# SUPPORTING EFFICIENT DATABASE PROCESSING IN MAPREDUCE

LU PENG

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 also not been submitted for any degree in any university previously.

Lu Peng 31 July 2014

# ACKNOWLEDGEMENT

With immense gratitude, I acknowledge my advisor, Professor Beng Chin Ooi, for providing continuous support, guidance, mentoring, and technical advise over the course of my doctoral study. I still remember the day in the Winter quarter of 2009 when I first met Professor Ooi discussing the possibilities of joining his research group. I had barely completed my first course on Distributed Systems and had superficial knowledge of Database Systems. On that day, I never imagined that five years down the line, I will be writing my dissertation on a topic that marries these two research areas. Professor Ooi's research insights have made this dissertation possible. Besides his great guidance on the technical side, I will never forget his kind fatherly and friendly attitude. His characters will continue to inspire me.

I would like to thank Dr. Divesh Srivastava, Dr. Lucasz Golab, and Dr. Philip Korn for their invaluable guidance during my internship in AT&T Research Labs. It was a wonderful and memorable summer with them. First time in my life I had the chance to meet some of the brightest minds on the planet who have their own wiki page.

I am grateful to my thesis committee, Professor Mong Li Lee, Professor Stephane Bressan, and the external examiner, for their insightful comments and suggestions to this thesis. Their comments helped me improve the presentation of this thesis in many aspects.

I would like to express my thanks to the collaborators during my Ph.D study, especially Professor Kian-Lee Tan. Dr. Sai Wu, Dr. Hoang Tam Vo, Dr. Dawei Jiang, and Dr. Wei Lu, for the helpful discussion and suggestions to my research work.

I am also thankful to all my friends for the fun moments in my PhD student life. Special thanks to Feng Li, Chen Liu, Xuan Liu, Feng Zhao, and Zhan Su for the wonderful moments we shared in the lab. I also thank my other past and present DB-Lab colleagues: Qiang Fu, Dongxiang Zhang, Su Chen, Jingbo Zhang, Shanshan Ying, Weiwei Hu and Chang Yao. I will also cherish the good times spent with my friends during my stay

in Singapore. Special thanks to Pangge (Hanwang Zhang), Jiangbo Yu, Xiaohu Zhang, Yongning Lu and the rest of the NUS gang for the wonderful moments.

Most importantly, my deepest gratitude is for my family for their constant support, inspiration, guidance, and sacrifices. My father and mother are constant source of motivation and inspiration. Words fall short here.

# CONTENTS

# ABSTRACT

Cloud computing has emerged as a multi-billion dollar industry and as a successful paradigm for web-scale application deployment. Represented by the MapReduce processing model, MPP (Massively Parallel Processing) systems form a critical component of the cloud software stack. Hailed for its high scalability, massive parallelism, and effectively programable interface, the MapReduce paradigm is widely recognized as a revolutionary advancement in large scale computation. However, due to the heterogeneity and massiveness nature of data in the Cloud, current Cloud systems trade rigorous data management functionalities for better versatility and scalability. On one hand, the absence of comprehensive data model and access methods, which have been developed extensively for relational database management systems (RDBMSs), has affected MapReduce-based system's applicability to a wider variety of real world analytical tasks. On the other hand, due to the complexity of processing logic layers in its system architecture, RDBMSs fail to provide desirable scalability and elasticity.

The overarching goal of this dissertation is to exploit the opportunity for a better marriage of RDBMS technologies and Cloud Computing systems. This dissertation shows that *with careful choice of design and features, it is possible to architect a large scale system that syncretizes the efficient access methods of RDBMS and the powerful parallelized processing of MapReduce.* This dissertation advances the research in this topic by improving two critical facets of large scale data processing systems. First, we propose an architecture to support the usage of DBMS-like indexes in MapReduce systems to facilitate the storage and processing of structured data. We start with devising a bitmap-based indexing scheme that provides superior space efficiency, and improves the performance of MapReduce programs on a specific category of data. We then generalize the index application, and propose a generalized index framework for MapReduce systems to handle large data and applications. Second, we propose models and techniques to incorporate the power of MapReduce with parallel database system technologies in query processing.

# LIST OF TABLES

# LIST OF FIGURES

# CHAPTER 1

# Introduction

## 1.1 Cloud Computing

We are in an era of Cloud.

With the irresistible trend of digitalization, the volume of data generated from online and off-line has reached an unprecedented scale. The emergence of Cloud Computing is a timely and practical response to the storage and processing demand in large scale computation. The Cloud has revolutionized the way computing infrastructure is abstracted and used. Analysts project the global cloud computing services revenue is worth tens of billion dollars and is growing [86]. The major features that make cloud computing an attractive service oriented architecture are: **elasticity**, i.e., the ability to scale the resources and capacity on-demand; **pay-as-you-go pricing** resulting in low upfront investment and low time to market for trying out novel application ideas; and the **transfer of risks** from the small application developers to the large infrastructure providers. Many novel application ideas can therefore be tried out with minimal risks, a model that was not economically feasible in the era of traditional enterprise infrastructures. This has resulted in large numbers of applications – of various types, sizes, and requirements – being deployed across the various cloud service providers.

Three cloud abstractions have gained popularity over the years. Infrastructure as a service (**IaaS**) is the lowest level of abstraction where raw computing infrastructure (such as CPU, memory, storage, network etc.) is provided as a service. Amazon web service (http://aws.amazon.com/) and Rackspace (http://www.rackspace.com/) are example IaaS providers. Platform as a service (**PaaS**) constitutes the next higher level of service abstraction where a platform for application deployment is provided as a service.

Applications are hosted and managed by a PaaS provider's platform throughout their lifecycles. Microsoft Azure (http://www.microsoft.com/windowsazure), Google AppEngine (http://code.google.com/appengine/), Engine Yard (http://www.engineyard.com/), and Facebook's developer platform (http://developers.facebook.com/) are example PaaS providers. Software as a Service (**SaaS**) is the highest level of abstraction where a complete application is provided as a service. A SaaS provider typically offers a generic application software targeting a specific domain (such as a customer relationship management, property management, payment processing and checkout, etc.) with the ability to support minor customizations to meet customer requirements. Google Apps for Business and Enterprises (http://www.google.com/enterprise/apps/business/), Salesforce.com (http://www.saleforce.com/), Akamai (http://www.akamai.com/), and Oracle's on demand CRM (http://www.oracle.com/us/products/applications/crmondemand/index.html) are example SaaS providers. The concept of service oriented computing abstractions can also be extended to Database as a Service, Storage as a service, and many more.

## 1.2 Motivations and Challenges

Irrespective of the cloud abstraction, data is central to applications deployed in the cloud. Data drives knowledge which engenders innovation. Be it personalizing search results, recommending movies or friends, determining which advertisements to display or which coupon to deliver, data is central in improving customer satisfaction and providing a competitive edge. Data, therefore, generates wealth and many modern enterprises are collecting data at the most detailed level possible, resulting in massive and ever-growing data repositories. Database management systems (DBMSs) therefore form a critical component of the cloud software stack.

Relational database management systems (**RDBMSs**) have been the solution to most of the data needs for the past few decades; such systems include both commercial (such as Oracle Database, IBM DB2, Microsoft SQL Server, etc.) and open source (such as MySQL, Postgres, etc.) systems. These systems have been extremely successful in classical enterprise settings. Some of the key features of RDBMSs are: *rich functionality*, i.e., handling diverse application workloads using an intuitive relational data model and a declarative query language; *high performance* by leveraging over three decades of performance optimizations; *data consistency*, i.e., dealing with concurrent workloads while guaranteeing that data integrity is not lost; and *high reliability and durability*, i.e., ensuring safety and persistence of data in the presence of different types failures.

In spite of the success of RDBMSs in conventional enterprise infrastructures, they are often considered to be less "cloud friendly" [82]. This is because *scaling* the databases

*on demand* while providing guarantees competitive with RDBMSs and ensuring high data availability in the presence of failures is a hard problem. The problem of *scaling* is primarily attributed to the complex software stack of database systems, and stringent ACID requirement. The database servers have to store a lot of tightly coupled states while guaranteeing stringent ACID properties and supporting concurrent access. Historically, there have been two approaches to scalability: scaling-up and scaling-out.

**Scaling-up**, i.e., using larger and more powerful servers, has been the preferred approach to scale databases in enterprise infrastructures. This allows RDBMSs to support a rich set of features and stringent guarantees without the need for expensive distributed synchronization. However, scaling-up is not viable in the Cloud primarily because the cost of hardware grows non-linearly, thus failing to leverage the economies achieved from commodity servers.

**Scaling-out**. i.e., increasing system's capacity by adding more (commodity) servers, is the preferred approach in the Cloud. Scaling-out minimizes the total system cost by leveraging commodity hardware and the *pay-as-you-go* pricing. Scaling out RDBMSs, while supporting flexible functionality, however, is expensive due to distributed synchronization and the cost of data movement for transactions whose execution cannot be contained in a single node[1]. Moreover, managing RDBMS cluster installations is a major engineering challenge with high administration cost [47].

Unfortunately, the rapid growth of the amount of information has outpaced the processing and I/O capabilities of single machines – even those of high-end servers. As a result, more and more organizations have to **scale out** their computations across clusters, and the emergence of Cloud Computing technologies is a response to this demand. The essence of a Cloud Computing system is to create a distributed cluster environment by leveraging massive commodity servers to achieve high scalability, elasticity, and fault tolerance. Although distributed systems have been studied and practiced for decades, the new Cloud paradigm enables efficient massively parallel processing (MPP) by encapsulating failure recovery, inter-machine communication in an execution engine, and bringing about programmability for upper layer applications. Example practises of Cloud MPP systems are Google's MapReduce [30], Microsoft's Dryad [48], Yahoo!'s Pig Latin [72], and their variants.

Diverse applications deployed in Cloud infrastructures result in very different schemas, workload types, and data access patterns, which requires the Cloud system to be able to efficiently store and process heterogeneous data, and adapt to different workloads. Unlike data processing in RDBMSs, the power of MapReduce programming model comes from

[1]We use the term **node** to represent a single server in a distributed system. These two terms, node and server, are used interchangeably throughout this dissertation.

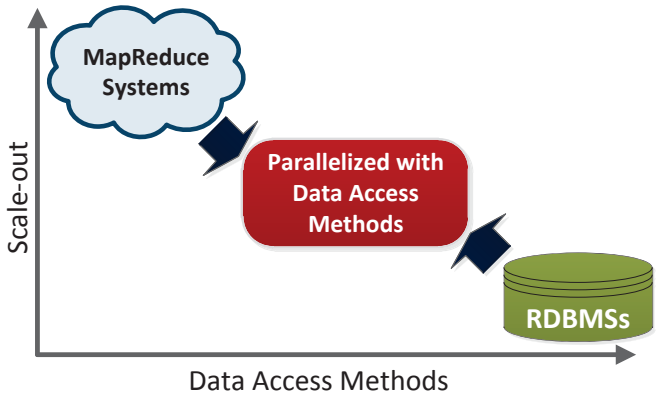



Figure 1.1: Scaling-out while providing data access functionalities. MapReduce systems are designed for large scale operations but support limited schema semantics while RDBMSs provide comprehensive data access methods. This dissertation bridges this chasm.

its simplicity – it provides simple model through which users are able to express relatively sophisticated distributed programs. But as all the good things in the world, this simplicity comes with a price. Due to the heterogeneity and massiveness nature of data stored in the system, most MapReduce systems employ a distributed file system as the storage layer, and data are mostly imported directly from sources and barely parsed using schema. As pointed out by Dewitt and Stonebreaker [34], MapReduce lacks many of the features that have been proven invaluable for structured data analysis workloads, and its immediate gratification paradigm precludes some of the long term benefits of first modeling and loading data before processing. The potential performance drawback of MapReduce has been reported [76] on the basis of experiments on two benchmarks – TPC-H and a customized benchmark tailored for search engines.

As a result, there exist a big chasm between RDBMSs that provide comprehensive data access methods (such as index, etc.) but are hard to scale-out and MapReduce systems that leverage parallelism but support limited schema semantics. Figure 1.1 depicts this balance between scale-out and data access functionalities. It is therefore critical to rethink the design of large scale data processing systems, that has the capability to *scale-out* while *providing comprehensive data access methods*.

## 1.3 Dissertation Overview

The overarching goal of this dissertation is to exploit the opportunity for a better integration of DBMS technologies and Cloud Computing systems. The underlying thesis of this dissertation is that *with careful choice of design and features, it is possible to ar-*

*chitect a large scale system that syncretize the efficient access methods of RDBMS and the powerful parallelized processing of MapReduce.* Using this principle as the cornerstone, this dissertation advances the state-of-the-art by improving two critical facets of large scale data management systems. First, we propose architectures and abstractions to support DBMS-like index in MapReduce systems to facilitate the storage and processing of structured data. Second, we propose models and techniques to incorporate the power of MapReduce with state-of-the-art parallel database system technologies in query processing. The prototype we build approaches parallel databases in performance and efficiency, yet still yields the scalability, fault tolerance, and flexibility of MapReduce-based systems.

### 1.3.1 Indexing the Cloud

The advent of cloud computing marked the beginning of global transformation in how data is created, shared, stored and archived. The explosion of data not only puts challenges on the storage capacity of current large scale systems, but also on their ability to efficiently process the data to uncover the hidden value. Analytical insight is critical from cutting-edge data-driven business to traditional industries, and using the immense volume of data in the Cloud to gather and derive meaningful knowledge creates a unique ground for Cloud analytical technologies to realize value. For example, retailers can track user web clicks to identify behavioral trends that improve campaigns, pricing and stockage. Governments and even Google can detect and track the emergence of disease outbreaks via social media signals. Oil and gas companies can take the output of sensors in their drilling equipment to make more efficient and safer drilling decisions. A recent study reports that the global Cloud analytics market is expected to grow from $5.25 billion in 2013 to $16.52 billion by 2018 [65].

Conventional RDBMSs organize data in the relational data model, provide comprehensive storage and query optimization, and a declarative query language (SQL). As a result, when an RDBMS is scaled-out and distributed over a cluster of servers, the bulky system incurs expensive management overhead and performance degradation. While not being able to be adopted as a whole, RDBMSs have a lot of nice features that can be "partially" applied to the Cloud to reinforce its functionality. Data access methods, among all, are what current Cloud systems fail to facilitate.

The most prevalent data access technique employed by conventional RDBMSs is **indexing**. By organizing a target attribute (table column) into a search friendly structure (index), an indexing technique is able to provide fast location of desired data without having to scan the whole database, and accelerate data retrieval. Ideally, indexing techniques are able to effectively speed up data retrieval in large scale systems, however, applying index in MapReduce is non-trivial mainly because of two reasons: (1) MapReduce does not

have built-in support for processing traditional index, and (2) scaling traditional indexes in a distributed environment is difficult due to undesirable maintenance and tuning overheads. Given the necessity and current absence of effective index application in the Cloud, we present the design of two index mechanisms tailored for large scale data processing systems.

The choice of an appropriate index for data with certain characteristic has decisive impact on query performance. For instance, in an update intensive environment, an LSM-Tree [73] serves better than $B^+$-Tree index. If we have a highly selective workload on wide range of numeric data, then $B^+$-Tree is preferable. In this thesis, we first investigate a specific category of data, namely, data with limited range of value. Bitmap index is traditionally employed to index data with such characteristic. More importantly, the space efficiency of bitmap index makes it a promising candidate for supporting retrieval over large scale datasets. Consequently, we propose **BIDS** [62], a bitmap based indexing scheme for large-scale data store. Our study shows that, the proposed bitmap index scheme effectively reduces the space overhead of indexing large volume of data by incorporating state-of-the-art bitmap compression techniques, such as WAH encoding [100] and bit-sliced encoding [83]. BIDS also adopts a query-sensitive partial indexing scheme to further reduce the index size at runtime. Moreover, BIDS is designed as a light-weighted service and can be seamlessly integrated into the current MapReduce runtime as a plug-in of execution engine. The architectural design of BIDS enables it to achieve high scalability by leveraging MapReduce to process index operations in parallel.

Indexing techniques are useful for locating a subset of data that satisfy the search condition quickly without having to scan the whole database. They are indeed the most effective means in reducing query processing cost and many indexes have been proposed for such purposes. However, it is not straightforward to introduce a new indexing structure to an existing system, as it affects not only the storage manager, but also query processor and concurrency controller. The problem is further complicated in distributed processing platforms as data and indexing structures may be distributed. Indexing in distributed processing platforms should have the following features:

1. To support different types of applications and queries, a general indexing framework is required which can be used to build all popular indexes, such as $B^+$-tree index and R-tree index, for the distributed systems. It should also provide unified interfaces for users to implement new types of index.

2. The framework should work as a non-intrusive component for existing systems such as MapReduce so that the previous algorithms written for those systems do not need to be modified to exploit the benefit of index-base processing.

3. As an index service for parallel data processing, the design of index framework must consider the efficiency, reliability and scalability as its first class citizen.

Based on the above rationale, we take our previous research one step further, and propose an indexing framework, **ScalaGiST** – Scalable Generalized Search Tree – which is intrigued by classical Generalized Search Tree (GiST) [45]. Traditional GiST provides functionalities of various types of database search trees in a single package, while *ScalaGiST* is designed for dynamic distributed environments to handle large-scale datasets and adapt to changes in the workload while leveraging commodity hardware. *ScalaGiST* is extensible in terms of both data and query in that it enables users to define indexes for new type of data and provides efficient lookup over the index data as built-in functions without the need of data mapping as being used in other distributed indexing frameworks [24, 70]. Indexes in *ScalaGiST* are distributed and replicated among index servers in the cluster for scalability, data availability and load balancing purposes. *ScalaGiST* develops a light-weight distributed processing service to process index requests in parallel and effectively reduce the overhead of searching over a large index. *ScalaGiST* is designed as an indexing service and can work with other systems in a non-intrusive way. While secondary indexes facilitate a more direct location of data of interest, they may incur non-negligible cost due to random accesses to the base data. Therefore, *ScalaGiST* develops a data access optimizer to compare two possible query execution plans, namely index scan and full table scan, and choose the better plan before running the query.

### 1.3.2 Parallelizing the RDBMSs

The production environment for analytical data management applications is rapidly changing. Many enterprises are shifting away from deploying their analytical databases on high-end proprietary machines, and moving towards cheaper, lower-end, commodity hardware, typically arranged in a shared-nothing MPP (Massively Parallel Processing) architecture, which is widely believed to scale the best [63]. However, there are very few known parallel database deployments consisting of more than one hundred nodes [5]. There are a variety of reasons why parallel databases generally do not scale well into the hundreds of nodes. First, failures become increasingly common as one adds more nodes to a system, yet parallel databases tend to be designed with the assumption that failures are a rare event. Second, parallel databases generally assume a homogeneous array of machines, yet it is nearly impossible to achieve pure homogeneity at scale. Third, until recently, there have only been a handful of applications that required deployment on more than a few dozen of nodes for reasonable performance, so parallel databases have not been tested at larger scales, and unforseen engineering hurdles await.

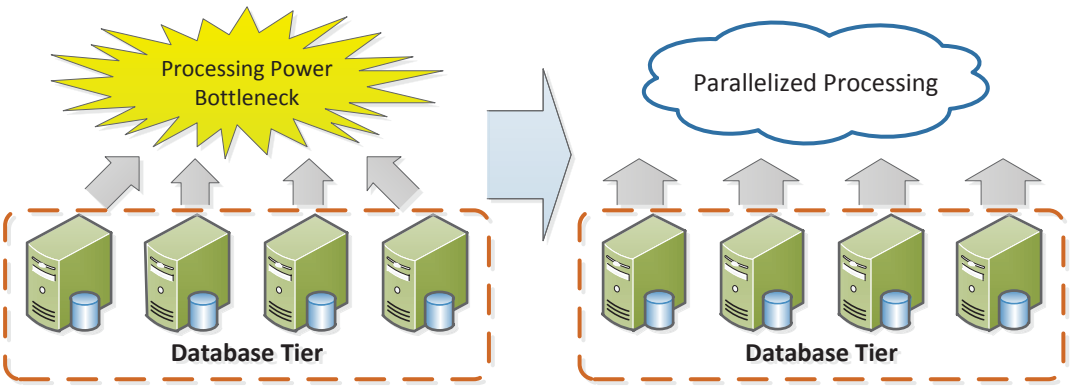



Figure 1.2: Shifting to a Hybrid Architecture

The widespread adoption of MapReduce for MPP systems unfolds discussions and attempts to extend MapReduce to handle data analytical workloads at unconventional scale instead of using parallel databases. Unfortunately, comparing to RDBMS, MapReduce lacks comprehensive query optimizations, and above all, assumes a relatively simplified unstructured data model. Although such design choice preserves the original form of data (e.g., crawled documents, web request logs, etc.) and shortens data-to-query time, it is criticized to place the burden of repeatedly parsing records and cause an order of magnitude slower performance than parallel databases [76].

Ideally, the scalability advantages of MapReduce could be combined with the performance and efficiency advantages of parallel databases to achieve a hybrid architecture that is well suited for large scale systems and can handle the future demands of data intensive application, as illustrated in Figure 1.2. We exploit the feasibility of building a hybrid system that takes the best features from both technologies, and propose BestPeer++ [104], an adaptive query processing engine that incorporates the query execution of traditional parallel databases and MapReduce. In particular, we identify the strategic differences between DBMS query execution and MapReduce, and model the query efficiency for both execution plans. Using the cost model, we devise a hybrid execution engine that adaptively generates the most cost effective plan for queries. The prototype we build approaches parallel databases in performance and efficiency, yet still yields the scalability, fault tolerance, and flexibility of MapReduce-based systems.

## 1.4 Contribution and Impact

This dissertation makes several fundamental contributions towards realizing our vision of building a large scale system that syncretize the efficient access methods of RDBMS

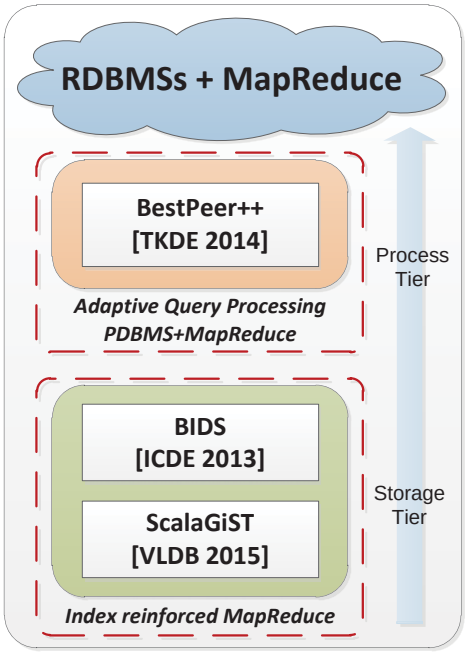



Figure 1.3: Overview of the dissertations contributions classified into the two thrust areas for this dissertation: indexes in MapReduce and adaptive data processing.

and the powerful parallelized processing of MapReduce. Our contributions significantly advance the state-of-the-art by supporting *index* and orchestrating a hybrid processing mechanism for large scale systems. Our technical contributions are in bitmap encoding and processing of large scale data, distributed index support in MapReduce systems, and adaptive query processing incorporating parallel databases and MapReduce. These technologies are critical to ensure the success of the next generation of large scale data processing systems in Cloud Computing infrastructures.

Figure 1.3 summarizes these contributions into the two major thrust areas of this dissertation: indexes in MapReduce and adaptive data processing. We now highlight these contributions and their impact.

- We present a thorough analysis of the state-of-the-art systems and distill the important aspects in the design of different systems and analyze their applicability and scope. We then articulate some basic design principles for designing new MPP systems for the cloud. A thorough understanding and a precise characterization of the design space are essential to carry forward the lessons learned from the rich literature in scalable and distributed database management.

- We design a bitmap-based indexing scheme for large scale distributed data store. Using effective bitmap encoding techniques and partial index mechanism, the in-

dexing scheme is able to achieve high space efficiency. Size is a vital factor for indexing data at large scale, and the compactness of our proposed scheme enables efficient indexing of large scale data.

- We present the architecture and implementation of BIDS [62], a full-fledged indexing and query processing technique based on bitmap. BIDS is one of the first systems to allow seamless integration of index processing in MapReduce runtime. We present the mechanisms for MapReduce-based systems to directly work on the underlying index, and the series of runtime optimizations to facilitate efficient query processing in MapReduce.

- We propose *ScalaGiST*, a generalized index framework to extend the indexibility in MapReduce systems. *ScalaGiST* provides extensibility in terms of data and query types, and hence is able to support unconventional queries in MapReduce system. We define the generalized index interface using which users are able to customized new types of index on their data.

- We present the design and implementation of a index processing mechanism to integrate *ScalaGiST* seamlessly with Hadoop platform, coupled with a cost-based data access optimizer for improving the performance of MapReduce execution. Indexibility in MapReduce systems is decisive in improving query performance, and *ScalaGiST* is the first system providing support to a wide variety of traditional indexes in distributed environment.

