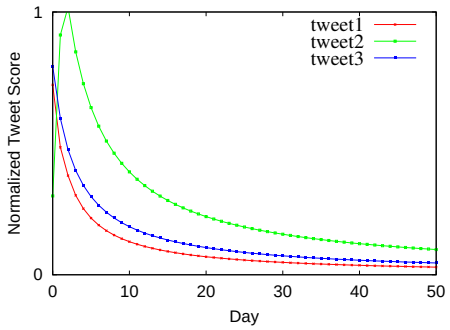


Figure 6.22: Score of Tweets by Time

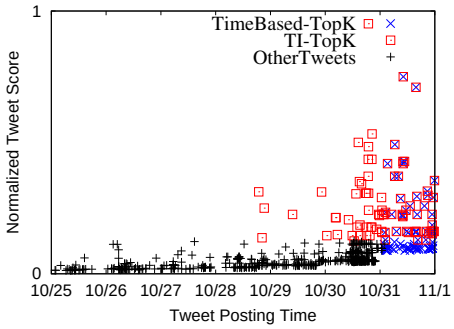


Figure 6.23: Distribution of Query Results

the memory use does not necessarily increase even when more keywords and queries are used. This is because more keywords and queries lead to more 0s and 1s in the matrix index, which improves the compression performance of the WAH.

6.5.4 Ranking Comparison

In the ranking function, we have three components, the similarity between query and tweets, the PageRank of authors and the popularity of topics. Figure 6.21 shows the distribution of users' PageRanks in our dataset. It is not surprising that the PageRank value follows a highly skewed distribution, resembling that of Zipf's or power law distribution. Figure 6.22 shows the effects of time over the score of tweets. In the figure, X-axis represents the elapsed time, where 0 indicates the starting time of the tweets. Y-axis is a score computed by Equation 6.8. In our ranking function, the score is inversely proportional to time. Thus, the score of a specific tweet will decrease with time. However, a

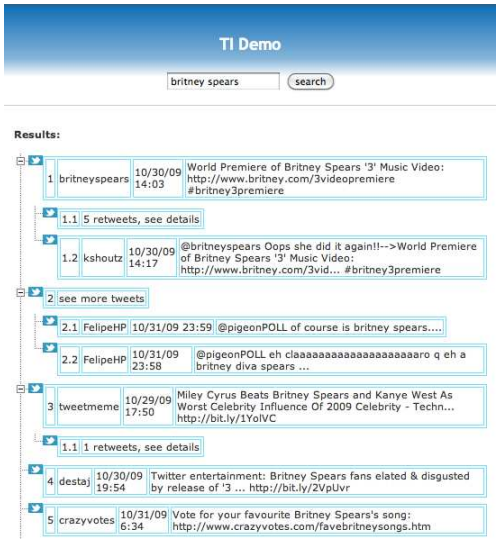


Figure 6.24: Search Result Ranked by *TI*

Figure 6.25: Search Result Ranked by Time

few popular tweets receive many replies within a short period of time after they are posted, contributing to a sudden rise in its score. Figure 6.23 illustrates the scores of the tweets involved with query "Britney Spears". In the figure, the X-axis is the posting time of tweets, while the Y-axis is the score computed by our ranking function. Based on observation of the results, time-based ranking scheme retrieves all recent queries as its top results, while our approach considers both time and other factors, which provides better results.

We show a demo result in Figure 6.24 and Figure 6.25. The search is processed by assuming the time is at *Nov 1, 2009 00:00:00*, when the last tweets in our dataset were crawled (tweets after Nov 1 are considered noisy and pruned). For each result, we show its ranking, author, timestamp and content. In Figure 6.24, we show the result of TI, where tweets are ordered by our ranking function. The first three tweets form a group, as they belong to the same tweet tree. The first tweet is posted by the official account of Britney Spears to publish a new video link. The second one represents 5 retweets. We aggregate them together, for all tweets have the same content. The third tweet is a reply to the first tweet, which shows the song name of the shared video. By grouping tweets via their tree structures, we provide a better visualization result.