- We study the query performance of parallel database systems and MapReduce, and identify the influencing factors with respect to query complexity. We then propose a cost model to evaluate the execution efficiency of a given query when using parallel database and MapReduce. This cost model takes into account data distribution and query parameters, and gives a quantitative guideline for runtime optimization.

- We present BestPeer++ [104], an adaptive query processing mechanism in distributed environment. BestPeer++ is a hybrid system incorporating query processing mechanism from parallel database and MapReduce. Using the proposed cost model, we implement an adaptive query processing mechanism that is able to provide optimal efficiency for different types of query.

- All three techniques have been prototyped in real MapReduce systems to demonstrate feasibility and the benefits of the proposed techniques. A detailed analysis of the trade-offs of each design allows future systems to make informed decisions based on insights from this dissertation.

## 1.5 Organization

In Chapter 2, we provide a systematic survey and analysis of the state-of-the-art in scalable and distributed data management systems, as well as index technologies used in RDBMSs. The rest of the dissertation is organized into two parts focussing on the two thrust areas of this dissertation.

Part I focuses on systems designed to support efficient index in MapReduce systems. Chapter 3 presents our first work on orchestrating bitmap indexing scheme in MapReduce systems. Chapter 4 presents the design of *ScalaGiST*, which provides a generalized index search tree framework for MapReduce.

Part III focuses on models and techniques to enable adaptive large-scale query processing. Chapter 5 presents the technical details of performance modeling of distributed query execution, and the architecture of an adaptive query engine incorporating parallel database and MapReduce.

Chapter 6 concludes this dissertation and outlines some open challenges.

# CHAPTER 2

# State of the Art

*"Stand on the shoulders of giants."*
– Bernard of Chartres and Isaac Newton.

Scalable distributed data management has been the vision of the computer science research community for more than three decades. This chapter surveys the related works in this area in light of the cloud infrastructures and their requirements. Our goal is to distill the key concepts and analyze their applicability and scope. A thorough understanding and a precise characterization of the design space are essential to carry forward the lessons learned from the rich literature in scalable and distributed database management.

## 2.1 Cloud Architectural Service Layers

The past decade has witnessed the emergence of "cloud computing". This paradigm shift entails harnessing large number of (low-end) processors working in parallel to solve a computing problem. While cloud computing has gained fast popularity, users might get overwhelmed with a variety of taxonomy such as cloud platform, platform as a service (PaaS), etc., introduced by various cloud service providers such as Microsoft Azure[1], Google AppEngine[2] and Amazon Web Services[3]. In this section, we review various cloud computing concepts and especially examine its architectural service layers.

One of the beauties of the cloud computing model is the simplicity with which they are presented to the end users. At the same time, the cloud computing model actually con-

[1]http://www.windowsazure.com/
[2]https://appengine.google.com/
[3]http://aws.amazon.com/

sists of a complex series of interconnected layers. Understanding these layers is essential to any organization that wishes to utilize cloud computing services in the most efficient manner. Like the seven-layer OSI model for networking, each layer of the cloud computing model exists conceptually on the foundation of the previous layers. Within this model, there are three different service layers that are used to specify what is being provisioned, Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). Additionally, there are three further layers that are not provided as user services. The Hardware Layer and the Virtualization Layer are owned and operated of the cloud services provider while the Client Layer is supplied by the end users.

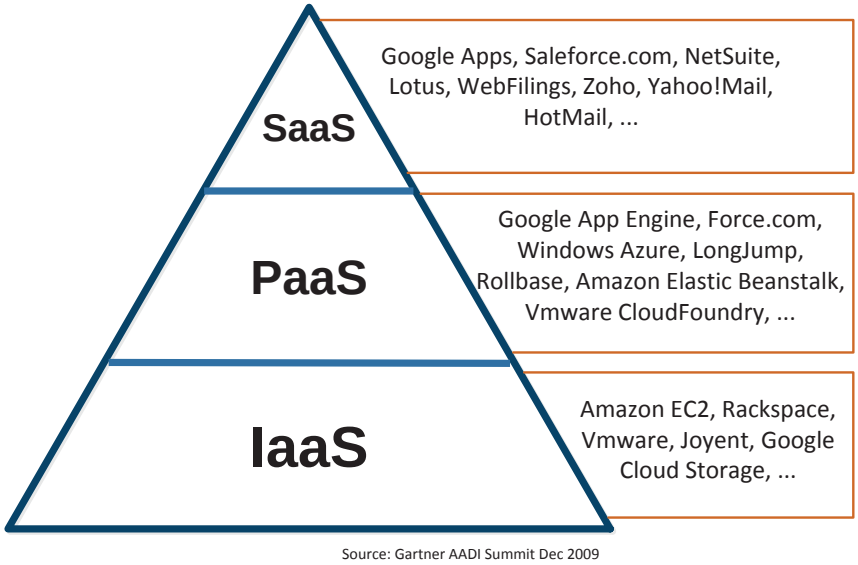



Figure 2.1: Cloud Computing Service Layers

**The Hardware Layer**

The hardware layer is sometimes referred to as the server layer. It represents the physical hardware that provides actual resources that make up the cloud. Since, by definition, cloud computing users do not specify the hardware used to provide services, this is the least important layer of the cloud. Often, hardware resources are inexpensive and are not fault tolerant. Redundancy is achieved simply by utilizing multiple hardware platforms while fault tolerance is provided at other layers so that any hardware failure is not noticed by the users.

**The Virtualization Layer**

Often referred to as the infrastructure layer, the virtualization layer is the result of various operating systems being installed as virtual machines. Much of the scalability and flexibility of the cloud computing model is derived by the inherent ability of virtual machines to be created and deleted at will.

Above these two layers are the service layers where the actual **cloud services** are delivered to users. In Figure 2.1, we can see how the analyst firm Gartner segregates the remaining three layers.

**SaaS**

Starting from the highest level: software applications that are only available online fall into the ”Software-as-a-Service” category, also known as “SaaS”. Services at the software level consist of complete applications that do not require development. Such applications can be email, customer relationship management, and other office productivity applications. Enterprise services can be billed monthly or by usage, while software as service offered directly to consumers, such as email, is often provided for free.

**IaaS**

On the opposite end of the spectrum, we have “Infrastructure-as-a-Service,” or “IaaS,” where hardware is outsourced. The infrastructure layer builds on the virtualization layer by offering the virtual machines as a service to users. Instead of purchasing servers or even hosted services, IaaS customers can create and remove virtual machines and network them together at will. Clients are billed for infrastructure services based on what resources are consumed. This eliminates the need to procure and operate physical servers, data storage systems, or networking resources.

**PaaS**

In the middle, we have “Platform-as-a-Service,” or “PaaS.” The platform layer rests on the infrastructure layer’s virtual machines. At this layer customers do not manage their virtual machines, they merely create applications within an existing API or programming language. There is no need to manage an operating system, let alone the underlying hardware and virtualization layers. Clients merely create their own programs which are hosted by the platform services they are paying for. While this service level is the least known or discussed, some feel that this is the most powerful of the three. Systems like Google AppEngine, Salesforce’s Heroku[4], Microsoft Azure, and VMwares Cloud Foundry[5], all fall under the PaaS umbrella.

## 2.2 Cloud Data Management

### 2.2.1 Early Trends

Early efforts targeting the design space of scalable data management systems resulted in two different types of systems: distributed DBMSs (**DDBMS**) such as R* [60] and

[4]http://www.salesforce.com/heroku/

[5]https://www.gopivotal.com/platform-as-a-service/cloud-foundry

SDD-1 [85] and parallel DBMSs (**PDBMS**) such as Gamma [32] and Grace [40]. DeWitt and Gray [33] and Ozsu and Valduriez [74] provide thorough surveys of the design space, principles, and properties of these systems. The goal of both classes of systems was to distribute data and processing over a set of database servers while providing the abstractions and semantics similar to centralized systems.

Different from the distributed and parallel DBMSs, another approach to scaling DBMSs while preserving the semantics of a single node RDBMS is through **data sharing**. In such a model, a common database storage is shared by multiple processors that concurrently execute transactions on the shared data. Examples of such systems are Oracle Real Application Clusters [20] and IBM DB2 data sharing [54]. A common aspect of all these designs is a shared lock manager responsible for concurrency control. Even though many commercial systems based on this architecture are still used in production, the scalability of such systems is limited by the shared lock manager and the complex recovery mechanisms resulting in longer unavailability periods as a result of a failure.

While conventional distributed and parallel database technologies lay the foundation for cloud-based data management systems, they are not sustainable beyond a few machines due to the crippling effect on performance caused by partial failures and synchronization overhead.

### 2.2.2 Eyes in the Cloud

Historically, data management systems are categorized by two different workloads: online transactional processing (OLTP) and online analytical processing (OLAP). Systems handling OLAP and OLTP workloads have distinctive architectural perspectives: RDBMS for OLTP and data warehousing system for OLAP. Periodically, data in RDBMS are extracted, transformed and loaded (a.k.a. ETL) into the data warehouse. This system-level separation is motivated by the facts that OLAP is computationally expensive and its execution on a separate system will not compete for resources with the response-critical OLTP operations, and snapshot-based results are generally sufficient for decision making.

With the advent of the Cloud paradigm, the two streams of systems both have their projections in the new era, in particular, Key-Value Stores for OLTP, and MapReduce and its derivatives for OLAP.

#### The Key-Value Store

With the growing popularity of the Internet, many applications were delivered over the Internet and the scale of these applications also increased rapidly. As a result, many Internet companies, such as Google, Yahoo!, and Amazon, faced the challenge of serving

hundreds of thousands to millions of concurrent users. Classical RDBMS technologies could not scale to these workloads while using commodity hardware to be cost-effective. The need for low cost scalable DBMSs resulted in the advent of Key-value stores such as Google's Bigtable [21], Yahoo!'s PNUTS [28], and Amazon's Dynamo [31].[6] These systems were designed to scale out to thousands of commodity servers, replicate data across geographically remote data centers, and ensure high availability of user data in the presence of failures which is the norm in such large infrastructures of commodity hardware. These requirements were a higher priority for the designers of the Key-value stores than rich functionality. Key-value stores support a simple key-value based data model and single key access guarantees, which were enough for their initial target applications [96]. In this section, we discuss the design of these three systems and analyze the implications of the various design choices made by these systems.

BigTable [21] was designed to support Google's crawl and indexing infrastructure. A BigTable cluster consists of a set of servers that serve the data; each such server (called a tablet server) is responsible for parts of the tables (known as a tablet). A tablet is logically represented as a key range and physically represented as a set of SSTables. A tablet is the unit of distribution and load balancing. At most one tablet server has read and write access to each tablet. Data from the tables is persistently stored in the Google File System (GFS) [42] which provides the abstraction of scalable, consistent, fault-tolerant storage. There is no replication of user data inside BigTable; all replication is handled by the underlying GFS layer. Coordination and synchronization between the tablet servers and metadata management is handled by a master and a Chubby cluster [16]. Chubby provides the abstraction of a synchronization service via exclusive timed leases. Chubby guarantees fault-tolerance through log-based replication and consistency amongst the replicas is guaranteed through a Paxos protocol [19]. The Paxos protocol [57] guarantees safety in the presence of different types of failures and ensures that the replicas are all consistent even when some replicas fail. But the high consistency comes at a cost: the limited scalability of Chubby due to the high cost of the Paxos protocol. BigTable, therefore, limits interactions with Chubby to only the metadata operations.

PNUTS [28] was designed by Yahoo! with the goal of providing efficient read access to geographically distributed clients. Data organization in PNUTS is also in terms of range-partitions tables. PNUTS performs explicit replication across different data centers. This replication is handled by a guaranteed ordered delivery publish/subscribe system called the Yahoo! Message Broker (YMB). PNUTS uses per record mastering and the master is responsible for processing the updates; the master is the publisher to YMB and

[6] At the time of writing, various other Key-value stores (such as HBase, Cassandra, Voldemort, MongoDB etc.) exist in the open-source domain. However, most of these systems are variants of the three in-house systems.

the replicas are the subscribers. An update is first published to the YMB associated to the record's master. YMB ensures that updates to a record are delivered to the replicas in the order they were executed at the master, thus guaranteeing single object time line consistency. PNUTS allows clients to specify the freshness requirements for reads. A read that does not have freshness constraints can be satisfied from any replica copy. Any read request that requires data that is more up-to-date than that of a local replica must be forwarded to the master.

Dynamo [31] is another highly available and scalable distributed data store built for Amazon's platform. In addition to scalability, high write availability, even in the presence of network partitions, is a key requirement for Amazons shopping cart application. Dynamo therefore explicitly replicates data and a write request can be processed by any of the replicas. It uses a quorum of servers for serving the read and writes. A write request is acknowledged to the client when a quorum of replicas has acknowledged the write. To support high availability, the write quorum size can be set to one. Since updates are propagated asynchronously without any ordering guarantees, Dynamo only supports eventual replica consistency [97] with the possibility that the replicas might diverge. Dynamo relies on application level reconciliation based on vector clocks [56].

The distinguishing feature of the Key-value stores is their simple data model. The primary abstraction is a table of items where each item is a key-value pair or a row. The value can either have structure (as in BigTable and PNUTS), or can be an uninterpreted string or blob (as in Dynamo). BigTables data model is a sparse multi-dimensional sorted map where a single data item is identified by a row identifier, a column family, a column, and a timestamp. The column families are the unit of data co-location at the storage layer. PNUTS provides a more traditional flat row-like structure similar to the relational model. Atomicity and isolation are supported at the granularity of a single key-value pair, i.e., an atomic read-modify-write operation is supported only for individual key-value pairs. Accesses spanning multiple key-value pairs are best-effort without guaranteed atomicity and isolation from concurrent accesses. These systems allow large rows, thus allowing a logical entity to be represented as a single row. Restricting data accesses to a *single-key* provides designers the flexibility of operating at a much finer granularity. Since a single key-value pair is never split across compute nodes, application level data manipulation is restricted to a single compute node boundary and thus obviates the need for multi-node coordination and synchronization [44]. As a result, these systems can scale to billions of key-value pairs using horizontal partitioning. The rationale is that even though there can be potentially millions of requests, the requests are generally distributed throughout the data set. Moreover, the single key operation semantics limits the impact of failure to only the data that was being served by the failed node; the rest of the nodes in

the system can continue to serve requests. Furthermore, single-key operation semantics allows fine-grained partitioning and load-balancing. This is different from RDBMSs that consider data as a cohesive whole and a failure in one component results in overall system unavailability.

### MapReduce in Action

MapReduce [30] and related software such as the open source Hadoop [1], useful extension [72, 93], and Microsoft's Dryad/SCOPE stack [48, 18] are all designed to automate the parallelization of large sale data analysis workloads.

MapReduce is a simplified parallel data processing approach for execution on a computer cluster. Its programming model consists of two user defined functions, *map* and *reduce* 2.1.

| | |
|---|---|
| *map* | $(k1, v1) \rightarrow list(k2, v2)$ |
| *reduce* | $(k2, list(v2)) \rightarrow list(v3)$ |

Table 2.1: *map* and *reduce* Functions

Users specify a *map* function that processing a key/value pair (e.g. filename/file) to generate a set of intermediate key/value pairs, and a *reduce* function that collect and aggregate all intermediate values associated with the same intermediate key. The beauty of MapReduce is that it provides the developers with conveniently programmable interface, while the system is responsible for scheduling and synchronizing the parallel computation. Its wide adoption and success lies in its distinguishing features, which can be summarized as follows.

1. **Flexibility**. Since the code for *map* and *reduce* are written by the user, there is considerable flexibility in specifying the exact processing that is required over the data rather than specifying it using SQL. Programmers can write simple map and reduce functions to process petabytes of data on thousands of machines without the knowledge of how to parallelize the processing of a MapReduce job.

2. **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, and MapReduce can support it seamlessly through data parallel execution. MapReduce was successfully deployed on thousands of nodes and able to handle petabytes of data.

3. **Efficiency**. MapReduce does not need to load data into a database, which typically incurs high cost. It is, therefore, very efficient for applications that require processing the data only once (or only a few times).

4. **Fault Tolerance**. In MapReduce, each job is divided into many small tasks that are assigned to different machines. Failure of a task or a machine is compensated by assigning the task to a machine that is able to handle the load. The input of a job is stored in a distributed file system where multiple replicas are kept to ensure high availability. Thus, the failed map task can be repeated correctly by reloading the replica. The failed reduce task can also be repeated by re-pulling the data from the completed map tasks.

Despite its evident merits, MapReduce often fails to exhibit acceptable performance for various processing tasks. The criticisms of MapReduce center on its reduced functionality, requiring considerable amount of programming effort, and its unsuitability for certain type of applications (e.g. those that requires iterative computations) [34, 76, 89]. MapReduce does not require the existence of a schema and does not provide a high-level language such as SQL. The flexibility advantage mentioned above comes at the expense of considerable (and usually sophisticated) programming on the end of the user. Consequently, a job that can be performed using relatively simple SQL commands may require considerable amount of programming in MapReduce, and this code is generally not reusable. To make MapReduce easier to use, a number of high-level languages have been developed, among which Pig Latin [72] and HiveQL [93] are the two representative practices.

Pig Latin [72] is a dataflow language that adopts a step-by-step specification method where each step refers to a data transformation operation. It supports a nested data model with user defined functions and the ability to operate over plain files without any schema information. The details of these features are discussed below:

1. Dataflow language. Pig Latin is not declarative and the user is expected to specify the order of the MapReduce jobs. Pig Latin offers relational primitives such as LOAD, GENERATE, GROUP, FILTER and JOIN, and users write a dataflow program consisting of these primitives. The order of the MapReduce jobs generated is the same as the user-specified dataflow, which helps users control query execution.

2. Operating over plain files. Pig is designed to execute over plain files directly without any schema information although a schema can also be optionally specified. The users can offer a user-defined parse function to Pig to specify the format of the input data. Similarly, the output format of Pig can also be flexibly specified by the user.

3. Nested data model. Pig Latin supports a nested data model. The basic data type is Atom such as an integer or string. Atoms can be combined into a Tuple, and a several Tuples form a Bag. It also supports more complex data types such as Map⟨sourceIP, Bag(Tuple1, Tuple2, ...)⟩. This model is closer to the recursive data type in object-oriented programming languages and easier to use in user defined functions.

4. User defined functions (UDFs). Due to the nested data model of Pig Latin, UDFs in Pig support non-atomic input parameters, and can output non-atomic values. The UDFs can be used in any context, while in SQL, the set-valued functions cannot be used in the SELECT clause.

HiveQL is a SQL-like declarative language that is part of the Hive [93] system, which is an OLAP execution engine built on top of Hadoop. HiveQL features are the following:

1. SQL-like language. HiveQL is a SQL-like query language that supports most of the traditional SQL operators such as SELECT, CREATE TABLE, UNION, GROUP BY, ORDER BY and JOIN. In addition, Hive has three operators, MAP, CLUSTER BY and REDUCE, which could integrate user defined MapReduce programs into the SQL statement. HiveQL supports equijoin, semijoin and outer join. Since Hive is a data warehouse system, the insert operation in HiveQL does not support in-place insertion into an existing table, instead it replaces the table by the output of a HiveQL statement.

2. Data Model. Hive supports the standard relational data model: data are logically modeled as rows and tables, and a table may consist of several logical partitions, whose purpose is mainly for load balancing. Tables are physically stored as directories in distributed file system (DFS).

Pig Latin and HiveQL supplement MapReduce with a language interface, enhance its programmability and usability. Most importantly, these efforts explore the feasibility of extending the generic MapReduce to serve a better data analytical purpose.

Besides generic MapReduce (and its language layer), there are many other distributed data processing systems that have been inspired by MapReduce but that go beyond the MapReduce framework. These systems have been designed to address various problems, such as iterative processing over the same dataset, that is not well handled by MapReduce, and many are still ongoing.

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 [48] and epiC [52].

Microsoft's Dryad [48] a general-purpose distributed execution engine for coarse-grain data-parallel applications. Dryad 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.

A Dryad job is coordinated by a process called the "job manager". The job manager contains the application-specific code to construct the job's communication graph along with library code to schedule the work across the available resources. The scheduler inside the job manager keeps track of the state and history of each vertex in the graph.

Driven by the limitations of MapReduce-based systems in dealing with "varieties" in cloud data management, epiC [52] 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 Dryads 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) [17]. 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 [24, 98, 101], handles the OLTP queries.

With the previous research paving the way, one of the most recent trends reinforcing MapReduce in data analysis context is the development of efficient full-fledged MapReduce-based RDBMSs. In their simplest form, these systems consist of only a SQL parser, which transforms the SQL queries into a set of MapReduce jobs. Examples include Hive [93] and Google's SQL translator [22]. In a more complete form, a MapReduce-based DBMS natively incorporates existing database technologies to improve performance and usability, such as indexing, data compression, and data partitioning. Examples include HadoopDB [7], Llama [59], and Cheetah [25]. Some of these systems follow the traditional relational DBMS approach of storing data row-wise (e.g., HadoopDB), and are, therefore, called row stores. Others (e.g., Llama) store data column-wise, and are called column stores. It is now generally accepted that column-wise storage model is prefer-

able for analytical applications that involve aggregation queries because (a) the values in each column are stored together and a specific compression scheme can be applied for each column, which makes data compression much more effective, and (b) it speeds up the scanning of the table by avoiding access to the columns that are not involved in the query [90]. In addition to pure row stores and column stores, some systems adopt a hybrid storage format (e.g., Cheetah): the columns of the same row are stored in the same data chunk, but the format of each data chunk is column oriented.

A full DBMS implementation over MapReduce usually supports the following functions: (1) a high level language, (2) storage management, (3) data compression, (4) data partitioning, (5) indexing, and (6) query optimization.

HadoopDB [7] introduces the partitioning and indexing strategies of parallel DBMSs into the MapReduce framework. Its architecture consists of three layers. The top layer extends Hive to transform the queries into MapReduce jobs. The middle layer implements the MapReduce infrastructure and DFS, and deals with caching the intermediate files, shuffling the data between nodes, and fault tolerance. The bottom layer is distributed across a set of computing nodes, each of which runs an instance of PostgreSQL DBMS to store the data.

HadoopDB combines the advantages of both MapReduce and conventional DBMSs. It scales well for large data sets and its performance is not affected by node failures due to the fault tolerance of MapReduce. By adopting the co-partitioning strategy, the join operator can be processed as a map-only job. Moreover, at each node, local query processing automatically exploits the functionality of PostgreSQL.

Llama [59] proposes the use of a columnar file (called CFile) for data storage. The idea is that data are partitioned in vertical groups, each group is sorted based on a selected column and stored in column-wise format in HDFS. This enables selective accesses only to the columns used in a query. In consequences, more efficient access to data than traditional row-wise storage is provided for queries that involve a small number of attributes.

Cheetah [25] also employs data storage in columnar format and also applies different compression techniques for different types of values appropriately. In addition, each cell is further compressed created using GZIP. Cheetah employs the PAX layout [11] at the block level, so each block contains the same set of rows as in row-wise storage, only inside the block column layout is employed. Compared to Llama, the important benefit of Cheetah is that all data that belong to a record are stored in the same block, thus avoiding expensive network access (as in the case of CFile).

The detailed comparison of the three systems is shown in Table 2.2. In systems that support a SQL-like query language, user queries are transformed into a set of MapReduce jobs. These systems adopt different techniques to optimize query performance, and many

| | HadoopDB | Llama | Cheetah |
|---|---|---|---|
| Language | SQL-like | Simple interface | SQL |
| Storage | Row store | Column store | Hybrid store |
| Data Compression | No | Yes | Yes |
| Data Partition | Horizontally partitioned | Vertically partitioned | Horizontally partitioned at chunk level |
| Indexing | Local index in each database instance | Local index + Bitmap Index | Local index for each data chunk |
| Query Optimization | Rule based optimization plus local optimization by PostgreSQL | Column-based optimization, late materialization and processing multiway join in one job | Multi-query optimization, materialized views |

Table 2.2: Comparison of MapReduce DBMS Implementations

of these techniques are adaptations of well-known methods incorporated into many relational DBMSs. The storage scheme of HadoopDB is row-oriented, while Llama is a pure column-oriented system. Cheetah adopts a hybrid storage model where each chunk contains a set of rows that are vertically partitioned. This "first horizontally-partition, then vertically-partition" technique has been adopted by other systems such as RCFile [43]. Both Llama and Cheetah take advantage of superior data compression that is possible with column-storage.

Generic MapReduce is designed for batch processing workloads in which a job scans through the data and generates result in one pass. Although several MapReduce jobs can be concatenated to implement more complex logic, this model is not well suited for a class of emerging data-intensive applications with much more diverse computation models, such as iterative computation [15, 106], graph processing [64, 61], and continuous processing [68, 3]. Detailed introduction and comparison of these system is presented in [58].

### 2.2.3 Design Choices and their Implications

Even though all the above systems share some common goals, they also differ in some fundamental aspects of their designs. We now discuss these differences, the rationale for these decisions, and their implications.

#### Data Model

Most generic MapReduce systems adopt simplified data model – by "simplified" it means that data are directly imported from sources without much effort in parsing or

any form of preprocessing. Therefore, relational **schema** does not fit in MapReduce's context. This design choice was initially made based on two unique characteristics of MapReduce systems: (a) the massiveness and heterogeneity of data do not allow upfront rigorous scheme design, and (b) MapReduce programs mostly use brute force scan instead of selective access to data.