In Figure 6.25, we show the result of time-based ranking, where tweets are strictly sorted by their timestamps. This time-based ranking has been adopted

by Palanteer [75] (a microblogging search engine proposed by Ee-peng Lim, etc) and Twitter. As a matter of fact, most results in Figure 6.25 also appear in Figure 6.24. And many results in Figure 6.25 are duplicates. This is because when a hot tweet is published, many users will retweet it within a short time after that. These retweets do not provide any new information, but the time-based ranking will somehow give them a high score. Another problem of the time-based results is the lack of tree structures. Both the first and second tweets are replies to another tweet, but the time-based scoring function shows them individually, while the *TI*'s ranking scheme groups them together, offering a better user experience and more meaningful results.

6.6 Summary

The quest for real-time indexing has recently become more pressing due to the inability of search engines in indexing and retrieving the huge volume of social networking data as soon as they are produced. The problem is further exacerbated by the increasing popularity of microblogging systems where millions of tweets are produced each day. In this chapter, we have proposed *TI*, an adaptive indexing system for supporting real-time search. *TI* adopts an adaptive indexing scheme to reduce the update cost. To this end, a new tweet will be indexed only if it appears in the top-K results of some cached queries. Otherwise, it is grouped with other unimportant tweets, and a batch indexing scheme is used to reduce the indexing latency. *TI* also has a cost-efficient and effective ranking function, by taking the users' PageRank, the popularity of topics, the similarity between the data and the query, and the time into consideration. To evaluate the performance of *TI*'s indexing scheme and ranking function, we conduct an extensive experimental study using a real dataset from Twitter. The experimental results show that *TI* is efficient in handling tweets as they are produced and is able to achieve high query effectiveness and efficiency at the same time.

This work is published as a full paper in *the ACM Special Interest Group on Management of Data* (SIGMOD) 2011 [35].

CHAPTER 7

Conclusion

Increasing data volume in microblogging systems require more scalable framework to process the queries executed in the systems. However, newly emerging "big data" systems such as parallel processing system, distributed key/value stores and real-time search engine have their limitations in efficiently processing the queries. In this thesis, we have designed ART (AQUA, R-Store and TI), a large scale microblogging data management system. We we have consequently proposed three approaches to improve the performance of three types of queries in ART.

First, we have explored the opportunity to efficiently process the multi-way join queries on MapReduce. Our proposed cost model theoretically analyzes the cost of each phase for an equi-join query on MapReduce. By calculating aggregated cost of the equi-join operators in a join tree, the cost of a multi-way join plan can be accurately estimated. We have also investigated how the best plan for the multi-way join is found. By our heuristic plan generating algorithm, the near-optimal plan can be found within an acceptable time. To the best of our knowledge, our cost model and plan generating algorithm is the first work that systematically studies the multi-way join implementations on MapReduce. By integrating the cost-based optimizer in Hive and evaluating the performance on both, we show that the cost-based optimization approach significantly outperforms the exiting rule-based optimization approach.

Second, we have investigated the possibility of supporting real-time aggre-

gation queries in a large scale system and hense propose RStore. In RStore, to support the real-time aggregation, the data are stored with multiple versions, and a snapshot of the versions that contains the most recent updates before the submission time of the query are directly processed by MapReduce. To efficiently obtain the snapshot, a real-time data cube is maintained inside RStore using a streaming approach. When an aggregation query is submitted to RStore, only the real-time data cube and the latest versions of the tuples that are updated after the refresh time of the data cube are shuffled to MapReduce. Furthermore, the global and local compaction schemes greatly reduce the size of data stored in the storage system, and the adaptive incremental scan operation proposed in Chapter 5 significantly improves the performance of scanning the real-time data.

Third, we have designed a new ranking and indexing scheme for the real-time search queries. Compared to the current ranking function which only sorts the result based on uploading time, our ranking function considers the page rank value of the user graph, the ranking score of the entire discussion topic, the relation between the keywords and the tweets and the freshness of the tweets. The result shown in Figure 6.24 demonstrates that the searched results returned by our ranking scheme are more meaningful than the default ranking approach. Moreover, the adaptive indexing scheme proposed in this thesis only indexes the tweets that have high probability to be searched by the search queries in real-time. The other tweets are indexed later with the traditional batch indexing approach. The experimental results show that this method can significantly improve the throughput of the indexing service without losing the quality of the search results much.