Key-value stores provide primary schema abstraction which can be seen as a table of items where each item is a *key-value* pair or a row. The purpose of this data model is to provide transactional semantics while guaranteeing high scalability and availability.

Be it generic MapReduce or Key-value store, the first class citizen in their design is scalability. The storage components in such systems are distributed file systems, and are decoupled from the upper layer processing engine. Although DFS provide massive storage capability and high scalability, its power in data representation is largely limited.

HadoopDB [7] and many other hybrid systems resort to RDBMS's style of storage by replacing the whole storage layer using RDBMSs. Albeit with stronger data model support, there remains a great deal of engineering efforts in deployment and tuning of the hybrid architecture, and the performance optimization issues are left untouched.

There is a tradeoff between comprehensive data model and scalability. In essence, the origin of the scalability problem is not in that the data model itself do not scale, but the algorithms and techniques built over the data model cannot scale well. Therefore, developing scalable techniques regardless of the underlying data model is the key to success in system design.

### Processing Strategy

Much of the performance issues of MapReduce and its derivative systems can be attributed to the fact that they were not initially designed to be used as complete, end-to-end data analysis systems over structured data. Their target use cases include scanning through a large set of documents produced from a web crawler and producing a web index over them [30]. In these applications, the input data is often unstructured and a brute force scan strategy over all of the data is usually optimal. MapReduce then helps automate the parallelization of the data scanning and application of user defined functions as the data is being scanned.

For more traditional data analysis workloads that work with data produced from business operational data stores, the data is far more structured. Furthermore, the queries tend to access only a subset of this data (e.g. breakdown the profit of stores *located in the Northeast*). Using data structures that help accelerates access to needed entities (such as indexes) and dimensions (such as column-stores), and data structures that precalculate common requests (such as materialized views) often outperform a brute-force scan ex-

ecution strategy. Therefore, it is desirable for MapReduce to support not only the data structures, but also the interface to incorporate the processing of these structures into its framework.

To facilitate data analysis workloads, there are a number of database operators implemented in MapReduce [71, 14, 94, 66, 8]. In addition to "partially" employ database technologies, a hybrid solution that combines the fault tolerance, heterogeneous cluster, and ease of use out-of-the-box capabilities of MapReduce with the efficiency, performance, and tool plugability of shared-nothing parallel database systems could have a significant impact on the cloud data analysis market.

## 2.3 Index Support in the Cloud

In generic MapReduce, data access are resorted to brute-force parallel scan. Moreover, given a set of input data partitions stored on DataNodes, the execution framework of MapReduce will initiate map tasks on all input partitions. However, for certain types of analytical queries, it would suffice to access only a subset of the input data to produce the result. Other types of queries may require focused access to a few tuples only that satisfy some predicate, which cannot be provided without accessing and processing all the input data tuples. In both cases, it is desirable to provide a selective access mechanism to data, in order to prune local non-useful data at a DataNode from processing as well as pruning entire DataNode from processing if necessary. In traditional RDBMSs, this problem is solved by means of *indexing*. There have been considerable amount of efforts incorporating the idea of index in MapReduce systems. Among them, some are more tree-oriented that migrate the traditional tree indexes to MapReduce framework [37, 9, 101, 99], while some others adjust the data layout to make data "recognizable" to MapReduce programs, and thus achieve better intellective data access [35, 36, 38].

In [9], B-tree was made scalable by distributed transactions. The design of the scalable B-tree relies on an underlying distributed data sharing service, Sinfonia [10], which provides fault tolerance and a light-weight distributed atomic primitive. The nodes of a B-tree are spread over multiple servers in a local-area network, and modifications to the B-tree are handled by distributed transactions.

The B-tree operations are implemented as natural extensions of centralized B-tree algorithms wrapped in optimistic transactions. As clients traverses the tree, it retrieves nodes from the servers as needed, and adds those nodes to the transaction's read set. If the client wants to change a node, say due to a key-value insertion or a node split, the client locally buffers the change and adds the changed node to the transaction's write set. To commit a transaction, the client executes a Sinfonia mini-transaction, which (a) validates

that the nodes in the read set are unchanged, by checking that their version numbers match what is stored at the servers, and (b) if so, atomically performs the updates in the write set.

However, this method suffers from two weakness. First, although it uses a B-tree based index, such index is mainly designed for simple lookup queries and is therefore not capable of handling range queries efficiently. To process a range query [*l*, *u*], it must first locate the leaf node responsible for *l*. Then, if *u* is not contained by the same leaf node, the next leaf node has to be retrieved from some other compute server based on the sibling pointer. Such form of retrieval continues until the whole range is covered, which would easily lead to high transaction and memory overhead. Second, it incurs high maintenance cost for the server nodes and huge memory overhead in the client machines, as the client node (user's own PC) lazily replicates all the corresponding internal nodes.

CG-index [101] is a secondary indexing scheme for Cloud storage systems. It is tailored for online queries and maintained in an incremental way. CG-index software consists of two components: a client library which is linked with user application and a set of index servers which store the index. The CG-index servers operate in a shared pool of compute nodes allocated from Cloud and the index server process can reside in the same physical machine with the storage server process.

In CG-index, a local $\mathrm{B}^+$-tree index which only indexes local data is built for each compute node. The compute nodes then are organized as a structured overlay and a portion of the local $\mathrm{B}^+$-tree nodes are published to the overlay for efficient query processing. In this method, an adaptive algorithm is proposed to select the $\mathrm{B}^+$-tree nodes to be published according to query patterns.

However, the above methods are largely tree-based methods which is known to incur high storage cost. Given the large amount of data in the Cloud, it is not storage efficient to build many such indexes.

Hadoop++ [35] is a system that provides indexing functionality for data store in HDFS by means of User-defined Functions (UDFs), i.e., without modifying the Hadoop framework at all. The indexing information (called Trojan Indexes) is injected into logical input splits and serves as a cover index for the data inside the split. Moreover, the index ins created at load time, thus imposing no overhead in query processing. Hadoop++ also supports joins by co-partitioning data and co-locating them at load time. Intuitively, this enables the join to be processed at the map side, rather than at the reduce side (which entails expensive data transfer/shuffling in the network).

HAIL [36] improves the long index creation times of Hadoop++, by exploiting the *n* replicas (typically n=3) maintained by default by Hadoop for fault-tolerance and by building a different clustered index for each replica. At query time, the most suitable

index to the query is selected, and the particular replica of the data is scanned during the map phase. As a result, HAIL improves substantially the performance of MapReduce processing, since the probability of finding a suitable index for efficient data access is increased. In addition, the creation of the indexes occurs during the data upload phase to HDFS (which is I/O bound), by exploiting "unused CPU ticks", thus it does not affect the upload time significantly. HAIL is shown in [36] to improve index creation time and the performance of Hadoop++.

SpatialHadoop [38] is an extension to Hadoop that injects spatial data awareness in the main layer of Hadoop. By implementing basic spatial operations, SpatialHadoop is able to support multi-dimensional queries in Hadoop runtime.

## 2.4 Peer-to-Peer Data Management Technology

To enhance the usability of conventional P2P networks, database community have proposed a series of PDBMS (Peer-to-Peer Database Manage System) by integrating the state-of-art database techniques into the P2P systems. These PDBMS can be classified as the unstructured systems such as PIAZZA [92], Hyperion [84] and PeerDB [69], and the structured systems such as PIER [46].

The work on unstructured PDBMS focus on the problem of mapping heterogeneous schemas among nodes in the systems. PIAZZA introduces two materialized view approaches, namely Local As View (LAV) and Global As View (GAV). PeerDB employs information retrieval technique to match columns of different tables. The main problem of unstructured PDBMS is that there is no guarantee for the data retrieval performance and result quality.

The structured PDBMS can deliver search service with guaranteed performance. The main concern is the possibly high maintenance cost [6]. To address this problem, partial indexing scheme [103] is proposed to reduce the index size. Moreover, adaptive query processing [105] and online aggregation [102] techniques have also been introduced to improve query performance.

The techniques of PDBMS also have been applied widely adopted in cloud systems. In Dynamo [31] and ecStore [95], a similar data dissemination and routing strategy is applied to manage the large-scale data.

### 2.4.1 Overview of the BestPeer++ System

The BestPeer++ [23] system is a P2P database system designed for Cloud deployment for corporate network applications. and is the foundation on which we conduct our

research in adaptive massive parallel processing in Chapter 5. In this section, we first describe the evolution of BestPeer platform from its early stage as an unstructured P2P query processing system to BestPeer++, an elastic data sharing services in the cloud. We then present the design and overall architecture of BestPeer++.

**BestPeer[7] data management platform.** While traditional P2P network has not been designed for enterprise applications, the ultimate goal of BestPeer is to bring the state-of-art database techniques into P2P systems. In its early stage, BestPeer employs unstructured network and information retrieval technique to match columns of different tables automatically [69]. After defining the mapping functions, queries can be sent to different nodes for processing. In its second stage, BestPeer introduces a series of techniques for improving query performance and result quality to enhance its suitability for corporate network applications. In particular, BestPeer provides efficient distributed search services with a balanced tree structured overlay network (BATON [51]) and partial indexing scheme [103] for reducing the index size. Moreover, BestPeer develops adaptive join query processing [105] and distributed online aggregation [102] techniques to provide efficient query processing.

**BestPeer++, a cloud enabled evolution of BestPeer.** Now in the last stage of its evolution, BestPeer++ is enhanced with distributed access control, multiple types of indexes, and pay-as-you-go query processing for delivering elastic data sharing services in the cloud. The software components of BestPeer++ are separated into two parts: *core* and *adapter*. The core contains all the data sharing functionalities and is designed to be platform independent. The adapter contains one abstract adapter which defines the elastic infrastructure service interface and a set of concrete adapter components which implement such an interface through APIs provided by specific cloud service providers (e.g., Amazon). This "two-level" design facilitates portability for the system. With appropriate adapters, BestPeer++ can be ported to any cloud environments (public and private) or even non-cloud environment (e.g., on-premise data center). In what follows, we first present this adapter and then describe the core components.

### Amazon Cloud Adapter

The key idea of BestPeer++ is to use dedicated database servers to store data for each business and organize those database servers through P2P network for data sharing. The Amazon Cloud Adapter provides an elastic hardware infrastructure for BestPeer++ to operate on by using Amazon Cloud services. The infrastructure service that Amazon Cloud Adapter delivers includes launching/terminating dedicated MySQL database servers and monitoring/backup/auto-scaling those servers.

[7] http://www.comp.nus.edu.sg/∼bestpeer/

Amazon EC2 service is employed to provision the database server. Each time a new business joins the BestPeer++ network, a dedicated EC2 virtual server is launched for that business. The newly launched virtual server (called a BestPeer++ instance) runs a dedicated MySQL database software and the BestPeer++ software. The BestPeer++ instance is placed in a separate network security group (i.e., a VPN) to prevent invalid data access. Users can only use BestPeer++ software to submit queries to the network.

Amazon Relational Data Service (RDS) is used to back up and scale each BestPeer++ instance[8]. The whole MySQL database is backed up to Amazon's reliable EBS storage devices in a four-minute window. In order to provide high availability service, BestPeer++ performs asynchronous back-up operation, and there will be no service interrupt during the back-up process. The scaling scheme of BestPeer++ consists of two dimensions: processing and storage, which scale up independently according to user's computation requirement. Initially, each BestPeer++ instance is launched as a `m1.small` EC2 instance (1 virtual core, 1.7 GB memory) with 5GB storage space. With the growth of business demand, user can scale up to a more powerful EC2 instance (e.g., `m1.large` instance which has 4 virtual cores and 7.5 GB memory). In another word, there is no limitation on the resources used.

Finally, the Amazon Cloud Adapter also provides automatic fail-over service. In a BestPeer++ network, a special BestPeer++ instance (called bootstrap peer) monitors the health of all other BestPeer++ instances, by querying the Amazon CloudWatch service. If an instance fails to respond to the bootstrap peer (e.g., crashed), Amazon Cloud Adapter is called to perform fail-over for that instance. The details of fail-over are presented in Section 5.2.1.

### The Core Components

The BestPeer++ core contains all platform-independent logic, including query processing and P2P overlay. It runs on top of the Cloud adapter and consists of two software components: *bootstrap peer* and *normal peer*. A BestPeer++ network can only have a single bootstrap peer instance which is always launched and maintained by the BestPeer++ service provider, and a set of normal peer instances.

The bootstrap peer is the entry point of the whole network. It has several responsibilities. First, the bootstrap peer serves for various administration purposes, including monitoring and managing normal peers and also scheduling various network management events. Second, the bootstrap peer acts as a central repository for meta data of corporate network applications, including shared global schema, participant normal peer list, and

[8]Actually, the server provisioning is also through RDS service which internally calls EC2 service to launch new servers.

role definitions. In addition, BestPeer++ employs the standard PKI encryption scheme to encrypt/decrypt data transmitted between normal peers in order to further increase the security of the system. Thus, the bootstrap peer also acts as a Certificate Authority (CA) center for certifying the identities of normal peers.

Normal peers are the BestPeer++ instances launched by businesses. Each normal peer is owned and managed by an individual business and serves the data retrieval requests issued by the users of the owning business. To meet the high throughput requirement, BestPeer++ does not rely on a centralized server to locate which normal peer hold which tables. Instead, the normal peers are organized as a balanced tree peer-to-peer overlay based on BATON [51]. The query processing is, thus, performed in entirely a distributed manner.

# Part I

# Indexing the Cloud

# CHAPTER 3

# Exploiting Bitmap Index in MapReduce

## 3.1 Motivation

The emergence of cloud computing techniques has brought about several challenges in data management. As an example, the global network traffic was estimated to be 160 PB per second in 2008. Consider a network monitoring system that is built on top of the traffic logs, and a query to discover a specific type of network attack. Clearly, without effective index support, it is very costly, if not impossible, to handle such a request, even with the help of highly parallel processing engines like the MapReduce-based systems [30]. Therefore, the main challenge of current information systems is not about the effective storage of data, but how to retrieve data.

The first step towards addressing this challenge is the development of key-value store in the Cloud, such as BigTable [21], Dynamo [31] and Cassandra [55]. In these systems, data are organized as key-value pairs and partitioned by a certain key order (range, hash, etc.). Given a key, one can locate the corresponding value efficiently. A wide variety of traditional database applications are now built on top of key-value store systems by specifying one attribute as key and bundling the remaining attributes as value. In this way, queries containing the key attribute in their predicates are efficiently facilitated. However, for other query types, the whole data set has to be scanned, incurring excessive cost. One simple extension is to store a tuple using multiple attributes as the keys. In other words, multiple replicas have to be maintained for a tuple, each using a different subset of attributes as the key. This strategy simplifies query processing, but incurs high storage and update overhead.

In fact, in key-value store, an index is built for the key attribute implicitly. To support

efficient search on other attributes, we can intentionally build additional indexes. In recent years, numerous tree-based indexes (e.g., distributed B-tree index [9, 101] and distributed R-tree index [37, 99]) have been proposed for Cloud systems. These indexing schemes offer good query performance, but are not space efficient and costly to maintain. This is because 1) the index size is usually proportional to, sometimes even larger than the data size itself; 2) to support various types of queries, a large number of indexes have to be built on different attributes, incurring more overhead. Due to the fact that data volume is extremely large in the Cloud, building indexes for these data yields unacceptable cost, as a large number of compute nodes have to be purchased to maintain the indexes. Therefore, to provide scalable data retrieval service in the Cloud, we need to re-examine how indexes should be designed.

In this work, we propose BIDS (Bitmap Index for Database Service), a specialized bitmap indexing scheme for a large-scale data store. BIDS is built on top of the underlying DFS, and adopts a set of techniques to make bitmap indexes more scalable. Only one MapReduce job is required to build the indexes for the columns of one table. Compared to tree-based indexes, BIDS can be built/rebuilt very efficiently, thanks to the application of parallel processing.

The most distinguishing feature of BIDS is its compactness. As an example, for a 100G TPC-H [4] dataset, BIDS takes up less than 20MB to index some columns, such as *Shipdate* and *Commitdate*. This high compactness is achieved by two bitmap encoding schemes, WAH encoding [100] and bit-sliced encoding [83]. Moreover, a query-sensitive partial indexing scheme is proposed to further reduce the index size at runtime. The intuition behind this scheme is to index the hot data to handle skewed query patterns. It selectively indexes those rows and columns that are touched most, based on the runtime statistics during query processing.

BIDS can facilitate a wide variety of queries. First, because of its compactness, more attributes can be indexed, and hence more queries (over different attributes) can be supported. Second, many queries can be directly answered by the bitmap indexes. That is, suppose all the requested columns of a query have BIDS index support, instead of retrieving the data from the DFS, we can recover the values from the index directly, without accessing the original data.

## 3.2 System Architecture

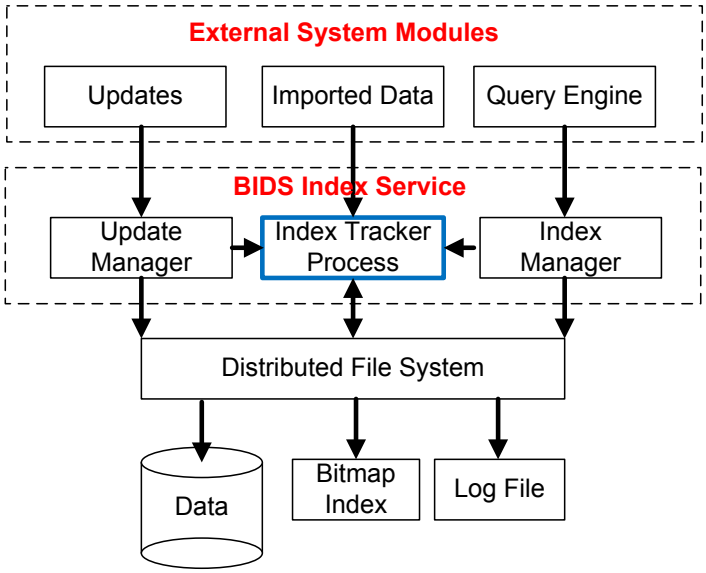



Figure 3.1: BIDS Overview.

In this work, we adopt the loosely connected model of cloud-based systems. Figure 3.1 illustrates how BIDS can be embedded into existing data storage systems. To support BIDS, an index layer is built between the underlying DFS and the external system modules. The BIDS index layer consists of three modules (update manager, index manager and index tracker). As an indexing service, BIDS interacts with external systems, such as Hive [93] and HadoopDB [7]. Our goal is to develop BIDS as a lightweight index module that can be deployed non-intrusively in existing systems.

As shown in Figure 3.1, an ***index manager*** is established in the system to build and maintain the BIDS index. When data are imported into the system, the index manager creates the BIDS index on-the-fly. Essentially, for each column/attribute, the index manager determines if an index should be built, and if so, whether the index should be *partial* or *full*. In a full index, every distinct value of a column is maintained. However, in a partial index, only a subset of the distinct values and/or tuples will be maintained. The subset of distinct values and/or tuples are determined based on the query patterns. We shall discuss this further in Section 3.3.

Both the BIDS index and the imported data are stored in the DFS. The index manager is also responsible for tuning the BIDS index. In particular, for partial indexes, the query patterns provide insights on the distinct values and/or tuples that should be indexed. The index manager will then dynamically adjust the indexes to respond to the queries' needs.

BIDS is defined for read-mostly applications. To handle infrequent updates, BIDS employs an ***update manager*** to buffer updates in memory. Occasionally, the update manager

interacts with the index manager to update the index in batch mode.

When receiving a query, the query engine interacts with the index manager to process the queries. Based on the query type, the query engine may choose different processing strategies. In our implemention, MapReduce is used as the default processing engine.

In our system, the BIDS index is distributed across nodes in the cluster. BIDS runs as an index tracker process in the cluster node, which is responsible for caching index data and performing index lookup operations. Like existing Cloud-based systems, the index tracker process, the DFS process and the MapReduce process run on the same set of nodes. The index service is independent of other services, e.g. storage service and processing service. This design allows the index service to scale up and down without affecting the other services deployed in the same Cloud. We refer to a node that runs an index tracker process an ***index node***.

## 3.3 Methodology

In this section, we present BIDS, a Bitmap Indexing scheme for Database Service in the Cloud. We shall look at how to build and maintain the BIDS index, and how it facilitates query processing. Before delving into the details of BIDS, we first give a brief review of the bitmap index.

### 3.3.1 Bitmap Index

Consider a table $T$ with $n$ records, and a column $c$ with $k$ distinct values. A bitmap index for column $c$ consists of $k$ bit arrays of size $n$ (in number of bits), each of which corresponds to a distinct value of $c$. For the bit array of value $d$, the $i$th bit is set to "1" if the $i$th record has value $d$ for column $c$; otherwise, the $i$th bit is set to "0". Table 3.1 illustrates a bitmap index built for column $returnflag$ of table $Lineitem$ in the TPC-H benchmark. $returnflag$ has three unique values, $n$, $r$ and $a$, resulting in three arrays: $n$:11000100, $r$:00101001 and $a$: 00010010.

Compared to other indexes, bitmap index has significant space and performance advantage. However, it is not suitable for columns with too many distinct values. One way to handle this limitation is to apply the WAH [100] and bit-sliced [83] encodings.

| Row ID | n | r | a |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 |
| 2 | 0 | 1 | 0 |
| 3 | 0 | 0 | 1 |
| 4 | 0 | 1 | 0 |
| 5 | 1 | 0 | 0 |
| 6 | 0 | 0 | 1 |
| 7 | 0 | 1 | 0 |

Table 3.1: Bitmap for Column $returnflag$ of $Lineitem$

#### WAH Encoding

Consider a table with $n$ records, and a column with $k$ distinct values. The size of the bitmap index for the column is $nk$ bits. If a column has too many distinct values, the bitmap index may end up being larger than the original dataset. To address this problem, WAH (Word Aligned Hybrid) [100] encoding can be employed to further compact the bitmap index.

WAH encoding is based on the observation that there are typically many consecutive 0s and 1s in a bitmap, and these consecutive sequences can be "compressed". In WAH, if

each word contains W bits, the data to be encoded is split into segments of W-1 bits, and the encoding scheme is applied to each segment as follows. The first bit of each encoded word is the signal bit and the remaining W-1 bits are used for the encoded data. If the next segment to be encoded contains a mixture of 0s and 1s, WAH leaves it untouched. The signal bit is set to 0, and the original form of the segment is stored in the remaining W-1 bits. If the next $m$ segments are all 0s or 1s, WAH sets the signal bit to 1 and the second bit to 0 or 1 to indicate whether the encoded bits are 0s or 1s. The remaining W-2 bits are now used to represent the number of consecutive 0 or 1 segments. By applying WAH encoding, the size of a bitmap index can be significantly reduced.

| 256 bit value | 1*1, 20*0, 3*1, 203*0, 29*1 | | | |
|---|---|---|---|---|
| 31-bit groups | 1,20*0,3*1,7*0 | 186*0 | 10*0, 21*1 | 8*1 |
| binary value(hex) | 4000380 | 00...000 | 001FFFFF | 000000FF |
| WAH(hex) | 4000380 | 80000006 | 001FFFFF | 000000FF |

Figure 3.2: Example of WAH Encoding.

Figure 3.2 shows an example of WAH encoding, where W=32. We encode a 256-bit value by using WAH encoding. The first 31 bits have both 0s and 1s. Thus, the WAH encoding is equal to the binary value. Then, we encounter six words of consecutive 0s (186*0), which is encoded as "80000006". The next 31 bits just copy their binary values, as we have both 0s and 1s. The last 8 bits are also equal to their binary values, as it is the last word. Thus, in this example, we see a 50% savings in space (a 256-bit value being encoded as 4 32-bit words).

### Bit-Sliced Encoding

It is commonly believed that the bitmap index is only beneficial for columns with few distinct values. One solution to this problem is to apply bit-sliced encoding [83] for the numeric values. In this work, we use 2 as the base in bit-sliced encoding. As such, a value is transformed into its binary representation[1]. To index a column with $k$ distinct values, only $\lceil \log_2 k \rceil$ bit arrays are needed, one for each digit in the binary representation. As an example, column $quantity$ in $Lineitem$ ranges from 1 to 50. Instead of creating a bitmap index with 50 bit arrays, we construct only $\lceil \log_2 50 \rceil = 6$ bit arrays, $b_1, ..., b_6$. If the $quantity$ value of the $i$th record is 6 (000110), the $i$th bit of bit arrays $b_4$ and $b_5$ is set to 1. In essence, bit-sliced encoding trades query performance for storage cost. To reconstruct the bitmaps for a specific value, all $\lceil \log_2 k \rceil$ bit arrays must be retrieved and

[1]Note that bit-sliced encoding is more general, and arbitrary base could have been used, in which case, the number of bits required to represent a value might be different from that for base 2.

combined together. In our system, bit-sliced encoding is blended with WAH encoding to compress the bitmap index, i.e., we apply WAH encoding on each bit array obtained from bit-sliced encoding.