7.1 Future Work

Although our first work, AQUA, can efficiently find a near-optimal plan for multi-way join query, the join operator of the join tree is restricted to "=". While the equi-join operator is the most used operator and has attracted most research interest, it would be useful to extend our proposed cost model to support the more general join operator, theta-join. Second, in R-Store, due to the time limit, we only delve in how to efficiently process the real-time aggregation queries. It might be difficult to process the join queries using

exactly the same approaches proposed in this thesis, and supporting real-time processing for more complex queries such as join would be an interesting future work.

In addition to the multi-way join queries, aggregation queries and real-time search queries, there are many other queries and tasks, such as iterative computation and continuous queries, remain to be solved in a microblogging system. For example, for a PageRank computation that requires several iterations of MapReduce jobs, it is not feasible to directly process it using MapReduce. There have been some work on extending MapReduce to support efficient iterative computation (e.g. HaLoop [27]) or designing new systems to handle these queries (e.g. Spark [112]), and it would be timely to address these new challenges within the context of microblogging data management systems.

Bibliography

[1] http://cassandra.apache.org/.

[2] http://hadoop.apache.org/hdfs/.

[3] http://hbase.apache.org/.

[4] http://hstreaming.com/.

[5] http://lucene.apache.org.

[6] http://staff.tumblr.com/post/434982975/a-billion-hits.

[7] http://sysomos.com/insidetwitter/engagement/.

[8] http://thenextweb.com/socialmedia/2010/02/22/ twitter-statistics-full-picture/.

[9] http://wiki.apache.org/hadoop/hive/languagemanual/ joins.

[10] http://www.aster.com.

[11] http://www.google.com/realtime.

[12] http://www.greenplum.com.

[13] http://www.slideshare.net/yousukehara/introduction-of-twitter-gizzard.

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

[15] http://www.tumblr.com.

[16] http://www.twitter.com.

[17] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. Hadoopdb: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *Proc. VLDB Endowment*, 2(1):922–933, August 2009.

[18] Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a map-reduce environment. *EDBT*, 2009.

[19] Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a map-reduce environment. In *Proc. 13th Int. Conf. on Extending Database Technology*, pages 99–110, 2010.

[20] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, pages 496–505, 2000.

[21] Manos Athanassoulis, Shimin Chen, Anastasia Ailamaki, Phillip B. Gibbons, and Radu Stoica. Masm: efficient online updates in data warehouses. In *SIGMOD*, pages 865–876, 2011.

[22] Lars Backstrom, Jon Kleinberg, Ravi Kumar, and Jasmine Novak. Spatial variation in search engine queries. In *WWW*, pages 357–366, 2008.

[23] Elena Baralis, Stefano Paraboschi, and Ernest Teniente. Materialized views selection in a multidimensional database. In *VLDB*, pages 156–165, 1997.

[24] Philip A. Bernstein and Dah-Ming W. Chiu. Using semi-joins to solve relational queries. *J. ACM*, 28(1):25–40, January 1981.

[25] Dimitris Bertsimas, Karthik Natarajan, and Chung-Piaw Teo. Tight bounds on expected order statistics. *Probab. Eng. Inf. Sci.*, 20(4):667–686, 2006.

[26] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A comparison of join algorithms for log

processing in MapReduce. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 975–986, 2010.

[27] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: efficient iterative data processing on large clusters. *Proc. VLDB Endowment*, 3(1-2):285–296, September 2010.

[28] Yu Cao, Chun Chen, Fei Guo, Dawei Jiang, Yuting Lin, Beng Chin Ooi, Hoang Tam Vo, Sai Wu, and Quanqing Xu. Es2: A cloud data storage system for supporting both oltp and olap. ICDE, pages 291–302, 2011.

[29] Stefano Ceri and Jennifer Widom. Deriving production rules for incremental view maintenance. In *VLDB*, pages 577–589, 1991.

[30] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI*, pages 205–218, 2006.