### 3.3.2 Index Creation

A BIDS index is essentially a bitmap index enhanced with techniques tailored for the Cloud system. In our implementation, we build a BIDS index for each table. When data are imported into DFS, the index manager generates statistics on the data distribution. Specifically, given a table $T$ and its column $T.c$, we determine the number of distinct values of $T.c$ (denoted as $f(T.c)$) and the number of bytes for storing a $T.c$'s value (denoted as $g(T.c)$)[2]. Based on the statistics collected, the indexing process examines each column of the table T to determine if an index should be built on the column. The intuition of BIDS is to improve query performance by scanning the bitmap index, which is expected to be much smaller than the original dataset. An adaptive indexing strategy is proposed to optimize BIDS' construction, which is based on the following observations.

**Lemma 3.1.** *Given a column $T.c$, if $g(T.c) \geq \frac{f(T.c)}{8}$, building bitmap index for $T.c$ is always cost effective.*

*Proof.* In this case, the bitmap index for column $T.c$ is no larger than the actual storage required for the column. Therefore, in query processing, we can always scan $T.c$'s bitmap index to get a better performance. □

**Lemma 3.2.** *Given a column $T.c$, the bit-sliced encoding bitmap is beneficial, i.f.f. $\frac{3\log_2 f(T.c)}{8} < g(T.c)$.*

*Proof.* When using bit-sliced encoding to process queries (for details, please refer to Section 3.3.3), we need to scan the bit arrays of all values. Namely, $\log_2 f(T.c)$ bit arrays are scanned. For each bit array, we need to test against the query and write the partial result (also a bit array with the same length) back to the file system. To get the final result, $\log_2 f(T.c)$ partial results are read out and combined. This process incurs $\frac{3\log_2 f(T.c)}{8}$ costs, while scanning the original dataset incurs $g(T.c)$ costs. Therefore, the bit-sliced encoding bitmap is beneficial only if the above condition holds. □

For a column $T.c$, the adaptive indexing scheme is shown in Algorithm 1. We will stick to the basic bitmap index whenever Lemma 3.1 is satisfied (***Strategy S1***), because compared to the bit-sliced encoding bitmap, the basic bitmap index is more efficient for

[2]In this thesis, we assume fixed size columns. For variable size columns, $g(T.c)$ will be equal to the maximum byte size among all values for $T.c$.

query processing. Otherwise, the bit-sliced encoding is used to reduce the size of index (***Strategy S2***). However, if the bit-sliced encoding incurs higher overhead for query processing, we will switch to the partial index. For the non-key attribute, we will initialize the partial index (***Strategy S3***), which is dynamically built during query time. And for the key attribute, bitmap index is not scalable and we will employ conventional indexes like B-trees (***Strategy S4***).

**Algorithm 1:** Adaptive Index(Table $T$, Column $c$)

```
1:  if g(T.c) ≥ f(T.c)/8 then
2:      build index with WAH encoding (Strategy S1)
3:  else
4:      if 3 log2 f(T.c)/8 < g(T.c) then
5:          build index with WAH and bit-sliced encoding (Strategy S2)
6:      else
7:          if T.c is not the primary key then
8:              initial the partial index (Strategy S3)
9:          else
10:             no bitmap index is built (Strategy S4)
```

Suppose *g(Lineitem)* = 150 (namely, the average size of tuples in table Lineitem is 150 bytes ). Table 3.2 shows the indexing strategies for table $Lineitem$.

| Column | Unique Value | Index Strategy |
|---|---|---|
| l_orderkey | depending on data size | S4 |
| l_partkey | depending on data size | S4 |
| l_suppkey | depending on data size | S4 |
| l_linenumber | 7 | S1 |
| l_quantity | 50 | S2 |
| l_extendedPrice | ∼1 million | S3 |
| l_discount | 11 | S1 |
| l_tax | 9 | S1 |
| l_returnflag | 3 | S1 |
| l_linestatus | 2 | S1 |
| *l_date.day | 31 | S1 |
| *l_date.month | 12 | S1 |
| *l_date.year | 7 | S1 |
| l_shipInstruct | 4 | S1 |
| l_shipMode | 7 | S1 |
| l_comments | depending on data size | S4 |

Table 3.2: Indexing Strategy for $Lineitem$

BIDS index is not applicable to the key-attributes ($Orderkey$, $Partkey$, $Suppkey$), as they have unique values for each record. For these attributes, B-tree based index is

more appropriate. Another exception is the $Comments$ column, which also has excessive strings to index. For the rest of the columns in $Lineitem$, we can build BIDS index using different strategies. Note that we split the columns of *date* type into three new columns, which represent *day*, *month* and *year*, respectively. In this way, all the dates can be indexed with strategy S1. Otherwise, we will have 7×356=2492 (TPC-H records 7 years' transactions) possible dates and bit-sliced encoding is required, which may affect the query performance.

To construct BIDS index for a table $T$, the indexing process generates two MapReduce jobs. Suppose the BIDS index is built for column set $\mathcal{C}$ in table $T$. The first MapReduce job applies a sorting function $\mathcal{S}$ to reorder the table records. This is because the compression rate of WAH encoding is highly correlated to the order of records. Sorting leads to a more compact bitmap index [77]. Various types of sorting functions can be applied. In [39], LSH (Locality Sensitive Hashing) is applied to group similar records together, while [77] uses Gray code reorder algorithm to sort data. We just adopt a general sorting function, as finding optimal sorting function is beyond the scope of this paper in our implementation. In particular, we will sort the table by the order of $(c_1, c_2, ..., c_k)$, where $c_i \in \mathcal{C}$, $\mathcal{C} = \bigcup\{c_i\}$ and $f(T.c_i) > f(T.c_{i+1})$. Namely, we first sort the table by the columns with more distinct values. This is because even without WAH encoding, a column with fewer unique values results in a compact bitmap index. The sorting is implemented by modifying the Tera-sort code distributed with Hadoop. We use the values of the sorting columns as the composite keys. Only one MapReduce job is required, regardless of the number of columns involved.

In the sorted data set, each record is represented by an equal size binary array. Thus, to retrieve a record with a specific row ID, we can compute the offset of the record in DFS and invoke the positional read interface. The sorting can be considered as a preprocessing for the BIDS index. It is only invoked when we first build the index. In our cluster, the 100G TPC-H data set can be sorted within 10 minutes.

The second MapReduce job is used to generate the BIDS index for columns. Algorithms 2 and 3 show the pseudo code of the MapReduce job for building BIDS index. To simplify the presentation, we did not show the details of WAH and bit-sliced encoding in the algorithmic descriptions. In the initial step, we will not build the partial BIDS index, which is actually created during query time.

In Algorithm 2, each map process scans a data chunk (512M by default) to build the BIDS index. The finished BIDS index is buffered in memory. As WAH encoding is used and the BIDS index is only created for a data chunk, the map process will not overflow the memory. When the map process detects the end of the stream, it disseminates the BIDS index by generating composite keys. The composite key includes the name of the column

and the value that the bitmap is created for.

**Algorithm 2:** BIDS_Map(Object $key$, Text $value$, Context $context$, BIDSSet $V$)

```
1: Tuple tuple = parse(value)
2: for ∀c_i ∈ C do
3:     BIDS idx = V.get(c_i)
4:     idx.setBitValue(tuple.c_i)
5: if last tuple in the data chunk then
6:     for ∀idx ∈ V do
7:         CompositeKey key =
               new CompositeKey(idx.column, idx.value)
8:         context.collect(key, idx)
```

In Algorithm 3, the reduce process collects the BIDS index created by the different map processes. Next, it applies bit-wise OR to generate the BIDS index for the whole dataset. The resultant BIDS index is then flushed back to DFS using the specific namespace.

**Algorithm 3:** BIDS_Reduce(Key $key$, Iterable $values$, Context $context$)

```
1: String column = key.first()
2: Object value = key.second()
3: BIDS idx = new BIDS()
4: for BitmapWritable val : values do
5:     idx = idx OR val
6: write idx to DFS file "tableName/column/value"
```

Table 3.3 shows the sizes of B-tree (Berkeley DB implementation) and bitmap indexes for different attributes of table $Lineitem$ with 6 million tuples. In the table, the column ***Unsorted BIDS*** refers to the sizes of bitmap indexes obtained without sorting the table, and the column ***BIDS*** refers to the sizes of bitmap indexes under BIDS where the table is sorted by the order of (shipdate, commitdate, receiptdate, quantity, discount, taxt, linestatus, returnflag). From the table, it is clear that BIDS is more space efficient and sorting can reduce the index size significantly.

| Attribute Name | B-tree | Unsorted BIDS | BIDS |
|---|---|---|---|
| shipdate | 601 MB | 35.5 MB | 82 KB |
| commitdate | 589 MB | 35.5 MB | 7.4 MB |
| receiptdate | 585 MB | 35.5 MB | 22.5 MB |
| quantity | 561 MB | 33.2 MB | 33 MB |
| discount | 536 MB | 6.46 MB | 7.9 MB |
| tax | 543 MB | 5.75 MB | 6.44 MB |
| linestatus | 519 MB | 2.15 MB | 5 KB |
| returnflag | 505 MB | 1.43 MB | 740 KB |

Table 3.3: Index Sizes For Six Million Tuples

| Row ID | 0.01 | 0.02 | 0.03 | ... |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | ... |
| 1 | 1 | 0 | 0 | ... |
| 2 | 1 | 0 | 0 | ... |
| 3 | 0 | 0 | 0 | ... |
| 4 | 0 | 0 | 1 | ... |
| 5 | 0 | 0 | 0 | ... |
| 6 | 0 | 0 | 0 | ... |
| 7 | 0 | 1 | 0 | ... |

Table 3.4: Bitmap for Column $l\_discount$ of $Lineitem$

### 3.3.3 Query Processing

BIDS can interact with query engine of external systems to speed up the query processing. It provides an index-based query optimizer. In this Section, we present our index processing strategy with BIDS.

Consider a typical database query,

SELECT $\mathcal{A}$ FROM $T$ WHERE $\mathcal{P}$

where $\mathcal{A}$ denotes an aggregate function over columns of $T$, and $\mathcal{P}$ denotes a set of predicates in the `where` clause. We use $\pi(\mathcal{A})$ and $\pi(\mathcal{P})$ to denote the columns involved in the aggregate function and where clause, respectively.

We note that the BIDS index has different roles in processing the aggregate function and the where clause. For $\mathcal{A}$, the bitmaps for the columns in $\pi(\mathcal{A})$ are used to compute the aggregate result; for $\mathcal{P}$, the bitmaps for the columns in $\pi(\mathcal{P})$ are used to prune the data. For example, let $\mathcal{A} = \{avg(discount)\}$ and $\mathcal{P} = \{returnflag = r\}$. Suppose we have the BIDS index as shown in Table 3.4. By checking the BIDS index for $returnflag$, we have a bitmap (00101001), denoting rows satisfying the predicate, $\mathcal{P}$. Applying this bitmap to BIDS index of $discount$, we can retrieve the discount values for rows 2, 4 and 7. The average discount value can then be computed as $0.02 = \frac{0.01+0.03+0.02}{3}$.

Formally, given a column $c$, let $idx_c$ be the BIDS index for $c$. $idx_c$ composes of a set of bit arrays $(b_1, b_2, ..., b_k)$. Without bit-sliced encoding, one bit array is built for a specific value of the column. Otherwise, it refers to a digit (0 or 1) of the binary encoding of the value. In our system, the index manager maintains a mapping table for the BIDS index and its values. For simplicity, we define function $m(b_i)$ to map between bit array $b_i$ and its value $v_i$. For example, in Table 3.4, $m(b_1)$ returns 0.01. When bit-sliced encoding is used, $m(b_i)$ returns $i$, the position of the bit array in the bitmap. BIDS provides the following operators to manipulate the indexes.

- ***Filter (F):*** If $c \in \pi(\mathcal{P})$ and the predicate for column $c$ is $L < c < U$, we will compute a filtering bit array $f_c$ for column $c$, denoting the rows that satisfy the

predicate. If bit-sliced encoding is not applied,

$$f_c = \bigvee_{L<m(b_i)<U} b_i$$

where $\bigvee$ denotes the bit-wise OR operation. If bit-sliced encoding is used, to generate $f_c$, we need to combine and scan all the bit arrays. Specifically, the $j$th bit of $f_c$ is computed as

$$f_c[j] = \begin{cases} 1 & \text{let } v_j = b_1[j]b_2[j]...b_k[j], \text{ if } L < v_j < U \\ 0 & \text{otherwise} \end{cases}$$

where $v_j$ is a binary value, generated by concatenating all the $j$th values of the bit arrays. Finally, the complete filtering array $f$ is computed by combining the filtering bit arrays of all columns in $\pi(\mathcal{P})$[3].

$$f = \bigwedge_{c\in\pi(\mathcal{P})} f_c$$

where $\bigwedge$ denotes the bit-wise AND operation. $f$ is used to prune the search space during query processing.

- ***Grouping (G):*** If the result is required to be grouped by column $c$, we will split its filter array based on $c$'s bit arrays. Specifically, $k$ filter arrays ($k$ is the number of groups) are created and the $i$th array is generated as

$$f_i[j] = \begin{cases} f[j] & \text{if } b_i[j] = 1 \\ 0 & \text{otherwise} \end{cases}$$

  The filter array $(f_1, f_2, ..., f_k)$ is then passed to the *Materialization* operator.

- ***Materialization (M):*** The ***Materialization*** operator will compute a result for each group. If $c \in \pi(\mathcal{A})$ and the computation is represented as $exp(c)$, we will retrieve $c$'s BIDS arrays $(b_1, b_2, ..., b_k)$ to calculate $exp(c)$. For the $j$th tuple, if $f_x[j] = 1$, it passes the filtering bit array of the $x$th group. We will compute its result for the $x$th group as $exp(c) = exp(m(b_i))$, if $b_i[j] = 1$ and no bit-sliced encoding is used. Otherwise, let $v_j$ be the binary value of the bitmap, namely $v_j = b_1[j]b_2[j]...b_k[j]$. In the bit-sliced encoding case, $exp(c) = exp(v_j)$.

The external query engine can directly apply the above operators to process queries

[3]For simplicity, we assume conjunctive predicates only.

by translating the query into a set of $F$, $G$ and $M$ operators. As an example, given the following query:

$Q_0$:

```
SELECT sum(l_extendedprice * l_discount)
       as revenue
FROM Lineitem
WHERE l_shipdate = Date(1994-01-01) AND
      l_discount < 0.02
GROUP BY l_returnflag
```

A possible plan is:

1. $f = F(l\_shipdate =' 1994-01-01') \bigwedge$ $F(l\_discount < 0.02))$
2. FilterArraySet $S_f = G(f, l\_returnflag)$
3. $\forall f_i \in S_f$, $result_i$=*M(l_extendedprice, $f_i$) op M(l_discount, $f_i$)*

The query engine first applies the $Filter$ operator to the $l\_shipdate$ and $l\_discount$ columns. The result filter array is then forwarded to the $Grouping$ operator, where three new filter arrays are generated as $l\_returnflag$ has three unique values. The filter arrays are passed to the $Materialization$ operator, which selectively extracts data of $l\_extendedprice$ and $l\_discount$ columns. The final result is computed via the materialized values, where $op$ denotes the multiplication operator. The above plan is broadcast to all index tracker processes, which perform the index lookup in parallel. Details of index distribution will be presented in the next section.

Alternatively, the external query engine can use BIDS' interfaces in its MapReduce jobs.

1. In the *map* phase, all the corresponding BIDS indexes are loaded. As maintained by the DFS, the BIDS index data are also partitioned into chunks. Each *mapper* is responsible for reading one specific chunk.
2. *Filter* operator is performed in the *map* phase, where a filtering array is generated for each column in the predicate.
3. The filter array and the BIDS index data for *Materialization* are shuffled to the *reducers*, where the *Grouping* operator is applied to generate the shuffling keys.

4. The filter arrays of different columns are combined in the *reducers* and the *Materialization* operator is applied to compute the aggregation result.

### 3.3.4 Partial Index

As noted in our discussions, the bitmap index is typically not suitable of indexing a column with a large number of distinct values. However, in practice, many queries access only a small portion of the data. For example, in data warehouse system, most queries focus on the (more) recent data rather than the historical data. One effective way to reduce the storage cost of BIDS index is to create indexes for the popular data only. For example, if 80% of the queries are directed at 20% of the data, by indexing the popular data only, the index size can be significantly reduced; while at the same time, we can still answer most queries via the index. Partial indexing has been applied in conventional DBMS [88] to reduce the index cost. In BIDS, we apply this technique to make more columns indexable.

In this work, there are two flavors of partial BIDS index. The first is applicable to conventional bitmap index (as used in our strategy S1). The other is applicable to bitmaps with bit-sliced encoding (as used in our strategies S2 and S3). We shall refer to the former one as partial bitmap index, and the latter as partial bit-sliced bitmap index.

#### Partial Bitmap Index

Consider a table $T$ with a column $c$ for which we would like to build a partial bitmap index. WLOG, suppose column $c$ has $d$ distinct values. The partial index of column $c$ composes of three parts, an indicator bit array $I_c$, an index range $r_c$ and the data bitmap $idx_c = \{b_1, b_2, ..., b_k\}$ ($k \leq d$). $I_c$ and $r_c$ denote the rows and columns that we have indexed, while $idx_c$ is the (partial) BIDS index for the corresponding rows. For example, for the full BIDS index in Table 3.5, suppose $I_c = 10001000$ (i.e., we only indexed records 0 and 4) and $r_c = \{n, r\}$ (i.e., we only indexed values $n$ and $r$), $idx_{returnflag}$ is transformed into Table 3.5.

Compared to the full index bitmap (Table 3.1), the uncompressed version of the partial index requires 66.7% of the storage of the full index. The savings is expected to be more significant when we apply WAH encoding. This is because there are more opportunities for compression as the bit vector of a distinct value has more "0" bits (than its counterpart in a full index).

The partial bitmap index is built as a by-product of query processing. Therefore, we discuss the query processing and tuning strategy together. We follow the discussion and notations in Section 3.3.3. Note that the partial indexing strategy is not applied to the

| Row ID | n | r |
|---|---|---|
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 2 | 0 | 0 |
| 3 | 0 | 0 |
| 4 | 0 | 1 |
| 5 | 0 | 0 |
| 6 | 0 | 0 |
| 7 | 0 | 0 |

Table 3.5: Partial Index for Column $l\_returnflag$ of $Lineitem$

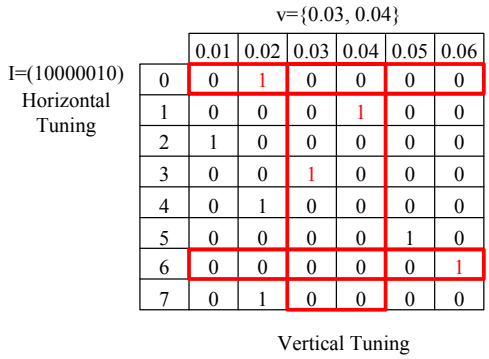
*Group* operator, as most queries require the results for all groups. For a partial index $idx_c$[4],

- ***Filter (F):*** If $c \in \pi(\mathcal{P})$, we generate a filtering bit array $f_c$ for $idx_c$ as in the full index case. Now, since records that are not indexed by $idx_c$ may also satisfy $\mathcal{P}$ (and may be answers to the query), we need to examine all records that are not indexed. To consider the records that are not indexed in $idx_c$, $f_c$ is updated as $f_c \bigvee \bar{I}_c$, where we use $\bigvee$ to denote bit-wise OR and $\bar{I}_c$ is the complement of $I_c$ and is computed as $\bar{I}_c = e$ XOR $I_c$ ($e$ is a bit array with all elements set to 1). The new $f_c$ is then combined with other filtering arrays and used in pruning the dataset. The index manager keeps a query histogram. If it detects that $c$ is always queried in a popular range $r$ which is currently not being indexed, a MapReduce job will be triggered to scan the dataset and materialize the bit arrays in $r$. $r$ is then inserted into the index range $r_c$. Similarly, if a range $r$ has not been used for a certain period of time, it may be removed from $r_c$. We call this type of index tuning *Vertical Tuning*.

- ***Materialization:*** If $c \in \pi(\mathcal{A})$, we first generate a bit array for the tuples that can be processed by the index. The bit array is computed as $x = f \bigwedge I_c$, where $f$ is the filtering array created from the where clause. If $x = f$, then the result can be directly computed via the index. Otherwise, we have $y = x$ XOR $f$. $y$ indicates the tuples that we need to retrieve from DFS on-the-fly. To process the query, we can access DFS to recover the missing tuples or fully scan the dataset if too many random accesses are required. In either case, we get the search result for column $c$. We can update the partial index by setting the bits for tuples in $y$ and changing the indicator as $I_c = I_c \bigvee y$. This type of index tuning is called *Horizontal Tuning*.

*Horizontal Tuning* incurs less overhead than *Vertical Tuning*, as it just reuses the results of queries. Figure 3.3 shows an example of the two types of index tuning. In *Hori-*

[4]Here, we assume a partial index already exists. If the bitmap for a distinct value does not exist, it can be initialized to a bit vector of all 0s.




| | 0.01 | 0.02 | 0.03 | 0.04 | 0.05 | 0.06 |
|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 2 | 1 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 1 | 0 | 0 | 0 |
| 4 | 0 | 1 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 1 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 1 |
| 7 | 0 | 1 | 0 | 0 | 0 | 0 |

Figure 3.3: Example of Partial Index.

*zontal Tuning*, tuples 0 and 6 are indexed and thus $I = (10000010)$. In *Vertical Tuning*, the tuples in the range $[0.03, 0.04]$ are indexed.

When more bits are set in the partial index, WAH encoding cannot provide a good compression performance. Consequently, the index is costly to maintain. Thus, occasionally, we rebuild the partial index by clearing all set bits in $I_c$ and $idx_c$ and resetting $r_c$. When queries follow a skewed distribution, the index reconstruction process only affects a few queries, as the popular tuples will be indexed soon. The details of index maintenance are discussed in the next section.

### Partial Bit-Sliced Bitmap Index

Besides the partial bitmap index, there is another type of partial index, the partial bit-sliced bitmap index. When bit-sliced encoding is used, the resultant partial index is similar to that of the partial bitmap index. However, none of the bitmaps can be dropped, and so $r_c$ is essentially not needed. In this case, though the storage for the uncompressed bitmaps is the same as that required for the full index, it is still more space efficient as WAH encoding can result in a more compact representation because of the larger number of "0" bits.

Let $idx_c = \{b_1, b_2, ..., b_k\}$. If $c = v$ appears in the search predicate $\mathcal{P}$, we transform $v$ into its binary representation v[1]v[2]...v[k]. Let the indicator be $I_c$. For tuple $t_i$ in $I_c$, if $b_j[i] = v[j]$ for all $1 \leq j \leq k$, $t_i$ passes the filter and thus the filter bitmap $f_c$ sets its $i$th bit to 1.

If $c$ appears in $\mathcal{A}$ and the filter bitmap is $f$, we first check whether $f \bigwedge I_c$ equals to $f$. If it does, then all results can be obtained from the index. For the $i$th tuple, we just transform $b_1[i]b_2[i]...b_k[i]$ back to its represented value and compute the query result. Otherwise, we need to retrieve the tuple from DFS.

For the partial bit-sliced bitmap index, we only need to consider horizontal tuning. If $c \in \pi(\mathcal{A})$ and we have not built full index for $c$, to process the query, we need to retrieve

data from DFS. Suppose bitmap $y$ indicates the missing tuples from the partial index. After processing the query, we decide to build partial index for tuples in $y$ by exploiting the query result. Let the partial index of $c$ be $\{b_1, b_2, ..., b_k\}$. $b_j$ is for the $j$th bit in the bit-sliced encoding. For tuple $t_i$ in $y$, we transform $t_i.c$ to a bit-sliced encoding string $s$ (in our case, $s$ is a binary string). We set $b_j[i]$ to $s[j]$ for $1 \leq j \leq k$. Besides the indicator is updated as $I_c = I_c \bigvee y$.

### 3.3.5 Discussion for Join Processing

Our BIDS index cannot be used to index the primary key as the number of unique values is equal to the number of tuples. Thus, directly applying the BIDS index to process primary-foreign key join is not efficient. However, it can be integrated with other index structures to handle the join processing. One possibility is to build the Trojan index [35], which organizes joinable tuples in a co-partition.

Alternatively, the BIDS index can be used to prune the participating tables before the join is performed. Consider the following query:

SELECT $\mathcal{A}$ FROM $T_1$, $T_2$
WHERE $T_1.key = T_2.fkey$ AND $T_1.a > x$

Assume there is no other index available. We use a MapReduce job to process the query. Before the MapReduce job starts, we apply the *Filter* operator on $T_1.a$ to get a filter array $S_r$, which represents the tuples satisfying the predicate. Then we prune table $T_1$ in two steps. In the first step, the data chunks in DFS that do not have any record in $S_r$ are pruned. This can reduce the number of mappers and improve the performance significantly [53]. In the second step, each mapper loads $S_r$ into memory when it starts up. The mapper processes tuples with continuous row IDs, as tuples are stored by their row IDs. Thus, the mapper can know the ID of the next tuple and test it against $S_r$. If the tuple does not qualify $S_r$, it is filtered out without a need to be parsed from the input stream. Parsing is another key factor that affects the performance of MapReduce, which can be effectively avoided by using BIDS.

## 3.4 Index Distribution and Maintenance

In our system, each distributed index node is assigned with a certain key range. When a new index node joins the system, it obtains information about the current index from the index manager. It then selectively loads the bit arrays, whose keys are covered by the node's key range. In this way, the BIDS index is cached in the distributed memory of the index nodes.

### 3.4.1 Distributing the BIDS Index

In our system, the index service is loosely connected with the underlying storage system and the upper-layer query engine. This is a popular design in Cloud systems, which makes the index service scale up and down independently of other services in the system. This feature is extremely important for our BIDS index, because as mentioned before, BIDS is designed as a lightweight index, which can be deployed non-intrusively in existing systems.

To exploit the shared memory, we distribute the BIDS indexes among the index nodes by applying the Chord [87] protocols. We generate a unique ID $K_i$ for each index node $n_i$ by using consistent hash function $\mathcal{H}$. Suppose we have $m$ index nodes $(n_1, n_2, ..., n_m)$, sorted by their IDs in ascending order. Node $n_i$ will be responsible for the key range $(K_{i-1}, K_i]$. The exception is node $n_1$, whose key range consists of two parts, $[0, K_1]$ and $(n_m, max\_key]$.

To distribute the BIDS index across nodes, we adopt a bit array as the basic unit of distribution. Suppose we have a BIDS index of column $c$ in table $T$ with $k$ bit arrays, $(b_1, b_2, ..., b_k)$. As mentioned before, we use a mapping function $m$ to retrieve the value of the bit array. Namely, $b_i$ is built for value $m(b_i)$. We generate a key for each bit array by concatenating its table name, column name and the value as follows:

- $key_{b_i}$=$T$.name + $c$.name + $m(b_i)$

Then, bit array $b_i$ is mapped to the key $\mathcal{H}(key_{b_i})$. There is a unique index node, $I$, whose key range covers the key. $I$ is responsible for handling the requests for $b_i$. Its role is similar to the region server in HBase [2]. To speed up the index lookup, we will buffer $b_i$ at index node $I$.

### 3.4.2 Load Balancing

Suppose each index node contributes $M$ bytes memory for caching the index, and the BIDS index is $N$ bytes. Ideally, we need $\lceil \frac{N}{M} \rceil$ nodes to cache all indexes in memory. However, in practice, more nodes are required as

- Even with consistent hashing, the load across nodes in Chord will not be balanced. It is estimated that the ratio of maximal load to minimal load is about $\log_2 m$ [87], where $m$ is the number of index nodes. Thus, some nodes may need more memory than others to cache the allocated index data.

- As we adopt WAH encoding, different bit arrays are encoded in different ways. Therefore, their sizes may vary a lot. This also leads to load imbalance.

**Algorithm 4:** LoadBalance(Node $n_i$, BitArrays $B$, MemorySize $M$)

```
1:  if sizeof(B) > M then
2:      sort bit arrays of B by keys in ascendant order
3:      p=n_i.getPredecessor()
4:      s=n_i.getSuccessor()
5:      x1=p.size, x2=s.size, x3=sizeof(B)
6:      for i=0 to B.size() do
7:          if B[i].size + x1 < M then
8:              x1 = x1 + B[i].size, x3 = x3 − B[i].size
9:      if x3 > M then
10:         for j=B.size() to i do
11:             if B[j].size + x2 < M then
12:                 x2 = x2 + B[j].size, x3 = x3 − B[j].size
13:     adjusting p's ID to B[i].key
14:     adjusting n_i's ID to B[j].key
15:     if x3 > M then
16:         n_j=getLigthlyLoadedNode()
17:         if n_j=null then
18:             n_j=buyNewNode()
19:         n_j.key = n_i.key
20:         adjusting n_i's ID to (n_i.key+p.key)/2
```

To address the above problems, we use an active load balancing approach. If the index node detects that its assigned bit arrays cannot be fully buffered in the shared memory, it invokes the balancing process given in Algorithm 4. The node first sorts its bit arrays by their keys (line 2). Then, it tries to forward part of its bit arrays to its predecessor (lines 6 to 8) and its successor (lines 10 to 12). In fact, we just need to change the IDs of the index nodes (lines 13 and 14). The index nodes will automatically dismiss or reload the bit arrays. Finally, if the node is still overloaded, the load balancing process tries to find a lightly loaded node (line 16). The selected node sheds its load to its adjacent nodes and rejoins the overloaded node to share its load. If we are not able to find such a node, we need to request for a new index node from the service provider[5].

### 3.4.3 Index Maintenance

The BIDS index is designed for read-mostly applications. It can provide high performance for queries with less maintenance cost. It can also handle infrequent updates efficiently. To update the index, we adopt a similar update strategy as BigTable [21] and Hbase [2].

In BIDS, we always index the latest version of the data. Therefore, we provide the

[5]At this moment, we do not consider storing the BIDS index on disk.

same consistency (eventual consistency) as BigTable [21] and Hbase [2]. The update model of BIDS is based on the characteristics of the bitmap index. The bitmap index cannot be frequently updated, as updating the index will lock the whole bit array, blocking all the other accesses. However, rebuilding the bitmap index is cost effective compared to other index structures. Thus, in BIDS, we adopt a "cache and rebuild" model. As shown in Figure 3.1, the updates are handled by the update manager. Recent updates are cached in a buffer and when the buffer is full, the update manager updates the data files and recomputes the BIDS index.

When we flush the updates into the data files, we rebuild the BIDS index simultaneously. This can be done efficiently by combining the old BIDS index and the updates. Specifically, we need to

1. Reset the bits of existing records, if they have been updated.

2. Create new bits for index, if new records are inserted or new values of a column are observed.

In this way, the size of the BIDS index will keep increasing and we cannot get a good compression ratio, because updates violate the order of data (sorting is important for WAH encoding). Hence, a total index rebuilding is required after too many updates are applied.

When processing a query, the query engine needs to ask the update manager to adjust the result. We build some in-memory indexes to speed up the data retrieval in the update buffer. When the index rebuilding process starts, the update manager flushes current update buffer to disk and opens another buffer for accepting new updates. This strategy is similar to Oracle's hot backup [79]. The old buffer acts as our "offline log" and the new buffer simulates the functionality of "online log". We keep the old index to enable query processing during the rebuilding process. When the new index is ready, the old one is replaced.

## 3.5 Performance Evaluation

BIDS is evaluated on our in-house cluster. In our cluster, the master node is a Dell PowerEdge R610 with dual CPU (each CPU has 4 cores, 2.4GHz). The slave node is powered with an intel Xeon CPU 2.4GHZ and 8GB memory. The nodes are connected with 1Gb bandwidth switch. We use TPC-H [4] data set as our benchmark. The scale factor ranges from 20 to 1000. When scale factor equals to 100, we generate a 100GB dataset on disk. In the current implementation, Hadoop and its file system, HDFS, are employed as our processing engine and storage system, respectively. Table 3.6 shows the default settings of our experiments.

| Parameter | Default Value |
|---|---|
| Hadoop Cluster Size | 50 |
| TPC-H Scale Factor | 100 |
| Size of Data Chunk | 512MB |
| Memory Per Indexing Tracker Process | 256MB |

Table 3.6: Experiment Settings

Besides $Lineitem$ table, there are 38 non-key attributes in $Part$, $Customer$, $PartSupp$, $Orders$ and $Lineitem$ tables. BIDS index are built for 28 attributes, among which 22 indexes are full indexes and the rest are partial indexes.

Each experiment is repeated 10 times and the average result is presented. Before a new experiment starts, we clear the cache of the file system. To avoid interference from other jobs, all nodes are exclusively reserved for our experiments. We evaluate the performance of BIDS in processing both OLAP and high-selective queries. For OLAP query, TPC-H $Q_6$ and $Q_{14}$ are used as our query templates to generate random queries.

$Q_6$: SELECT sum(extendedprice*discount) as revenue
FROM Lineitem
WHERE shipdate$\geq x$ AND shipdate$< x+1$ year AND
discount$\geq y$ AND discount$< y-0.02$ AND
quantity$< z$

$Q_{14}$: SELECT 100*sum(case when P.type like 'PROMO%'
then extendedprice*(1-discount) else 0 end)/
sum(extendedprice*(1-discount))
FROM Lineitem L, Part P
WHERE L.partkey=P.partkey and L.shipdate$\geq x$
and L.shipdate$< x+1$ month

For the high-selective query, we use the following simple query to evaluate the efficiency of BIDS.

SELECT extendedprice
FROM Lineitem
WHERE shipdate=$x$ and discount=$y$ and taxt=$z$

All parameters, except $x$, follow the uniform distribution in their domain. To simulate a skewed query pattern, $x$ follows Zipf distribution, namely about $80\%$ queries request for records in 1993-1994. The storage cost and query processing time are used as our metrics.

### 3.5.1 Storage Cost

Figure 3.4 shows the compression ratio of $Lineitem$, which is computed as the data size in disk divided by its BIDS index size. We observe an extremely high compression ratio for BIDS index – our BIDS index can be between 27-45 times smaller than the original data size. Moreover, as the data size increases, BIDS achieves a better compression ratio. This can be attributed to the use of WAH encoding, which can effectively reduce the size of sorted data.

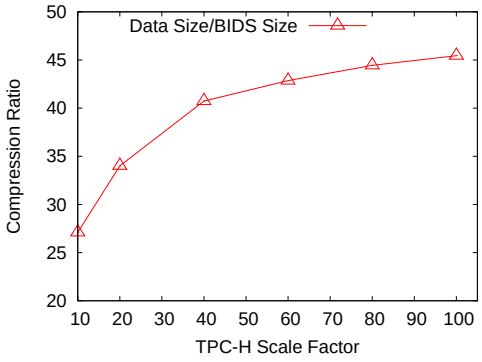



Figure 3.4: Compression Ratio.

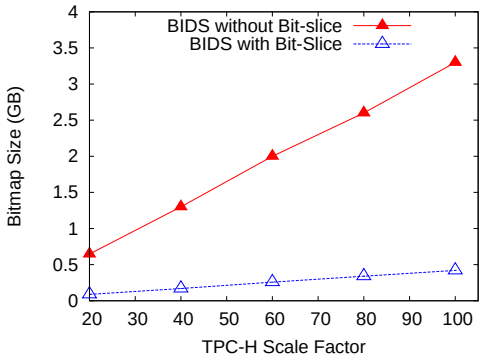



Figure 3.5: Effect of Encodings.

Another reason of the high compression ratio is due to the adoption of bit-sliced encoding scheme, whose effect is depicted in Figure 3.5. We build BIDS index using different encoding schemes for column $quantity$, which has 50 unique values. The index encoded by bit-sliced encoding is much smaller than the one with WAH encoding only. And the saving increases for a larger dataset.

However, even with bit-sliced encoding, fully built BIDS may incur high storage overhead for some columns. Figure 3.6 shows the benefits of our partial indexing scheme. The figure compares two indexing schemes for column $extendedprice$, which has about 1 million unique values. For this experiment, we run variations of TPC-H $Q_9$ for 10 minutes to

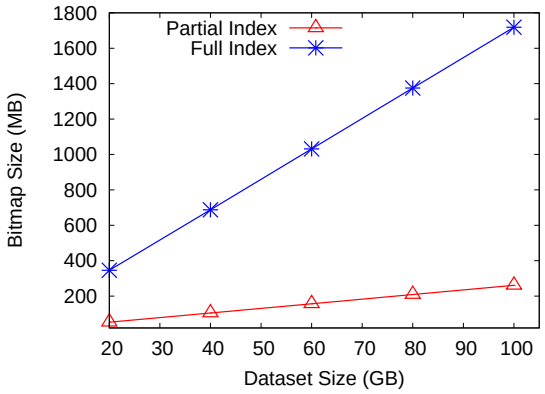



Figure 3.6: Effect of Partial Indexing.

warm up the system and measure the size of the partial index for $extendedprice$. Figure 3.6 verifies that partial indexing is more cost-effective than full indexing in terms of storage overhead. As we shall see shortly, when queries follow a skewed distribution, partial index can be used to answer most queries at a much lower storage cost.

### 3.5.2 Index Construction Cost

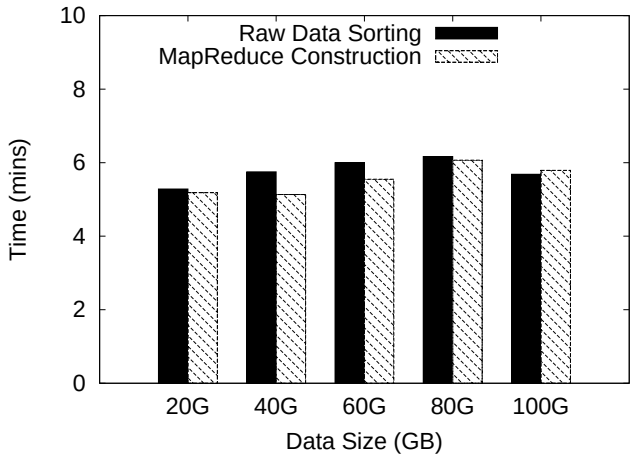



Figure 3.7: BIDS Construction Cost.

BIDS can be built and rebuilt efficiently by running a handful of MapReduce jobs. The cost of BIDS index construction composes of two parts, data sorting cost and index building cost. Sorting is adopted to achieve better compression ratio, which is used in the index initialization as preprocessing. Figure 3.7 shows the time of building BIDS index for $Lineitem$. When data size increases, we also increase the number of cluster nodes. In particular, we set $\frac{TPC-H\_factor}{node\_number} = 2$. Namely, we use 50 nodes to handle TPC-H 100GB dataset. As we apply MapReduce to create the index in parallel, only a few minutes are

required. Compared to BIDS, building B-tree index for a single column may take an hour for 10G TPC-H dataset [4].

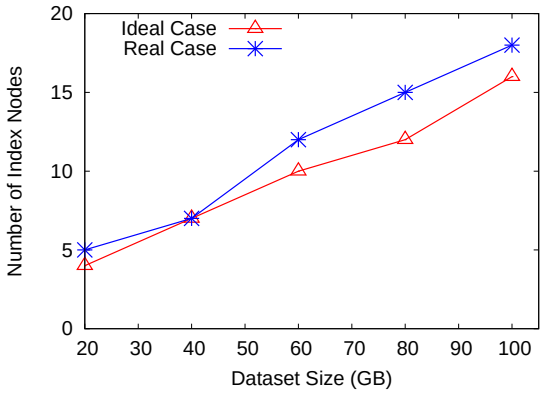
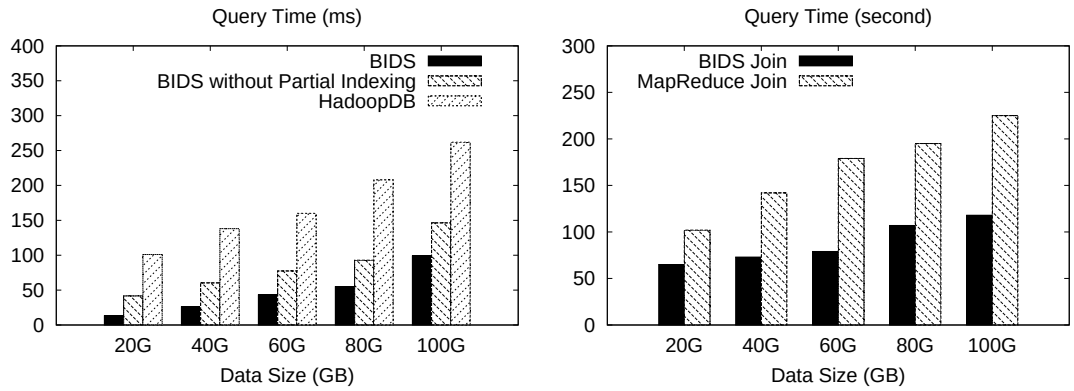



Figure 3.8: Efficiency of Memory Management.

In BIDS, all index data are buffered in distributed shared memory. When the system starts, we set the initial number of index processes to 5, each of which shares 256MB memory for buffering index data. In the case where the index data overflows the memory, Algorithm 4 is invoked to create more index processes and perform load balancing. Figure 3.8 compares the number of required index processes in the ideal case and in practice. It demonstrates the effectiveness of our load balancing algorithm.

### 3.5.3 OLAP Performance




(a) TPC-H Q6 (b) TPC-H Q14

Figure 3.9: OLAP Performance.

In Figure 3.9a, we show the performance of BIDS for TPC-H $Q_6$. We consider two cases, namely the naive MapReduce and MapReduce powered by BIDS. These two processing models differ in that the latter utilizes BIDS operators to take advantage of BIDS

index. Typically, data are pruned by the filter operator in the *map* phase, then are passed to the materialization operator invoked in the *reduce* phase to come up with the aggregation results. We also take the effect of partial index into account. Initially, there is no index for column $extendedprice$ at system start time. After the system runs for a while, partial index is adaptively built for $extendedprice$ and the queries can be directly answered by searching BIDS index. Figure 3.9a shows that BIDS index performs better than conventional MapReduce processing in both cases. BIDS requires only approximately 20-40% of the time required for MapReduce processing.

BIDS can be used in multi-relational queries to prune tables before join processing. Figure 3.9b shows the performance of BIDS in processing TPC-H $Q_{14}$, which joins $Lineitem$ and $Part$ on $partkey$. BIDS index wins by a wide margin, and effectively reduces the processing cost to about 50% that of MapReduce processing.

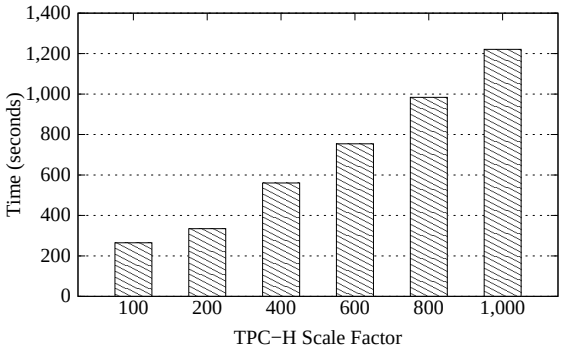



Figure 3.10: Scalability of BIDS.

Figure 3.10 shows the scalability of BIDS, where we increase the data size from 200G to 1T. The experiments are conducted on the 50-node cluster and we use template $Q_6$ to generate the test queries. In Figure 3.10, the query processing time is proportional to the data size, which indicates that BIDS is a linear scale-up indexing scheme.

### 3.5.4 High-Selective Query Performance

One advantage of BIDS index is the ability to support various types of queries. Figure 3.11 shows the performance of high-selective queries, which are processed by the index manager directly (MapReduce job is not necessary as only a few tuples are retrieved). Similar to the last experiment, two cases are tested, namely BIDS with partial index and BIDS without partial index. When partial index is created, most queries can be answered by BIDS index directly. The main cost comes from the computation cost of bit-wise operations. On the contrary, when partial index is not ready, we need to retrieve tuples from HDFS, which incurs significant overhead due to the inefficient random I/O.

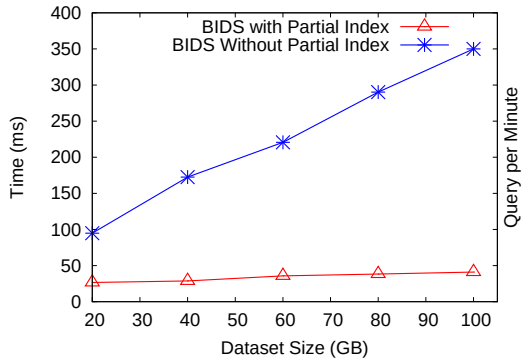



Figure 3.11: Performance of OLTP.

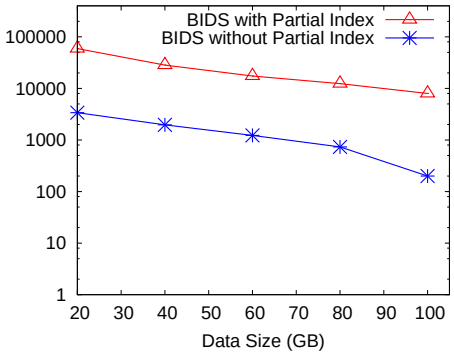



Figure 3.12: Mixed Workload.

To evaluate the performance of update, in Figure 3.12, we measure the throughput of mixed workload who runs queries and updates concurrently. The ratio of the number of queries to the number of updates is set to 1. Among the update requests, half are modifications to existing data and the rest are new insertions. Figure 3.12 shows that BIDS can handle highly concurrent mixed workload efficiently. This is because all updates are processed in memory. However, when the update buffer is full, we need to flush the updates out to disk and rebuild the index. Figure 3.13 studies the effect of index rebuilding.

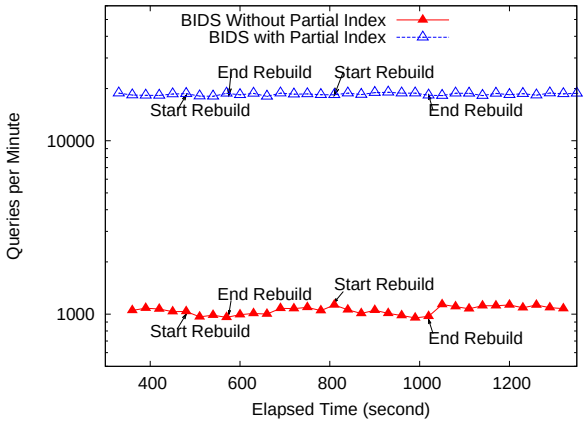
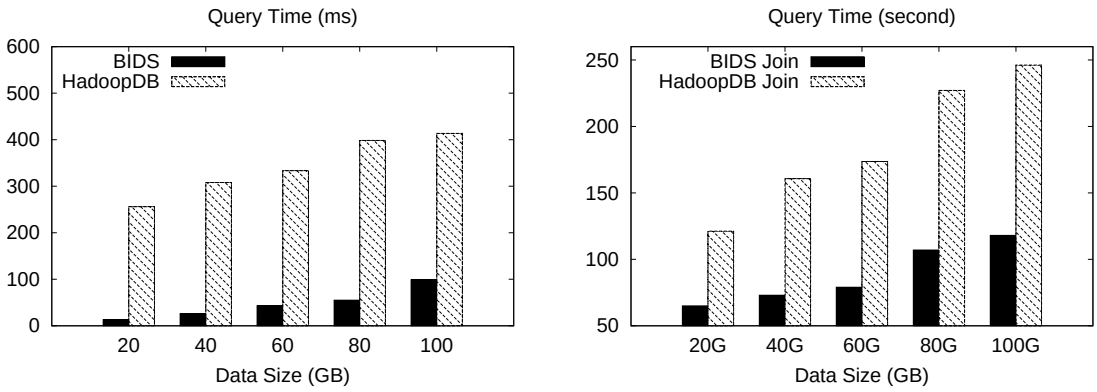



Figure 3.13: Effect of Index Rebuilding.

The buffer size is set to 1MB for the case where no partial index is built for $extendedprice$. To achieve a similar rebuilding frequency, we set the buffer to 12MB for BIDS with partial index. As observed from Figure 3.13, the update buffer becomes full for about 400 seconds. Therefore, given 1GB update buffer, the intervals of rebuilding process are 9.4 hours and 113.6 hours for BIDS with partial index and without partial index, respectively. The rebuilding process is invoked infrequently where a reasonable large buffer is provided. Moreover, based on the results in Figure 3.13, without partial index, the performance is

only slightly affected by the rebuilding process, as query process and rebuilding process compete for the HDFS. When partial index is ready, most queries can be answered via BIDS index and the performance is not affected by the rebuilding process.

### 3.5.5 Comparison with HadoopDB




(a) TPC-H Q6 (b) TPC-H Q14

Figure 3.14: Comparison with HadoopDB.

In this experiment, we deploy HadoopDB on our in-house cluster and compare its performance with BIDS using the same query templates. Before each experiment, we empty the cache of Postgres. Therefore, in our configuration, the HadoopDB starts without a warmup process. Figure 3.14a and Figure 3.14b show the comparison results for $Q_6$ and $Q_{14}$. For single-table queries, BIDS performs 2-5 times better than HadoopDB. This is because instead of scanning the original table, BIDS can answer the query partially or fully via the compact index data, resulting in a big saving for I/O costs. For multi-table queries, BIDS still outperforms HadoopDB. But as BIDS index is only used in filtering, the performance gap is not that significant as the single-table case. In fact, HadoopDB benchmark requires the tables involved in a join to be co-partitioned on their join key in advance [7]. This strategy pushes joins to local DB nodes to avoid broadcasting data among the cluster nodes. However, such a method only facilitates limited joins on a predetermined join key. On the contrary, BIDS does not need co-partitioning as it can be exploited at the filtering phase instead of the joining phase, and thus BIDS is able to benefit almost all joins. Moreover, in HadoopDB, co-partitioning yields inevitable preprocessing cost, while BIDS does not require such preprocessing. The cost of co-partitioning is not shown in Figure 3.14b.