[31] Biswapesh Chattopadhyay, Liang Lin, Weiran Liu, Sagar Mittal, Prathyusha Aragonda, Vera Lychagina, Younghee Kwon, and Michael Wong. Tenzing: A SQL implementation on the MapReduce framework. *Proc. VLDB Endowment*, 4(12):1318–1327, 2011.

[32] Surajit Chaudhuri. An overview of query optimization in relational systems. In *PODS*, pages 34–43, 1998.

[33] Surajit Chaudhuri and Gerhard Weikum. Rethinking database system architecture: Towards a self-tuning RISC-style database system. In *Proc. 26th Int. Conf. on Very Large Data Bases*, pages 1–10, 2000.

[34] Chun Chen, Gang Chen, Dawei Jiang, Beng Chin Ooi, Hoang Tam Vo, Sai Wu, and Quanqing Xu. Providing scalable database services on the cloud. pages 1–19, 2010.

[35] Chun Chen, Feng Li, Beng Chin Ooi, and Sai Wu. Ti: An efficient indexing mechanism for real-time search on tweets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 649–660, New York, NY, USA, 2011. ACM.

[36] Gang Chen, Hoang Tam Vo, Sai Wu, Beng Chin Ooi, and M. Tamer Özsu. A framework for supporting DBMS-like indexes in the cloud. *PVLDB*, 4(11):702–713, 2011.

[37] Ming-Syan Chen, Philip S. Yu, and Kun-Lung Wu. Optimization of parallel execution for multi-join queries. *IEEE Trans. on Knowl. and Data Eng.*, 8(3):416–428, 1996.

[38] Songting Chen. Cheetah: A high performance, custom data warehouse on top of MapReduce. *Proc. VLDB Endowment*, 3(2):1459–1468, 2010.

[39] Rada Chirkova, Chen Li, and Jia Li. Answering queries using materialized views with minimum size. *The VLDB Journal*, 15(3):191–210, 2006.

[40] M. D. Choudhury, Y-R. Lin, H. Sundaram, K. S. Candan, L. Xie, and A. Kelliher. How does the sampling strategy impact the discovery of information diffusion in social media? In *ICWSM*, 2010.

[41] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. Technical Report UCB/EECS-2009-136, EECS Department, University of California, Berkeley, Oct 2009.

[42] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *NSDI*, pages 313–328, 2010.

[43] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.

[44] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. 6th USENIX Symp. on Operating System Design and Implementation*, pages 137–150, 2004.

[45] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.

[46] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *Proc. VLDB Endowment*, 3(1):518–529, 2010.

[47] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. Sap hana database: data management for modern business applications. *SIGMOD Rec.*, 40(4):45–51, January 2012.

[48] Michael J. Franklin, Björn Thór Jónsson, and Donald Kossmann. Performance tradeoffs for client-server query processing. *SIGMOD Rec.*, 25(2):149–160, 1996.

[49] Eric Friedman, Peter Pawlowski, and John Cieslewicz. SQL/MapReduce: a practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proc. VLDB Endowment*, 2:1402–1413, August 2009.

[50] Amol Ghoting, Rajasekar Krishnamurthy, Edwin P. D. Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. SystemML: Declarative machine learning on mapreduce. In *Proc. 27th Int. Conf. on Data Engineering*, pages 231–242, 2011.

[51] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[52] Lukasz Golab, Theodore Johnson, and Vladislav Shkapenyuk. Scheduling updates in a real-time stream warehouse. ICDE, pages 1207–1210, 2009.

[53] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, June 1993.

[54] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[55] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally (extended abstract). In *SIGMOD*, pages 157–166, 1993.

[56] Sándor Héman, Marcin Zukowski, Niels J. Nes, Lefteris Sidirourgos, and Peter Boncz. Positional update handling in column stores. In *SIGMOD*, pages 543–554, 2010.

[57] iProspect. iprospect search engine user behavior study.

[58] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. pages 59–72, 2007.

[59] Akshay Java, Xiaodan Song, Tim Finin, and Belle Tseng. Why we twitter: understanding microblogging usage and communities. In *WebKDD*, pages 56–65, 2007.

[60] Jeffrey Jestes, Ke Yi, and Feifei Li. Building wavelet histograms on large data in mapreduce. *Proc. VLDB Endowment*, 5(2):109–120, October 2011.