## 3.6 Summary and Contributions

In this chapter, a bitmap based indexing scheme, BIDS, is proposed to manage large amount of data in the Cloud. BIDS index is storage efficient and easy to maintain, which makes it more scalable. It is built on top of the underlying DFS and cached in the distributed memory. BIDS adopts WAH encoding, bit-sliced encoding and pre-sorting to ensure compactness. To further reduce the index size, the index is dynamically tuned based on the query patterns. We also introduced BIDS-based query processing. The query operators are transformed into a set of bit-wise AND/OR operators, which can be handled more efficiently.

In summary, the technical contributions of this work are as follows.

- BIDS is generated by a few MapReduce jobs. It can be created from scratch in minutes for 1TB dataset in a small cluster.
- In addition to the application of WAH encoding and bit-sliced encoding, BIDS adopts a novel and query efficient partial indexing technique to reduce the size of indexes. The partial index is built as a by-product of query processing, and hence its maintenance cost is negligible.
- A new query processing strategy that employs BIDS within MapReduce is designed to achieve high performance for various types of queries, such as range queries and join queries.
- BIDS is buffered in the distributed shared memory based on the Chord [87] protocols, which enables efficient search over distributed indexes.
- A new "cache and rebuild" update strategy is designed to dynamically refresh the index to handle updates in a batch mode.
- Extensive experiments are conducted to depict the efficiency of BIDS and we compare BIDS with HadoopDB [7] on TPC-H benchmark.

This work lays the ground for our further research on index utilization in MapReduce system. Based on the lessons learned from this work, in the next chapter, we will generalize index usage in MapReduce.

# CHAPTER 4

# Scalable Generalized Search Tree

## 4.1 Motivation

By design, traditional parallel database systems are optimized for fairly static environments with a relatively small number of high-end machines. While this architecture provides the desired performance, its capability is limited in scaling dynamically with loads and needs. To take advantage of dynamic cluster environments comprising a large number of commodity machines, MapReduce was first introduced by Dean and Ghemawat [30] to simplify the building of web-scale inverted indexes, and the framework has gained fast popularity as the state-of-the-art of data parallel programming model.

However, based on the evaluation of an open-source implementation of MapReduce, namely Hadoop[1], the framework has been noted to yield sub-optimal performance in the database context [76]. In generic MapReduce, data access are resorted to brute-force parallel scan. Moreover, given a set of input data partitions stored on DataNodes, the execution framework of MapReduce will initiate map tasks on all input partitions. For certain types of analytical queries, it would suffice to access only a subset of the input data to produce the result. Other types of queries may require focused access to a few tuples only that satisfy some predicate, which cannot be provided without accessing and processing all the input data tuples. In both cases, it is desirable to provide a selective access mechanism to data, in order to prune local non-useful data at a DataNode from processing as well as pruning entire DataNode from processing if necessary. In traditional RDBMSs, this problem is solved by means of *indexing*.

Consider the following example query which tries to generate the statistics about the

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

regional behavior of users in a certain age group:

```
SELECT count(*)
FROM mobile m, user u
WHERE m.x< x_0 + α and m.x> x_0 − α and
m.y< y_0 + β and m.y> y_0 − β and m.uid=u.uid
m.dataUsage>3000MB and u.age>20 and u.age<30
GROUP BY u.age
```

Using MapReduce, the query is translated to a MapReduce job comprises of two types of *mappers*. One type of *mapper* scans the mobile table and filters the tuples based on the data usage and location information. The other type scans through the user table to retrieve the users with ages within the query range. Both types of *mapper* shuffle the data to the *reducers* by using *uid* as the partition key. Then a *reducer* can generate the partial results for its designated *uid*. But to merge the users from the same age group, one additional MapReduce job is required to aggregate the results using *age* as the key .

In fact, the above MapReduce query processing strategy incurs unnecessary I/O overheads, which can be effectively avoided by using **indexes**. The challenge, however, is that we need to build various types of indexes to support the query. For example, an R-tree index can be employed to locate the mobile phones within the range $[(x_0-\alpha, x_0+\alpha), (y_0-\beta, y_0+\beta)]$. An un-clustered B$^+$-tree index can be built for attribute *dataUsage* to track user's data consumption. For the user table, a clustered B$^+$-tree index is preferred as it can efficiently prune the users that are not in the qualified age group. With those indexes available, the *mappers* can push down query predicates for the indexes to evaluate and scan only the tuples that contribute to the query result, in which way query performance can be significantly improved.

There have been several proposals on distributed index schemes in cloud environments. For example, a distributed B$^+$-tree-like index was proposed to support single-dimensional range queries [101]. To facilitate multi-dimensional queries, SpatialHadoop [38] realized multi-dimensional indexes in Hadoop using specialized index operators, and can support several types of spatial indexes, such as R-tree and grid files. Another distributed R-tree-like index to support multi-dimensional range and $k$-NN ($k$-nearest neighbors) queries was introduced in [99]. Our previous work on bitmap index [62] provides the functionality for MapReduce to process numeric data with limited range values. However, these index schemes are specialized in a certain type of index. Moreover, these distributed index schemes need to be implemented and deployed separately on the same cluster. Such an approach results in high index maintenance overheads, and lacks code and interface reuse. To facilitate comprehensive analytical queries in MapReduce, it is desirable for the system to support multiple indexes of different types at the same time.

More recently, a generalized distributed indexing framework based on Cayley graph model has been proposed to address the scalability and performance issues of supporting a large number of indexes of different types in dynamic cluster environments [24]. Although this work provides a generic framework for the declaration and implementation of scalable distributed indexes, it requires users to define a data mapping function for each specific index (for example, mapping from multi-dimensional to single dimensional data). This could result in inefficiency of range query processing because such data mapping may incur redundant I/Os for the system having to scan false positive candidates.

In summary, the challenges of incorporating indexes for MapReduce and other large-scale data processing systems are:

1. To support different types of applications and queries, a general indexing framework is required which can be used to build all popular indexes, such as $B^+$-tree index and R-tree index, for the distributed systems. It should also provide unified interfaces for users to implement new types of index.

2. The framework should work as a non-intrusive component for existing systems such as MapReduce so that the previous algorithms written for those systems do not need to be modified to exploit the benefit of index-base processing.

3. As an index service for parallel data processing, the design of index framework must consider the efficiency, reliability and scalability as its first class citizen.

Based on the above rationale, in this work, we present an distributed index framework, *ScalaGiST* – Scalable Generalized Search Tree – which is intrigued by classical Generalized Search Tree (GiST) [45]. GiST provides functionalities of various types of database search trees in a single package, while *ScalaGiST* is designed for dynamic distributed environments such as in-house clusters and public clouds so as to handle large-scale data sets and adapt to changes in the workload while leveraging commodity hardware. *ScalaGiST* is extensible in terms of both data and query in that it enables users to define indexes for new type of data and provides efficient lookup over the index data as built-in functions without the need of data mapping as being used in other distributed indexing frameworks [24, 70].

Indexes in *ScalaGiST* are distributed and replicated among index servers in the cluster for scalability, data availability and load balancing purposes. *ScalaGiST* develops a light-weight distributed processing service to process the request in parallel and effectively reduce the overhead of searching over a large index. *ScalaGiST* is designed as an indexing service and can work with other systems in a non-intrusive way. In Section 4.4, we show how to embed *ScalaGiST* into the execution of MapReduce-based systems by only

launching appropriate map tasks on selected data chunks containing records (for primary indexes) that satisfy the query predicate. This strategy creates opportunities in reducing the startup cost of MapReduce jobs, and most importantly, avoids unnecessary I/Os and computation that do not eventually contribute to the query results. While secondary indexes facilitate a more direct location of data of interest, they may incur non-negligible cost due to random accesses to the base data. Therefore, *ScalaGiST* develops a data access optimizer to compare two possible query execution plans, namely index scan and full table scan, and choose the better plan before running the query.

## 4.2 Architecture Overview

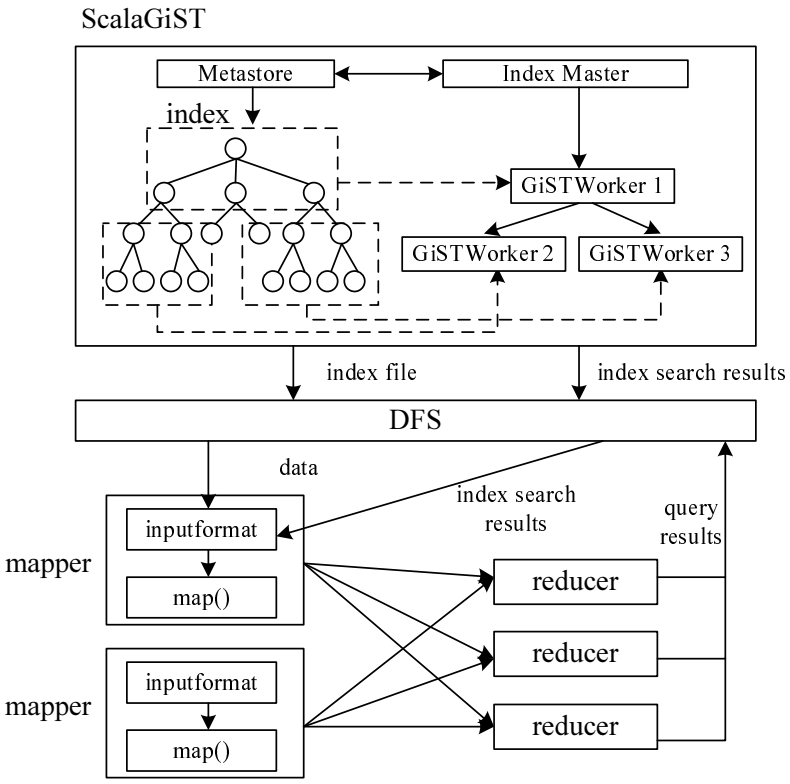



Figure 4.1: Overview of ScalaGiST.

*ScalaGiST* is designed as a scalable and non-intrusive indexing framework for MapReduce systems as illustrated in Figure 4.1. The index is organized as a tree structure and stored as a sequential file in the DFS (HDFS in this paper). Similar to a normal DFS file, the index file is also partitioned into multiple chunks. Each chunk, in fact, contains the index data of one or multiple sub-trees of an index. Given limited memory, *ScalaGiST* selectively loads some index chunks into memory. More specifically, *ScalaGiST* employs a metastore to collect the query statistics, based on which a prediction model is applied to generate a caching strategy to maximize the performance (the details will be discussed in the next section.) If multiple indexes are created, the metastore also maintains the basic information of the index including:

1. The DFS file for which the index is constructed. By default, *ScalaGiST* considers all DFS files as an unstructured format, where each line contains a key and a value. If it is built for the relational data, the user should define a parser for the DFS file and specify the indexed column.

2. The role of the index, e.g. primary and secondary. Primary index is built for the

sorted data and if the data are not sorted by the indexed column, *ScalaGiST* will invoke the tera-sort algorithm of MapReduce to do the sorting.

3. The type of indexes. *ScalaGiST* includes B-tree index and R-tree index in it implementations. For other customized indexes, the user can register their index types in the metastore via the interface provided by *ScalaGiST*.

To process index search requests, *ScalaGiST* develops a light-weighted distributed processing service which includes an index master and multiple index workers. As shown in Figure 4.1, each index worker handles one index chunk by scanning the file and materializing the index in memory. Among the workers, a specific worker (worker 1 in Figure 4.1) is responsible for the root node. Once receiving a search request, the index master forwards it to the worker hosting the root node, which progressively forwards the request to the other workers. When receiving the request, all workers start the search concurrently to exploit the parallelism to maximize the performance. To reduce maintenance overhead, when no query is being processed, the worker process releases all its resources.

The search results of *ScalaGiST* are offsets of the DFS that refer to the tuples that satisfy the predicates. The offsets are flushed back to the DFS as a temporary file. In *ScalaGiST*, we provide a specific ***IndexInputFormat*** for MapReduce runtime to read the temporary files. In the ideal case, if *ScalaGiST* finds that a data chunk cannot contribute to any query result, the corresponding *mapper* will not be scheduled. In most cases, the *mapper* will adopt the skip-and-scan strategy to read a few tuples that qualifies the predicates, which effectively reduces the processing cost.

## 4.3 System Implementation

In this section, we list the interfaces of *ScalaGiST* and the techniques adopted in our implementation. More specifically, we discuss how multiple indexes are used together to facilitate the query processing and how the memory is exploited to reduce the index search overhead.

### 4.3.1 Interface of ScalaGiST

The essence of *ScalaGiST* is to provide template algorithms for traversal and modification of the tree structures distributed in dynamic environments. These algorithms are designed and implemented to work with a generic class of data key.

Unlike classical B-trees whose keys typically are numerical values or short strings, *ScalaGiST*'s keys are instances of a user-defined class overrides the abstraction key class. This capability allows users to define new types of indexes by customizing the key class.

As an illustration, *ScalaGiST* can be instantiated as a distributed $B^+$-tree-like index structure by defining keys as ranges of numbers, which means that all index entries descending from a certain index node have the values between the range. Similarly, *ScalaGiST* can be instantiated as a distributed R-tree-like index structure by defining keys as bounding boxes so that all index entries descending from a certain index node are bounded by the box[2].

Overall, in order to instantiate *ScalaGiST* as a specific type of search tree, the only thing that users are required to do is to define what represents a key, and implement, i.e., override, abstracted methods in the key class as discussed below. These methods will be invoked at runtime by the template algorithms implemented within the *ScalaGiST* framework to realize basic tree operations such as search and modification.

- **Consistent**$(N.p, q)$**.** This method provides the basis for guiding the search operation correctly. It takes as input two parameters, namely a key predicate $p$ of a tree node $N$ and a query predicate $q$. It returns true if both $p$ and $q$ are satisfied for a given data key, and returns false otherwise.

- **Penalty**$(e, N)$**.** This method provides an indication of the cost if the new index entry $e$ is inserted to the subtree rooted by node $N$. The path that has the least penalty in the tree is chosen for inserting the new entry.

- **Union**$(S)$**.** This method defines how to merge a set $S$ of index nodes. It returns a new key predicate $p$ that evaluates to true for all the index entries contained in or reachable from the index nodes in $S$.

[2]In this chapter, we use the term "node" and "page" interchangeable

- **PickSplit**$(N)$**.** This method is invoked when there is a node split upon the insertion of a new index entry. It decides which index entries stay on the old node, and which ones go to a newly allocated index node.
- **Parse**$(InputStream)$**.** This method reads the binary data from the DFS and parses it into a user-defined tree node.
- **Store**$(N, OutputStream)$**.** This method serializes node $N$ into its binary representation and flushes it back to the DFS.

In *ScalaGiST*, users can define a customized node type, and thereby a new index type, by implementing the above interfaces. An abstract class, *GiSTWorker*, is used as our index processing unit and users should pass the node definition as a template to the worker process. The declaration of *GiSTWorker* class is:

```
abstract class GiSTWorker<AbstractNode>
```

As shown in Figure 4.1, we actually maintain multiple nodes in one *GiSTWorker*. The *GiSTWorker* loads a data chunk of the index file and adopts the `Parse` method to reconstruct the tree nodes. For each node, *GiSTWorker* invokes the user-defined function to process the request. In current implementation, *ScalaGiST* has created two sub-classes of the *AbstractNode*, namely the *BTreeNode* and *RTreeNode* for supporting the B-tree index and and R-tree index respectively. In the experiments, we also show that using *ScalaGiST*, we can build various types of indexes, such as a metric index MTree [27], by overriding the interface functions. The new indexes can provide a scalable performance as well. In the following discussion, we use the R-tree as our running example to demonstrate the index construction and search process in *ScalaGiST*.

### 4.3.2 Tree Methods

#### Index Construction

When a user requests to build an index using *ScalaGiST*, a new MapReduce job is submitted for the index construction. Figure 4.2 illustrates the idea of how an R-tree index is built.

*ScalaGiST* first randomly picks $K$ samples from the indexed attributes and then partitions the key space into $W$ sub-spaces ($W$ is the number of *reducers* used to construct the index), so that each sub-space has the same number of samples. For single dimension case, the partitioning process works as building a equal-depth histogram, while for multi-dimension case, it just simulates the KD-tree algorithm. After the partitioning, $C$ *mappers* are started to scan the data where $C$ is the total number of data chunks. Each *mapper* generates $W$ intermediate files, recording how tuples are distributed to different sub-spaces.

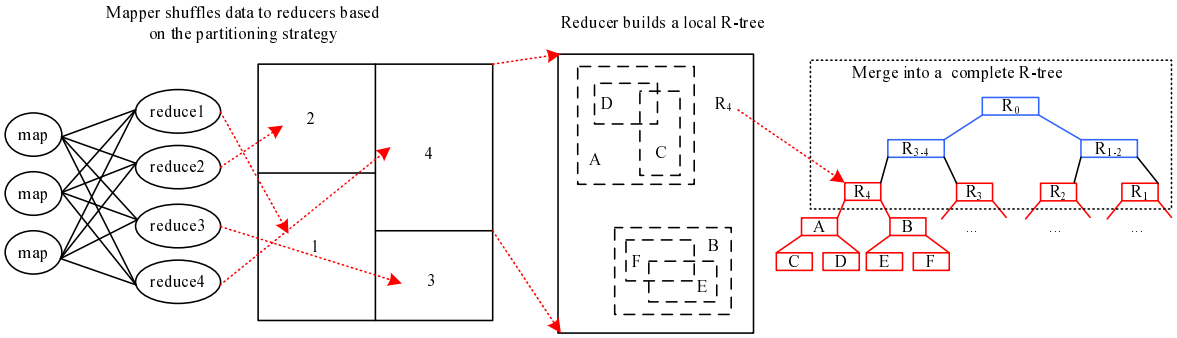



Figure 4.2: Building an R-tree Index.

In the *reduce* phase, each *reducer* collects the intermediate files from the *mappers* for a specific sub-space and constructs a local R-tree using Algorithm 5. In the ChooseSubtree method, we apply the user-defined `Penalty` function to recursively select the subtree that the key should be inserted into until reaching the leaf node. After inserting the new key, we check whether the node needs to split. If so, a recursive split process is invoked using the `PickSplit` function.

After the MapReduce job completes, the index master reads the root nodes of all sub-trees (e.g., $R_1$, $R_2$, $R_3$ and $R_4$ in Figure 4.2) and builds a top R-tree by using those root nodes as its leaf nodes. The top R-tree is the merging result of the sub-trees. Both the top R-tree and the sub-trees are written back to the HDFS as index chunks. In particular, the sub-trees are serialized into a sequential file based on the in-order traversal. The sub-tree in Figure 4.2 is serialized as *C, D, A, E, F, B,* $R_4$. For a large sub-tree, it may be stored as multiple index chunks. Suppose *C, D, A* and *E, F, B,* $R_4$ are two index chunks. For the second chunk, we include a pointer to the offset of $A$ in the first chunk to indicate the position of the left child of $R_4$.

To give a detailed illustration of the index construction process, here we show the

**Algorithm 5:** Insert(Key key)

```
1:  AbstractNode root = getRoot()
2:  if root==null then
3:      createRootNode(key)
4:  else
5:      AbstractNode node = ChooseSubtree(root, key)
6:      node.insert(key)
7:      if node.needSplit() then
8:          Key splitkey=PickSplit(node)
9:          create two new nodes based on the splitkey
10:         notify the parent node about the two new nodes and do the recursive split if
            necessary
```

performance breakdown of indexing 10 GB 2-dimensional data. There are two strategies that are commonly used for R-Tree construction, namely **sequential insertion** (insert data points one by one sequentially) and **bulk-loading**. We implement both methods and compare their efficiency. The data set contains approximately 10 million records.

| | Sequential Insertion | Bulk Loading |
|---|---|---|
| No. of Mapper | 20 | 20 |
| Map Time (second) | 151 | 151 |
| No. of Reducer | 40 | 40 |
| Reduce Time (second) | 1094.2 | 26.63 |
| Sub-Tree Merging (millisecond) | 178 | 178 |

Table 4.1: Comparison of Index Construction Strategies

As shown, bulk loading outperforms sequential insert by a large margin. There are in total 20 mappers launched. Each mapper reads in 512MB of data, maps them to sub-spaces and shuffles the data to 40 reducers. This process takes up to 2.5 minutes (151 seconds) for both methods, inclusive of instantiation time of the job. When a reducer receives the data, it performs R-Tree construction (locally). In this phase, sequential insertion spends 18 minutes due to large amount of keys (over 10 millions), whereas bulk loading is very efficient and costs only 27.19 seconds. The final merge phase reads in the root nodes of local R-Trees as leaf nodes and inserts them into a top layer R-Tree. This phase involves reading the 40 root nodes from DFS and inserting them into an in-memory R-Tree, which is rather fast and can be finished in 178 milliseconds. It is notable that sequential insertion is directly supported by the GiST interface for all types of indexes, while bulk loading requires some customized codes for each index.

### Search

Figure 4.3 shows how the *GiSTWorker* processes a range query. It simulates the typical tree search algorithm. The search starts from the index master which maintains the top R-tree. Based on the search range, it sends the query to *GiSTWorker1* and *GiSTWorker2*. The two workers start the tree search in parallel. When reaching the leaf nodes, a worker checks whether it can return the result or it needs to forward the search message to other workers. Algorithm 6 illustrates the local search process inside each worker. The index master monitors the whole search process. Once it detects that all workers have finished their tasks, it notifies the MapReduce scheduler for further query processing.

On top of range search, we provide two $k$-NN algorithms, one is generic for all indexes defined by the *ScalaGiST* interface and one is specific for the R-tree index. The generic $k$-NN algorithm iteratively expands its search range until $k$ results are obtained. Suppose

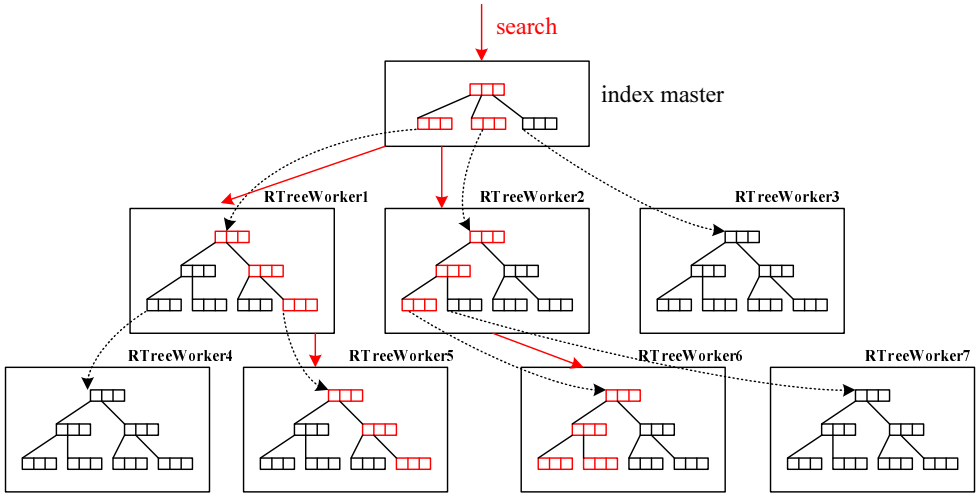



Figure 4.3: Search with R-tree.

**Algorithm 6:** GiSTWorker.Search(Query $q$)

```
1:  Set<AbstractNode> nodes = new Set(root)
2:  while nodes.hasMoreElement() do
3:      AbstractNode n̄ = nodes.next()
4:      if Consistent(n̄.key, q) then
5:          if n̄.isLeaf() then
6:              result.add(n̄)
7:          else
8:              Set<AbstractNode> child = n̄.getChild()
9:              while child.hasMoreElement() do
10:                 AbstractNode c = child.next()
11:                 nodes.add(c)
12: return result
```

the query point is $p = (v_1, ..., v_d)$. We submit an initial query as $Q_0 = ([v_1 - r_0, v_1 + r_0], ..., [v_d - r_0, v_d + r_0])$. If more than $k$ results are obtained, the search completes. Otherwise, we enlarge the search range by $\theta$. So the new query is $Q_1 = ([v_1 - r_0 - \theta, v_1 + r_0 + \theta], ..., [v_d - r_0 - \theta, v_d + r_0 + \theta])$. To avoid repeating the search of the initial query, we also include $Q_0$ when processing $Q_1$. The query will not be sent to the tree nodes that only overlaps with $Q_0$. $r_0$ and $\theta$ are two tunable parameters in the $k$-NN search which have been well studied [91]. Using their analysis, we set $r_0 = \theta$, and $\theta$ is estimated as:

$$\theta = \frac{D_k}{k} = \frac{2\sqrt[d]{\Gamma(\frac{d}{2}+1)}}{k\sqrt{\pi}(1-\sqrt{1-\sqrt[d]{\frac{K}{N}}})}$$

where $D_k$ represents the distance between the $k$th nearest neighbor and the query point

and $N$ is the estimated number of data in the whole space. $\Gamma$ is a function defined as: $\Gamma(x+1) = x\Gamma(x)$ with $\Gamma(1) = 1$ and $\Gamma(0.5) = \frac{\pi}{2}$. We start with an initial $\theta$ value and each time when we complete a range search, we will update our estimation for $D_k$, so the next range is more accurate and tight.

We also include the classic branch-and-bound $k$-NN algorithm for R-tree which provides a near optimal performance [75]. It first retrieves the nearest neighbor to the query point and then computes its minimal and maximal distances to the nearby bounding boxes. The distances are used to expand the search range and decide when to terminate. In fact, both the performances of the two $k$-NN algorithms differ marginally in both our experiments and the theoretic analysis [91].

It is also noted that different index construction strategies has substantial impact on search performance [26]. We use the two R-Trees populated by sequential insertion and bulk loading in the last benchmark, and compare their query performance using range search and $k$-NN search.

| | Sequential Insertion | Bulk Loading |
|---|---|---|
| Range Query (second) | 0.185 | 10.28 |
| $k$-NN Query (second) | 3.97 | 27.19 |

Table 4.2: Comparison of Query Performance

Table 4.2 shows that, although bulk loading speeds up index construction, it compromises the query performance to some extend comparing to that of sequential insertion, because one-by-one insertion allows the index to adaptively pick a better sub-tree in the R-tree to insert and reduces the size of bounding boxes. In the above evaluation, query performances for range query and $k$-NN query are about 55 times and 7 times worse when using bulk loading. Therefore, there is a trade-off between index construction cost and run-time performance. It is up to the users to decide which method to use.

**Insertion**

As an indexing service for the MapReduce system, *ScalaGiST* only supports batch insertion. For a new batch of data, instead of appending them to the existing DFS file, we import them as a new file under the same directory. *ScalaGiST* checks the metastore whether we need to build indexes for the new data and starts the index construction process if necessary. *ScalaGiST* creates a new index tree and registers it in the index master. Therefore, for an increasing dataset, we may have multiple index trees and we will route the query to all trees for processing.

Periodically, *ScalaGiST* merges the index trees to reduce the search cost. Let $T_0$ be the original tree and $T_1$,...,$T_k$ be the new trees. $T_1$,..,$T_k$, in fact, are discarded and we build a

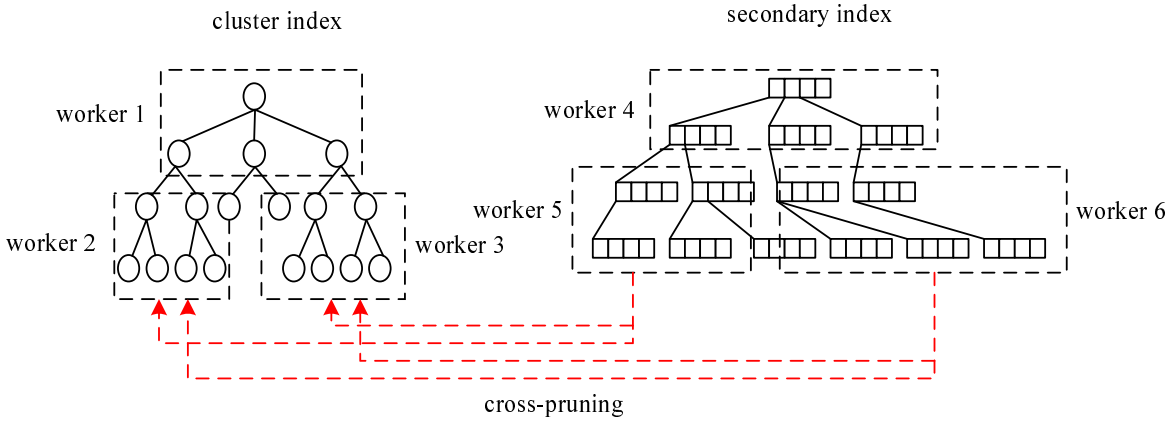



Figure 4.4: Search With Multiple Indexes.

new $T_0$ by inserting their data into $T_0$. We start a MapReduce job to perform the merging. Specifically, in the *map* phase, we scan data files of $T_1$,..,$T_k$ and partition them based on the same partitioning strategy of $T_0$. In the *reduce* phase, the *reducer* loads its specific sub-tree of $T_0$ and inserts the new data into $T_0$. After all *reducers* complete their insertion, the index master will generate a new index tree for all existing data files. For example, in Figure 4.2, the new data are partitioned into four sub-spaces defined by the previous sampling process. In *reducer4*, we load $R_4$ from the DFS and insert the received data into $R_4$'s sub-tree. The insertion process applies the user-defined **Consistent**, **Penalty** and **PickSplit** functions to guide the tree construction. After the MapReduce job, a new $R_4$ covers all existing data in the sub-space is built.

### 4.3.3 Search with Multiple Indexes

One of the most distinguished features of *ScalaGiST* is its capability of supporting various types of index. As shown in our example query in the introduction section, we can build both a B-tree index and an R-tree index for different attributes of a table. In *ScalaGiST*, only one clustered index can be built for a table, while the number of secondary indexes is not limited. Figure 4.4 shows how *ScalaGiST* exploits multiple indexes together to process the query.

Suppose we have a clustered B-tree index which is disseminated to worker 1, worker 2 and worker 3. We also have a secondary R-tree index which is maintained by worker 4, worker 5 and worker 6. Given a query with two predicates, $p_0$ and $p_1$, suppose $p_0$ is on the clustered attribute and $p_1$ is on the other attributes. *ScalaGiST* splits the search into two parts. $p_0$ is forwarded to worker 1 to worker 3, while $p_1$ is forwarded to the other 3 workers. All the workers start their search in parallel. When we complete the search of clustered index, suppose only worker 2 has the query results (DFS offsets that

point to the corresponding tuples). Instead of returning the results to the applications, e.g., MapReduce jobs, worker 2 waits for the search results of the secondary index. Once worker 5 and worker 6 finish their search, they broadcast their results to worker 2 and worker 3. As a matter of fact, worker 3 will not be notified by *ScalaGiST*, as it does not have the query results for $p_0$. On the other hand, worker 2 will merge its results with the results from the other two workers. The final index search results are then returned to the users. The flexibility of *ScalaGiST* allows us to link the workers in an arbitrary way, simplifying the search algorithm design for the multiple indexes.

### 4.3.4 Memory Management

After a worker completes the job, we destroy its memory stacks and reclaim all the used memory. When the next query comes, *ScalaGiST* will wake up the worker and reconstruct its states. Such initialization cost and the cost of loading index nodes from the DFS into memory cannot be ignored. One way to address the problem is to maintain some workers and their states in memory. In other words, those workers are maintained as a "persistent worker" in *ScalaGiST*. They are always running, waiting for receiving the requests from the users. Their states, e.g., the tree structure, are also cached in memory. This is similar to using the RDD as the storage in Spark [106]. However, given limited memory, we must adaptively select the memory-resident workers to maximize the performance.

**Definition 4.1. Benefit of A *GiSTWorker***
*The benefit of a GiSTWorker regarding to a query $q$ is defined as the total size of index tree nodes (except the root nodes) that are required to read from the DFS to process $q$.*

In *ScalaGiST*, we record the last processed $k$ queries in the metastore and use that statistics to measure the benefit of buffering each *GiSTWorker*. So the memory management problem is transferred into an optimization problem:

**Definition 4.2. Optimal Buffering Strategy**
*Given a query set $Q$ and a GiSTWorker set $U$, suppose we only have limited memory $M$, we want to select a subset GiSTWorkers $\bar{U}$ from $U$, so that:*

1. *The memory for buffering GiSTWorkers in $\bar{U}$ is less than $M$.*
2. *For any other subset $\widehat{U} \subseteq U$ satisfying the memory constraint, its benefit is less than that of $\bar{U}$.*

Note that although all *GiSTWorkers* handle the same-size index chunks, when materializing the tree nodes in memory, the *GiSTWorkers* require different sizes of memory

because the index nodes may have different data structures. Therefore, the optimal buffering strategy is, in fact, a set-packing problem which is NP-hard. In *ScalaGiST*, we adopt a greedy-based heuristic approach as shown in Algorithm 7. The intuition is to compute a score for each worker as $\frac{benefit}{memory\_size}$ and rank workers based on the scores. The top ranked workers are set as "persistent workers" which are maintained in memory for speeding up the processing.

**Algorithm 7:** ManageBuffer()

```
1: for each GiSTWorker u ∈ U do
2:     u.score = u.benefit/u.memory_size
3: Heap H = sortByScore(U)
4: while Ū.size < M and H.size>0 do
5:     Worker u = H.pop()
6:     ū.add(u)
7: return ū
```

In *ScalaGiST*, Algorithm 7 is invoked periodically to adjust the buffer strategy. We provide a parameter for users to tune the frequency. By default, Algorithm 7 is invoked only when new indexes have been created since the last adjustment.

### 4.3.5 Tuning the Fanout

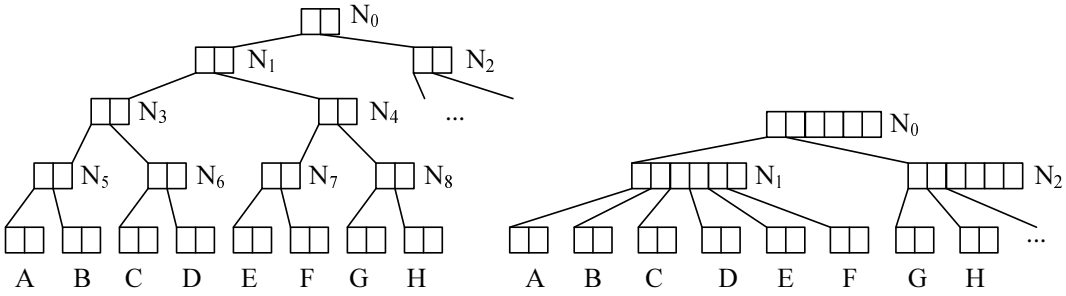



Figure 4.5: Effect of Fanout.

For tree-based index, fanout $F$ affects the search performance. We illustrate the problem using Figure 4.5. Suppose each index chunk can maintain three leaf nodes or 6 pointers of the internal nodes. For the left binary tree, the workers and their tree node assignment is: { *($W_1$: A, B, $N_5$), ($W_2$: C, D, $N_6$, $N_3$), ($W_3$: E, F, $N_7$), ($W_4$: G, H, $N_8$, $N_4$, $N_1$), ...*}. For the right tree, the tree node assignment is: {*($W_1$: A, B, C), ($W_2$: D, E, F), ($W_3$: $N_1$),...*}. Given a query that retrieves data from leaf nodes $B$, $C$, $D$ and $E$, the left tree first forwards the query to $W_4$. $W_4$ then forwards the query to $W_2$ and $W_3$. $W_2$ further forwards the query to $W_1$. There are totally four workers involved in the processing and three workers ($W_1$, $W_2$ and $W_3$) perform their jobs concurrently. On the contrary, in the

right tree, the query is first routed to $W_3$ and then forwarded to $W_1$ and $W_2$. Only two workers can run concurrently. However, the left binary tree requires a longer search path $W_4 \rightarrow W_2 \rightarrow W_1$, while the right tree has a much shorter path $W_3 \rightarrow W_1$. In summary, the fanout has the following two properties:

1. A small fanout can increase the level of parallelism by involving more workers in the processing. This can effectively improve the performance when the query needs to retrieve a large portion of data. But it also incurs more communication costs, when only a few results are required.

2. A large fanout can reduce the search path and hence, lead to a lower communication and I/O cost. However, it may result in load imbalance, as the query is processed by fewer workers.

In fact, most B$^+$-tree style hierarchies in production such as Bigtable [21] and HBase [2] also use a small number of levels and very high fanouts. This is because they are targeting at high-selective queries (e.g., key-based retrieval). *ScalaGiST*, on the other hand, is designed for the MapReduce system. So we also want to benefit the large analytic queries. In *ScalaGiST*, we group the historical queries into two categories, high-selective queries and large analytic queries. Our purpose is to estimate a fanout $F$ that can achieve a good performance for both types of queries.

We apply a coarse estimation and for space limitation, we use the single-dimension index to briefly demonstrate the idea. We assume that the index evenly partitions the key range. Therefore, at level $l$, we have $F^l$ leaf nodes and the domain is partitioned into $F^l$ sub-ranges. Let $t$ be the size of a tree node and $C$ be the size of DFS file chunk. Each worker handles $\frac{C}{t}$ index nodes. Using the partitioning strategy shown in Figure 4.1, we can estimate how the $F^0 + ... + F^l$ nodes are distributed to different workers. Given a list of historical queries $\{q_0, ..., q_k\}$, we can also estimate how many workers are involved for each query. Based on the query pattern, we set two selectivity thresholds $\theta_x$ and $\theta_y$. For queries with selectivity smaller than $\theta_x$, we want to set a $F$ that only one worker is involved in the search concurrently. So we can get a lower bound $\epsilon_{low}$ for $F$. For queries with selectivity larger than $\theta_y$, we want as many index workers as possible in the search process. Namely, the involved index workers are no less than $W$ where $W$ is the number of available cluster nodes that *ScalaGiST* is deployed on. This constraint can generate an upper bound $\epsilon_{up}$ for $F$. Let $\{\theta_0, ..., \theta_n\}$ denote the selectivity of historical queries on table $T$. The total cost can be estimated as (detailed cost model will be discussed in Section 4.4.2):

$$\sum_{i=0}^{n} \lceil \frac{\sum_{j=1}^{\log_F |T|} \theta_i F^j e^2}{W} \rceil$$

We then iterate $F$ in the range of $[\epsilon_{low}, \epsilon_{up}]$ and compute the above equation respectively. *ScalaGiST* selects the $F$ value that minimizes the cost estimation.

## 4.4 Hadoop Integration and Data Access Optimization

Unlike existing proposals [9, 24, 101, 99], our proposed *ScalaGiST* has been designed for seamless integration with Hadoop and its data access optimization algorithm helps MapReduce select an index scan versus a full table scan method depending on characteristics of queries.

### 4.4.1 Leveraging Indexes in Hadoop

One of the major advantages of MapReduce is that it is a generic execution engine and independent of the underlying data storage system. Consequently, to make MapReduce able to read input data from a new type of storage, users only need to extend the ***InputFormat*** abstraction class and provide appropriate implementation for the functions that will be invoked by MapReduce to retrieve data from the new storage. In *ScalaGiST*, we implement an ***IndexInputFormat*** class for *ScalaGiST* so that its data can be accessed by MapReduce. This class overrides the required public methods such as ***getSplits()*** and ***createRecordReader()***.

More specifically, the ***getSplits()*** routine will be called during the starting up of a MapReduce job to identify how the index data are split into chunks, which in turn will determine the number of map tasks that are required to execute the query processing job. Given a query, the system parses its range predicates and composes an appropriate ***Scan*** operator on the corresponding index. All index pages between the start and end keys of the range ***Scan*** operator are included for processing the query. *ScalaGiST* splits the index data at index pages' boundaries, and therefore the number of splits is essentially the number of index pages located within the query range.

During the execution of the MapReduce job, the framework iterates over the splits and calls the ***createRecordReader()*** for each split. Each calling creates a new ***IndexRecordReader*** to access the corresponding index page, and process the index page in two steps: (1) retrieving the base record referred by the index entry, and (2) mapping the record based on the map function customized for processing the query and shuffling intermediate data to the appropriate reduce task.

Overall, in this MapReduce execution with index scan, the selectivity of the query predicate determines the number of map tasks to be launched for processing the query. By utilizing the index, the number of needed maps is restricted to the minimal and only relevant records satisfying the query predicate are retrieved from the base table, thus reducing the task's startup and I/O cost significantly. However, the benefits of this index scan execution do not come for free. In fact, this index scan strategy introduces other overhead that does not exist in the full table scan approach. First, index traversals are needed before the MapReduce execution in order to identify leaf pages where necessary maps should be launched. However, as the height of a practical *ScalaGiST* tree is typically low (and most hot internal index pages are cached in memory) this extra overhead is negligible. Second, scanning the index data in each map to get record pointers stored in index entries also incurs additional I/Os. This part of overhead may not cause serious issues since the index data usually have much smaller size than the base data. The third part of overhead in this index scan approach, which is the most significant one, is the I/O cost of random access to the base data in the case of secondary indexes. Consequently, even though the index access execution provides an alternative option for processing queries, it does not always achieve better performance than the full parallel sequential scan execution. A cost model is therefore essential and proposed in the following section to estimate the performance of the two strategies so that the system is able to choose the optimal one for query execution.

### 4.4.2 Data Access Optimization Algorithm

To identify an optimal access method, we build histograms to collect statistics of data distribution and design a cost model to select the data access plan.

#### Construction of Histograms

At regular time, the system runs a background MapReduce job for constructing histograms of tables. 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. That is, we split $[l_i, u_i]$ into $B$ buckets, and for each bucket, we count the number of tuples whose attribute value falls within the bucket. 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 bucket containing the corresponding attribute value.

To reduce shuffling cost, we customize the combiner function to aggregate key-value pairs within the same bucket so that each mapper only generates at most one key-value

pair for a bucket. In the reduce phase, we group key-value pairs by their $columnID$ and combine the results from multiple mappers. Finally, the metadata of a histogram bucket, including 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, e.g., the histogram for column $a_i$ of table $T$ is stored in "$/histogram/T/a_i$".

**Selection of Optimal Data Access Plan**

After having constructed the histograms for selectivity estimation of range predicates, we proceed to design an algorithm for selection of optimal data access plan.

The base tables are comprised of equal-size ($s_d$) data chunks in the underlying distributed file system (e.g., HDFS ). Consider a query $Q$, we use the function $f(Q)$ to denote the size of data involved in the processing of that query. For full table scan, if the query $Q$ involves multiple tables $T_1$,...,$T_k$, then $f(Q)$ is computed as $\sum_{i=1}^{k} |T_i|s_i$, where $s_i$ denotes the average size of records in the table $T_i$. For index scan approach, $f(Q)$ is estimated as $\sum_{i=1}^{k} g(T_i, Q)s_i$. $g(T_i, Q)$ denotes the number of tuples in the table $T_i$ that satisfy the selection predicates of the query $Q$. In the following discussion, we estimate the cost of *map* phase for processing a table $T_i$, as index is mainly used by the *mappers* to reduce the I/O cost.

**Full scan.** The total number of data chunks in the base tables referred in the query is $\frac{|T_i|s_i}{s_d}$ We need the same number of *mappers* in our processing. The underlying distributed file system (HDFS) ensures that the data chunks are roughly distributed across machines in the cluster. Suppose we have $N$ cluster nodes. Let $c_s$ be the cost ratio of sequential scan. The cost of the slowest node is:

$$c_{pscan} = \lceil \frac{|T_i|s_i}{s_d N} \rceil \times s_d c_s = \lceil \frac{|T_i|s_i}{N} \rceil c_s \tag{4.1}$$

**Index scan.** If the query can be processed by the primary index of $T_i$, we can effectively reduce the number of data chunks in the MapReduce job. The scan cost of the slowest node is reduced to

$$c_{iscan} = \lceil \frac{|T_i|s_i g(T_i, Q)}{N} \rceil c_s \tag{4.2}$$

If the query only involves the secondary index, *ScalaGiST* groups the pointers that refer to the same data chunk and performs random accesses to the base records in sequential offsets. Let $c_r$ denote the cost ratio of random read with sequential offsets. The cost of slowest node is

$$c_{iscan} = \lceil \frac{|T_i|s_i g(T_i, Q)}{N} \rceil c_r \tag{4.3}$$

For a query involving $k$ tables, we normally generate $k-1$ MapReduce jobs to perform the join. If we stick to the left-deep plan, except the first job, the rest jobs join a raw table with an intermediate result table. For an intermediate result table $T_i$, we consider it as a table without indexes (namely, $g(T_i, Q) = |T_i|$). Given two table $T_i$ and $T_j$, the scan cost of slowest node in the *map* phase is:

$$\lceil \frac{|T_i|s_i + |T_j|s_j}{N} \rceil c_s$$

And the cost of primary index scan is:

$$c_{iscan} = \lceil \frac{|T_i|s_i g(T_i, Q) + |T_j|s_j g(T_j, Q)}{N} \rceil c_s \tag{4.4}$$

Similarly, the cost of secondary index scan can be estimated.

Another cost is the index lookup cost. As most internal tree nodes are buffered in memory, our model only computes the network communication cost and the scan cost of leaf nodes. Let $L$ be the number of index workers in the longest search path of the index. So the maximal network cost is $Lc_n$, where $c_n$ is the network cost ratio. If the size of each leaf index entry is $e$, we can maintain approximately $\frac{s_d}{e}$ leaf nodes in one index chunk. Namely, each index worker can handle about $\frac{s_d}{e}$ leaf nodes. Suppose we have $W$ index workers, the index search cost is estimated as:

$$c_{lookup} = Lc_n + \lceil \frac{g(T_i, Q)e^2}{W} \rceil c_s \tag{4.5}$$

The second term in above equation denotes the average cost of each index worker when processing the leaf node scan. For two table join, we need to add up the index search costs of both tables.

**Data access optimizer.** After estimating the cost of the two data accessing schemes (scan and index-based processing), we now present a data access optimization algorithm as a guiding principle for the system to dynamically choose the optimal data access plan for the execution of a specific query with MapReduce. Given a query, we split it into multiple MapReduce jobs $\{j_0, j_1, ..., j_k\}$. For each job $j_i$, we estimate the cost of $c_{pscan}$ and $c_{iscan} + c_{lookup}$ and select the optimal strategy. At regular time, the system runs a background a micro-benchmark on the underlying distributed file system to measure the performance of raw random and sequential I/Os and update the values of $c_s$, $c_r$ and $c_n$ respectively.

## 4.5 Performance Evaluation

We have conducted a series of experiments to evaluate the effectiveness and scalability of *ScalaGiST*. First, we evaluate the performance of *ScalaGiST* using the YCSB benchmark [29]. Then, we compare the performance of *ScalaGiST*-integrated MapReduce with generic MapReduce in processing analytical queries. We also study the performance of *ScalaGiST* in terms of analytic query and multi-dimensional query, and compare its performance with other distributed indexing frameworks, namely Data Mapping [24], SpatialHadoop [38], and RT-CAN [99]. To show the flexibility of *ScalaGiST*, we implement a new index MTree [27] on top of *ScalaGiST* and evaluate its performance on processing multi-dimensional queries. Lastly, we show the effectiveness of *ScalaGiST* in an application scenario involving multiple indexes in a single query.

### 4.5.1 Experimental Setup

The experiments are conducted on an in-house cluster, which includes 64 commodity machines equipped with Intel X3430 2.4 GHz processors, 8 GB of memory, two 7200 RPM SATA disks with 500 GB capacity each, and 1 Gb ethernet. The machines in the cluster are connected via a flat network.

A Hadoop cluster is set up as the infrastructure system for index storage and query processing with *ScalaGiST*. We keep the settings of Hadoop as default. Each machine in the cluster runs three daemon processes and plays multiple roles as a data node for HDFS, a worker node for MapReduce, and an index worker for *ScalaGiST*. The index master process is configured to run on the same machine as Hadoop cluster's master node.

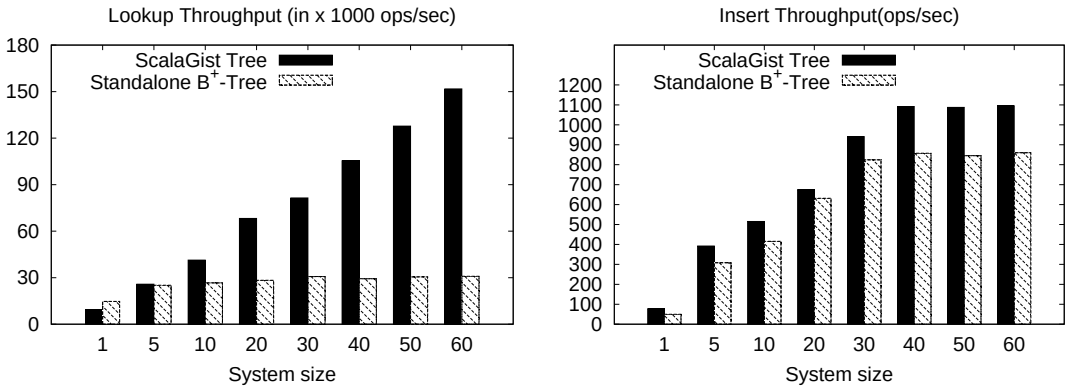
### 4.5.2 Micro-benchmarks

In this test, we study the performance of index construction and index lookup operations with *ScalaGiST* using the YCSB [29] benchmark. We build a $B^+$-tree index using *ScalaGiST*. We generate the following two workloads.

- *Insert.* New records (key-value pairs) are randomly generated and inserted into the system. Note that when working with MapReduce system, *ScalaGiST* only supports batch insertion, but for the indexing service itself, *ScalaGiST* can support realtime insertion. The master forwards the updates to the corresponding index workers who update their local sub-trees. This experiment shows the raw performance of *ScalaGiST* and also indicates the cost of batch insertion using MapReduce.
- *Lookup.* The previously inserted records are searched, with the keys are randomly chosen.