[61] Yuntao Jia. Running TPC-H queries on Hive. Available at: http://issues.apache.org/jira/browse/HIVE-600 (Accessed on 25 June 2012.), 2009.

[62] David Jiang, Anthony K. H. Tung, and Gang Chen. MAP-JOIN-REDUCE: Toward scalable and efficient data analysis on large clusters. *IEEE Trans. Knowl. and Data Eng.*, 23(9):1299–1311, 2011.

[63] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. The performance of MapReduce: An in-depth study. *Proc. VLDB Endowment*, 3(1):472–483, 2010.

[64] Thomas Jorg and Stefan Dessloch. Near real-time data warehousing using state-of-the-art etl tools. In *Enabling Real-Time Business Intelligence*, volume 41, pages 100–117. 2010.

[65] Daniel M. Kane, Jelani Nelson, and David P. Woodruff. An optimal algorithm for the distinct elements problem. PODS '10, pages 41–52.

[66] Alfons Kemper and Thomas Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 195–206, Washington, DC, USA, 2011. IEEE Computer Society.

[67] Marcel Kornacker and Justin Erickson. Cloudera impala: real-time queries in apache hadoop, for real, 2012.

[68] Tei-Wei Kuo, Yuan-Ting Kao, and Chin-Fu Kuo. Two-version based concurrency control and recovery in real-time client/server databases. *IEEE Trans. Comput.*, 52(4):506–524, April 2003.

[69] Avinash Lakshman and Prashant Malik. Cassandra: structured storage system on a p2p network. In *PODC*, page 5, 2009.

[70] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, 2010.

[71] Ki Yong Lee and Myoung Ho Kim. Efficient incremental maintenance of data cubes. In *VLDB*, pages 823–833, 2006.

[72] Feng Li, Beng Chin Ooi, M. Tamer Özsu, and Sai Wu. Distributed data management using mapreduce. *ACM Comput. Surv.*, 46(3):31:1–31:42, January 2014.

[73] Feng Li, M. Tamer Özsu, Gang Chen, and Beng Chin Ooi. R-store: A scalable distributed system for supporting real-time analytics. In *Proc. 30th Int. Conf. on Data Engineering*, 2014.

[74] Wentian Li. Random texts exhibit zipf's-law-like word frequency distribution. *IEEE Transactions on Information Theory*, pages 1842–1845, 1992.

[75] Ee-Peng Lim and Palakorn Achananuparp. Palanteer: A search engine for community generated microblogging data. In *ICADL*, pages 239–248, 2012.

[76] Lipyeow Lim, Min Wang, Sriram Padmanabhan, Jeffrey Scott Vitter, and Ramesh Agarwal. Efficient update of indexes for dynamically changing web documents. *World Wide Web*, 10(1):37–69, 2007.

[77] Boon Thau Loo, Joseph M. Hellerstein, Ryan Huebsch, Scott Shenker, and Ion Stoica. Enhancing p2p file-sharing with an internet-scale query processor. In *VLDB*, pages 432–443, 2004.

[78] Inderpal Singh Mumick, Dallan Quass, and Barinderpal Singh Mumick. Maintenance of data cubes and summary tables in a warehouse. In *SIGMOD*, pages 100–111, 1997.

[79] Arnab Nandi, Cong Yu, Philip Bohannon, and Raghu Ramakrishnan. Distributed cube materialization on holistic measures. In *ICDE*, pages 183–194, 2011.

[80] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *ICDMW*, pages 170–177, 2010.

[81] Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. MRShare: Sharing across multiple queries in MapReduce. *Proc. VLDB Endowment*, 3(1):494–505, 2010.

[82] Alper Okcan and Mirek Riedewald. Processing theta-joins using MapReduce. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 949–960, 2011.

[83] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.

[84] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Springer, 3 edition, 2011.

[85] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. In *Technical Report, Stanford University*, 1998.

[86] Viswanath Poosala, Peter J. Haas, Yannis E. Ioannidis, and Eugene J. Shekita. Improved histograms for selectivity estimation of range predicates. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pages 294–305, New York, NY, USA, 1996. ACM.

[87] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill higher education. McGraw-Hill Education, 2003.

[88] Takeshi Sakaki, Makoto Okazaki, and Yutaka Matsuo. Earthquake shakes twitter users: real-time event detection by social sensors. In *WWW*, pages 851–860, 2010.

[89] Jagan Sankaranarayanan, Hanan Samet, Benjamin E. Teitler, Michael D. Lieberman, and Jon Sperling. Twitterstand: news in tweets. In *GIS*, pages 42–51, 2009.

[90] Donovan A. Schneider and David J. Dewitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 110–121, 1989.

[91] Jangwon Seo, W. Bruce Croft, and David A. Smith. Online community search using thread structure. In *CIKM*, pages 1907–1910, 2009.

[92] Kuznecov Sergey and Kudryavcev Yury. Applying map-reduce paradigm for parallel closed cube computation. In *DBKDA*, pages 62–67, 2009.

[93] Praveen Seshadri and Arun N. Swami. Generalized partial indexes. In *ICDE*, pages 420–427, 1995.

[94] Adam Silberstein, Jeff Terrace, Brian F. Cooper, and Raghu Ramakrishnan. Feeding frenzy: selectively materializing users' event feeds. In *SIGMOD*, pages 831–842, 2010.

[95] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Heuristic and randomized optimization for the join ordering problem. *VLDB Journal*, 6:191–208, 1997.

[96] M. Stonebraker. The case for partial indexes. *SIGMOD Rec.*, 18(4):4–11, 1989.

[97] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). pages 1150–1160, 2007.

[98] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: a column-oriented dbms. In *VLDB*, pages 553–564, 2005.

[99] Aixin Sun, Meishan Hu, and Ee-Peng Lim. Searching blogs and news: a study on popular queries. In *SIGIR*, pages 729–730, 2008.

[100] Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and querying ordered xml using a relational database system. In *SIGMOD*, pages 204–215, 2002.

[101] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - a warehousing solution over a map-reduce framework. *Proc. VLDB Endowment*, 2(2):1626–1629, 2009.

[102] Panos Vassiliadis and Alkis Simitsis. Near real time ETL. In *Annals of Information Systems*, volume 3, pages 1–31. 2009.

[103] Jinbao Wang, Sai Wu, Hong Gao, Jianzhong Li, and Beng Chin Ooi. Indexing multi-dimensional data in a cloud system. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 591–602, 2010.

[104] Jianshu Weng, Ee-Peng Lim, Jing Jiang, and Qi He. Twitterrank: finding topic-sensitive influential twitterers. In *WSDM*, pages 261–270, 2010.

[105] Colin White. Intelligent business strategies: Real-time data warehousing heats up. *DM Review*, 2012.

[106] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. Compressing bitmap indexes for faster search operations. In *SSDBM*, pages 99–108, 2002.

[107] Sai Wu, Dawei Jiang, Beng Chin Ooi, and Kun-Lung Wu. Efficient B-tree based indexing for cloud data processing. *Proc. VLDB Endowment*, 3(1):1207–1218, 2010.

[108] Sai Wu, Feng Li, Sharad Mehrotra, and Beng Chin Ooi. Query optimization for massively parallel data processing. In *Proc. 2nd ACM Symp. on Cloud Computing*, pages 12:1–12:13, 2011.

[109] Sai Wu, Jianzhong Li, Beng Chin Ooi, and Kian-Lee Tan. Just-in-time query retrieval over partially indexed data on structured p2p overlays. In *SIGMOD*, pages 279–290, 2008.

[110] Wensi Xi, Jesper Lind, and Eric Brill. Learning effective ranking functions for newsgroup search. In *SIGIR*, pages 394–401, 2004.

[111] Jian Yang, Kamalakar Karlapalem, and Qing Li. Algorithms for materialized view design in data warehousing environment. In *VLDB*, pages 136–145, 1997.

[112] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. 2010.

[113] Xiaofei Zhang, Lei Chen, and Min Wang. Efficient multi-way theta-join processing using MapReduce. *Proc. VLDB Endowment*, 5(11):1184–1195, 2012.