For each index server, we configure the YCSB runtime to instantiate 4 client threads to concurrently access the *ScalaGiST* tree. That is, client workloads submitted to the system scale up much faster than the system size, and index servers will observe heavy loads when the system size is large. The aggregate throughput of the two workloads are measured as the system scales out from 10 to 60 index servers.

We also run the workload on a centralized system with a standalone B$^+$-tree implementation to show the advantages of scalable distributed search trees. The B$^+$-tree is deployed on a Dell PowerEdge R610 server (which has a much higher hardware configuration compared to other commodity machines in the cluster), and is configured to have a cache of 4 GB. The number of concurrent client threads submitting workloads to the standalone B$^+$-tree is configured to be equivalent with the above setting of *ScalaGiST* tree. Before running the insert workload, both the *ScalaGiST* tree and the stand-alone B$^+$-tree are pre-populated with 10,000 records.




(a) Lookup (b) Insertion

Figure 4.6: Micro-benchmark: Aggregated Throughput.

As shown in Figure 4.6a, the *ScalaGiST* tree scales almost linearly with the system size. On the contrary, the stand-alone B$^+$-tree achieves a much lower throughput and its capacity is saturated quite soon due to the lack of the ability to handle concurrent requests. The high throughput performance of *ScalaGiST* is attributed to its scalable architectural design. In addition, with the help of caching internal index nodes, most of the tree traversals along the read path can be finished within one network hop before reaching the appropriate index server to retrieve the desired index leaf page.

Figure 4.6b plots the system throughput for the insert workload. The aggregate throughput grows almost linearly with system sizes at low and medium scales (up to 40 machines). As the workload gets heavier (by increasing the number of machines and hence the number of client threads also scales up four times as much as described in the experiment settings), new insertions incur more network communication overhead and I/O contentions.

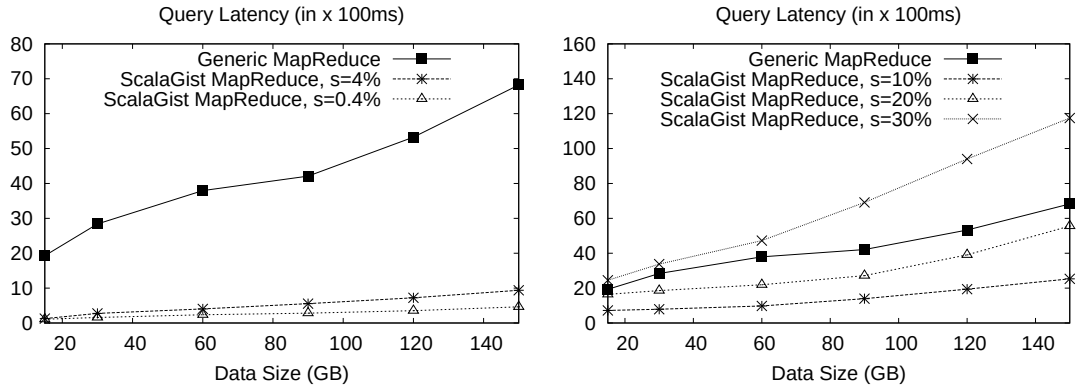
Overall, the above experimental results confirm the elastic performance of raw *ScalaGiST* framework. More index servers can be added into the system to serve the increasing workloads.

### 4.5.3 MapReduce Scan vs. Index Scan

In this section, we compare the performance of *ScalaGiST*-integrated MapReduce with generic MapReduce in processing analytical queries. We conduct the experiment on TPC-H benchmark dataset [4] which models the workload of a decision support system.

```
Q1:SELECT custkey, count(orderkey)
   FROM Orders
   WHERE totalprice ≥ y and totalprice ≤ y + 100
   GROUP BY (custkey)
```

We consider a selective query above on the `Orders` table. The scale factor of `Orders` table is varied from 10 to 100. Under each scale factor, the workload generator produces 1.5 million records for the table. Each record has an average size of 1 KB. Thus, the total data size ranges from 15 GB to 150 GB. The data records are stored in the underlying HDFS and sorted by the selection key, i.e., the $totalprice$ attribute. A 20-machine cluster is set up for this experiment. The *ScalaGiST* index built on the data set is configured to instantiate a scalable B$^+$-tree-like index on the $totalprice$ column.




(a) High selectivity (b) Low selectivity

Figure 4.7: MapReduce Scan vs. Index Scan.

By setting the value of $y$ in the query predicate, we can define the selectivity of the query (denoted as $s$ in Figure 4.7). Two sets of experiments were conducted to evaluate the query processing time of generic MapReduce and *ScalaGiST*-integrated MapReduce under different selectivity settings.

In the first experiment, we study performance characteristics of the systems when executing highly selective queries – the selectivity is set to 0.4% and 4%. The results plotted in Figure 4.7a confirm the effectiveness of *ScalaGiST* in supporting query processing over large scale data. Generic MapReduce program implements this query by performing a full parallel scan on the entire data chunks of `Orders` table, and hence its performance is not affected much by the query selectivity. In contrast, *ScalaGiST* helps to achieve a better performance by first querying the distributed index to identify the qualified data, then launching map tasks only on the data chunks hosting the target data. In this way, *ScalaGiST* avoids yielding unnecessary overhead on full table scan over irrelevant data. The overhead of traversal within internal nodes of *ScalaGiST* index tree is negligible, because most internal nodes are cached locally.

Based on the insights of our cost model, *ScalaGiST*-integrated MapReduce underperforms in the case where index search cannot prune out enough data chunk. Hence, in the second experiment, we test the two approaches with non-selective queries to see the crossover point. With the decrease in selectivity, query latency increases drastically when *ScalaGiST* is used. As depicted in Figure 4.7b, a selectivity of 30% is low enough for *ScalaGiST*-integrated MapReduce to perform worse than the generic MapReduce. Low query selectivity results in a larger result set that may span across more data chunks, and hence more map tasks have to be launched. Further, larger result set incurs abundant random I/Os to retrieve the data records because of the non-clustered secondary index tested in this experiment. These two factors have significant impacts on the performance of *ScalaGiST*-integrated MapReduce. Note that in this experiment we disabled the data access optimizer so that the execution engine would stick to either generic MapReduce plan (full table scan) or *ScalaGiST* plan (index scan).

### 4.5.4 Multi-Dimensional Index Performance

In this section, we demonstrate the effectiveness of *ScalaGiST* in terms of its support for multi-dimensional data. We first compare the performances of *ScalaGiST* and three systems (namely, Data Mapping [24], SpatialHadoop [38] and RT-CAN [99]) on a 2-dimensional dataset. We then evaluate the performance of *ScalaGiST*'s M-tree implementation in higher dimensional (up to 10) settings.

For the first set of experiments, we construct a 2-dimensional table T with schema $T(a_1, a_2, p)$ where each attribute $a_i$ uniformly generated from the domain of $10^9$ integer values, and attribute $p$ is a payload of 1 KB string data. The table is populated with 10 million to 100 million records, thus the size of the table varies from 10 GB to 100 GB. R-Tree indexes are built on $(a_1, a_2)$ pair using the three systems, respectively. The number of index servers in the system is fixed to be 20.

We run both range queries and $k$-NN queries to evaluate the systems' performance. Specifically, range queries are run on $(a_1, a_2)$ against the indexes with the following template:

```
Q2:SELECT p FROM T
   WHERE a1_l ≤ a1 ≤ a1_u and a2_l ≤ a2 ≤ a2_u
```

We define the selectivity as the percentage of searched space. By adjusting the lower bounds and upper bounds for both $a_1$ and $a_2$, we are able to control the query selectivity, which is set to 0.4% in this experiment. $K$-NN queries are processed via a set of range queries. For $k$-NN queries, $k$ is set to be 16 in the experiments. The results are presented in the following sections.

### Generalized Search Tree vs. Data Mapping

For multi-dimensional domains, data mapping approach partitions the original space into sub-spaces by different dimension iteratively, then links the partitions with adjacent identifiers to form the Z-ordering [67], which is a 1-dimensional representation of the original multi-dimensional domain, and thus range query in higher dimensional spaces could be transformed into querying intervals along the Z-ordering.

As can be seen in Figure 4.8, *ScalaGiST* gains a better performance over the "Data Mapping" approach. As the data size increases, the latency of queries with "Data Mapping" grows proportionally, while remaining stable with only a slight increment for *ScalaGiST*.

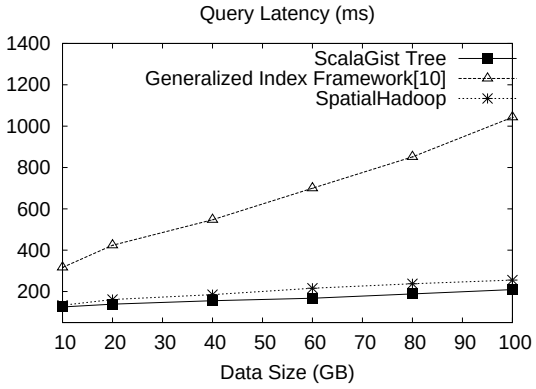



Figure 4.8: Range Query Performance.

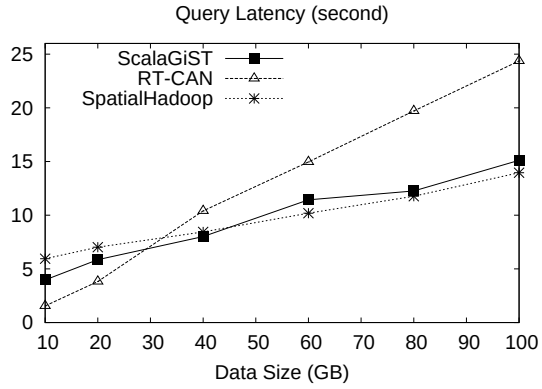



Figure 4.9: $k$-NN Query Performance.

"Data Mapping" handles multi-dimensional query by mapping both the data and query into a 1-dimension space and disseminating the query to the underlying overlay network. Such dimension reduction provides a unified key space for different types of overlays at the price of false-positive candidates in the result set that add extra overhead to process.

It is because of the fact that two adjacent points in the multi-dimensional space might be mapped to non-adjacent partitions in the single dimensional space. As a result, there could be considerable amount of false positive points along the Z-ordering of the query range, incurring undesirable I/Os and computation overheads. On the contrary, by customizing the abstracted key class and tree methods *ScalaGiST* can avoid such problem since it resembles the traditional R-tree's structure and search algorithms on multi-dimensional spaces, which makes its query performance more efficient compared to the 'Data Mapping' approach for processing much less false-positive candidates.

**Generalized Search Tree vs. SpatialHadoop**

SpatialHadoop extents Hadoop to support spatial index operations. It organizes the spatial index in a layered structure, namely global partition index and local indexes. SpatialHadoop provides a layer of abstraction upon MapReduce by implementing its own multi-dimensional index operators, such as range operator and $k$-NN operator in order to facilitate multi-dimensional queries. In comparison, *ScalaGiST* adopts a different index processing mechanism in which the index operations are performed by index workers, and DFS data requests are handled by MapReduce(e.g. Hadoop) runtime.

Figure 4.8 and 4.9 compare the performance of *ScalaGiST* and SpatialHadoop in terms of range queries and $k$-NN queries. From the results, we observe a close performance for the two systems. In both systems, index operations are mostly done in memory, while the local index (in SpatialHadoop) or index workers (in *ScalaGiST*) take care of DFS I/Os. The slight difference comes from different implementation of index operators, in particular, the different instantiation cost for the two systems.

However, it is a promising result for *ScalaGiST* in that the performance of its generalized framework is comparable to that of SpatialHadoop's specially built and tuned index.

**Generalized Search Tree vs. RT-CAN**

RT-CAN is a multi-dimensional indexing framework for cloud environments. RT-CAN organizes the servers into an overlay based on an extended CAN routing protocol, and utilizes R-tree based index scheme at each server to support multi-dimensional query.

In the experiments, data are pre-partitioned into 5,000 grids and disseminated to the servers of RT-CAN. A local R-tree is built for the grids at each server with a page size of 4 KB.

The results plotted in Figure 4.9 confirm the extensibility of *ScalaGiST* to support complex multi-dimensional query. In particular, the overall performance of *ScalaGiST*

is better than RT-CAN in terms of supporting $k$-NN query. Even though *ScalaGiST* incurs higher latency for $k$-NN queries at small data size, we observe a better performance of *ScalaGiST* as the data size increases. When data size is small, the start-up time for MapReduce tasks has more significant impact on the query efficiency with *ScalaGiST*, while RT-CAN does not suffer such overhead due to its different (peer-to-peer) processing model. However, at lager scales, the iterative overlay lookup and local R-tree search yield relatively high I/Os and computational cost.

**Scalagist in Multi-dimensional Metric Space**

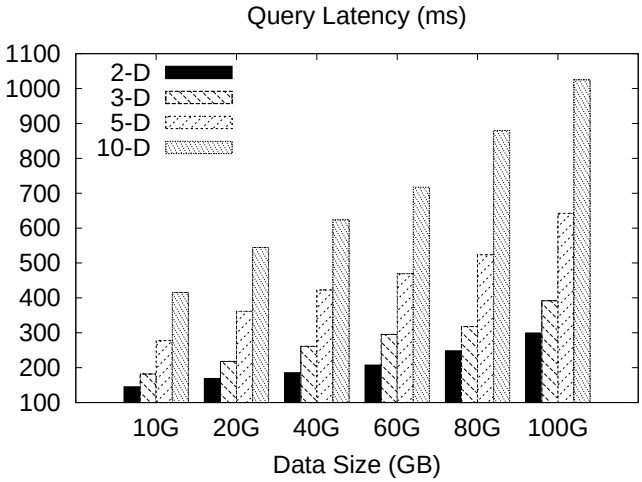



Figure 4.10: Effect of Dimensionality.

Using *ScalaGiST*, we can build new distributed indexes with ease by overriding the interface functions. In this experiment, we demonstrate the *ScalaGiST*'s M-tree implementation which is employed to index multi-dimensional data in a metric space. We use synthetic Random-Cluster (R-Cluster) data sets to evaluate the performance of *ScalaGiST*on varied dimensionalities (up to 10). The R-Cluster data sets consist of records with a tuple ID and $d$-dimensional coordinates. The IDs are 4-byte integers and the coordinates are 4-byte floating-point types. Distance between two records can be calculated using any user defined metric distance function. In this experiment, we adopt the $L_\infty$ metric, i.e. $L_\infty(O_x, O_y) = max_{j=1}^{Dim}\{|O_x[j] - O_y[j]|\}$. System settings remain the same as in the previous experiments. A range query with selectivity of 0.4% in the $d$-dimensional space is run on the indexed data. Under each scale, we report the effect of varying dimensionality from 2 to 10 in Figure 4.10.

As depicted in Figure 4.10, with the increase of dimensionality and data size, the average time of running a range query also increases. This trend coincides with the typical

performance of M-tree in the stand-alone setting. In addition, we are able to observe a good scalability both in terms of dimensionality and data size. The time-to-dimensionality and time-to-size pairs both scale nearly linearly. These results verify the functionality of *ScalaGiST* in supporting multi-dimensional data.

### 4.5.5 Multiple Indexes Performance

As mentioned in Section 4.3.3, one of the most distinguished features of *ScalaGiST* is its capability of supporting various types of indexes. In this experiment, we demonstrate this merit by incorporating multiple indexes in a single query. As most of the real data are business sensitive and are not publicly available, we synthetically construct our data to have multiple dimensional characteristics. Our purpose is to use this simple but straightforward example to exhibit how *ScalaGiST* benefits query using multiple indexes.

The schema $T\{orders, (a_1, a_2)\}$ is composed of the `Orders` table from TPC-H dataset, and the two-dimensional attribute $(a_1, a_2)$ we generate in the last experiment. The table is sorted by $totalprice$ column in `Orders` table. Given its characteristic, a B$^+$-tree can be built on the $totalprice$ column, and the 2-dimensional column $(a_1, a_2)$ can be indexed by an R-tree.

Note that when building multiple indexes on one table, factors such as the clustering of data, the choice of primary index, etc., would all have substantial influence on the performance. *ScalaGiST* is designed to provide flexible functionality and APIs, and leaves other decisions to the user.

The query used in this experiment is a simple extension of Q1 with a range search on the 2-dimensional column:

```
Q3:SELECT custkey, count(orderkey)
   FROM Orders
   WHERE totalprice ≥ y and totalprice ≤ y + 100
   and a1_l ≤ a1 ≤ a1_u and a2_l ≤ a2 ≤ a2_u
```

Q3 is evaluated in three execution modes. The first is *ScalaGiST* multiple indexes mode, who has both B$^+$-tree and R-tree built on the two columns respectively. The second mode only builds R-tree on the 2-dimensional column. And in the third mode we use SpatialHadoop. The size of data varies approximately from 15G to 150G (`Orders` table plus additional column). The results are plotted in Figure 4.11.

To process Q3, two columns of the table are touched. The indexed column is searched via index workers first. With the knowledge of index search result, MapReduce jobs are launched on the chunks hosting the interested data. For the column without index, a

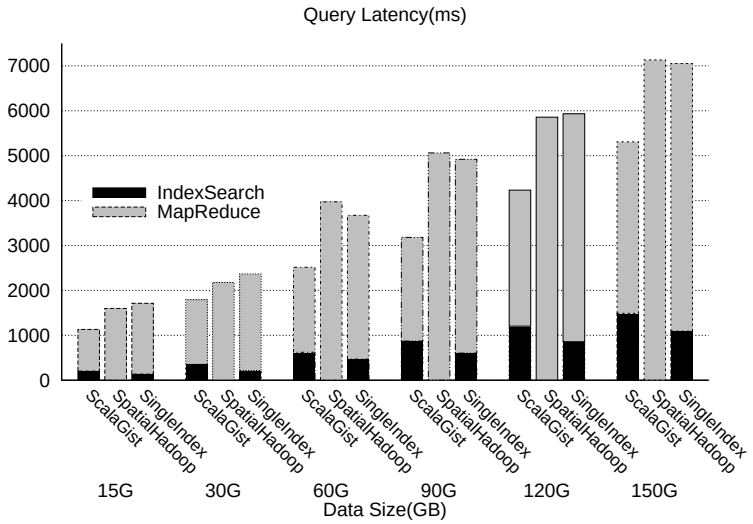



Figure 4.11: Multiple Index Performance.

MapReduce scan must be launched. Specifically, in the R-tree only case, index search on R-tree returns a super set of the accurate selection result. Then a "partial" MapReduce job scans through the chunks included in the search result to test against the $totalprice$ column before generating the final results. For SpatialHadoop, two set of MapReduce jobs are launched. The first MapReduce job runs range search using original SpatialHadoop function. The second set of MapReduce jobs are used to scan the whole table for selective condition, and merge the scan result with result of range search.

In Figure 4.11, the processing time is broken down to highlight the index search phase and the MapReduce phase. As shown, the overall execution time of *ScalaGiST* is significantly reduced comparing to those of SpatialHadoop and single index. *ScalaGiST* has longer index processing time, since the runtime need to wait until all index workers complete the search and merge the results. However, with the benefit of more accurate index search result, the subsequent MapReduce job in *ScalaGiST* is able to avoid launching redundant mappers, and enjoys better performance.

## 4.6 Conclusion

In this work, we bring the previous work on bitmap indexing in MapReduce one step further, and present *ScalaGiST* – scalable generalized search tree – which provides the much desired extensibility in terms of data and query type. It supports multiple types of indexes, and can be dynamically deployed on large clusters while resilient to machine failures. We have implemented *ScalaGiST* and demonstrated that it can be easily instantiated as scalable $B^+$-tree and R-tree like indexes for dynamic cluster environments. More importantly, its seamless integration with Hadoop platform, coupled with a cost-based data access optimizer, provide promising opportunities for significant performance improvement on query processing in MapReduce-based systems. Our experiments on *ScalaGiST*'s performance with respect to multiple types of indexes confirmed the effectiveness and efficiency of our proposed indexing mechanism.

In summary, the contributions of this work are as follows.

- We introduce *ScalaGiST* – scalable generalized search tree for dynamic cluster environments such as the Cloud. It provides extensibility in terms of data and query types for supporting unconventional queries (e.g., multi-dimensional range and $k$-NN queries), and more importantly, can be dynamically deployed on large clusters for handling big users and data.

- We present an approach to integrating *ScalaGiST* seamlessly with Hadoop platform, coupled with a cost-based data access optimizer for improving the performance of MapReduce execution.

- We have built *ScalaGiST* and conducted an extensive performance study on an in-house cluster. We compare the R-tree and $B^+$-tree-like indexes implemented using *ScalaGiST* with recent indexes such as Data Mapping [24], RT-CAN[99] and SpatialHadoop[38]. The results confirm its efficiency and scalability in terms of write and read performance, as well as effective support of exact match, range, and similarity queries.

# Part II

# Parallelizing the RDBMSs

# CHAPTER 5

# Adaptive Massive Parallel Processing

## 5.1 Motivation

The production environment for analytical data management applications is rapidly changing. Many enterprises are shifting away from deploying their analytical databases on high-end proprietary machines, and moving towards cheaper, lower-end, commodity hardware, typically arranged in a shared-nothing MPP (Massively Parallel Processing) architecture, which is widely believed to scale the best [63]. However, there are very few known parallel database deployments consisting of more than one hundred nodes. There are a variety of reasons why parallel databases generally do not scale well into the hundreds of nodes. First, failures become increasingly common as one adds more nodes to a system, yet parallel databases tend to be designed with the assumption that failures are a rare event. Second, parallel databases generally assume a homogeneous array of machines, yet it is nearly impossible to achieve pure homogeneity at scale. Third, until recently, there have only been a handful of applications that required deployment on more than a few dozen of nodes for reasonable performance, so parallel databases have not been tested at larger scales, and unforseen engineering hurdles await.

The widespread adoption of MapReduce for MPP systems unfolds discussions and attempts to extend MapReduce to handle data analytical workloads at unconventional scale instead of using parallel databases. Much of the performance issues of MapReduce and its derivative systems can be attributed to the fact that they were not initially designed to be used as complete, end-to-end data analysis systems over structured data. Their target use cases include scanning through a large set of documents produced from a web crawler and producing a web index over them [30]. In these applications, the input data is

often unstructured and a brute force scan strategy over all of the data is usually optimal. MapReduce then helps automate the parallelization of the data scanning and application of user defined functions as the data is being scanned.

For more traditional data analysis workloads that work with data produced from business operational data stores, the data is far more structured. Furthermore, the queries tend to access only a subset of this data (e.g. breakdown the profit of stores *located in the Northeast*). Using data structures that help accelerates access to needed entities (such as indexes) and dimensions (such as column-stores), and data structures that precalculate common requests (such as materialized views) often outperform a brute-force scan execution strategy. As pointed out by Dewitt and Stonebreaker [34], MapReduce lacks many of the features that have been proven invaluable for structured data analysis workloads, and its immediate gratification paradigm precludes some of the long term benefits of first modeling and loading data before processing. The potential performance drawback of MapReduce has been reported on the basis of experiments on two benchmarks [76] – TPC-H and a customized benchmark tailored for search engines.

Therefore, it is now clear that neither MapReduce-like software, nor parallel databases are ideal solutions for data analysis in the Cloud. Hence, a **hybrid** solution that combines the fault tolerance, heterogeneous cluster, and ease of use out-of-the-box capabilities of MapReduce with the efficiency, performance, and tool plugability of shared-nothing parallel database systems could have a significant impact on the Cloud database market.

There has been some recent work on bringing together ideas from MapReduce and database systems, however, these works focus on language and interface issues. The Pig project at Yahoo! [72] and the SCOPE project at Microsoft [18] aim to integrate declarative query constructs from the database community into MapReduce-like software to allow greater data independence, code reusablity, and automatic query optimization. Greenplum and Aster Data have added the ability to write MapReduce functions (instead of, or in addition to, SQL) over data stored in their parallel database products. Although these four projects are without question an important step in the direction of a hybrid solution, there remains a need for a hybrid solution at the systems level in addition to at the language level.

### 5.1.1 The BestPeer++ Lesson

BestPeer++ [23] is a cloud enabled data sharing platform designed for corporate network applications. BestPeer++ integrates cloud computing, database, and peer-to-peer (P2P) technologies, and is a practical, flexible and efficient solution for corporate network applications. However, with the majority of the reported experiments having dominant advantage, the performance of BestPeer++ was surpassed by that of HadoopDB in the

face of complex query which involves multiple joins and aggregations (TPC-H Query 5).

Analysis of query execution in BestPeer++ shows that the performance degradation of BestPeer++ in TPC-H Query 5 is mainly caused by the way that BestPeer++ implements join. In the shared-nothing architecture of BestPeer+, tables are distributed among local RDBMSs, and there is few global knowledge of data maintained. During a join, local RDBMSs have to perform rounds of replicated joins which inevitably leads to undesirable I/O and computation overhead. In addition, the final round of join has to be executed in a single peer in order to gather the partial results from distributed peers. Such way of join execution incurs redundant I/O and computation cost, and the single peer becomes the bottleneck if the final result table is exceedingly large. On the contrary, by encapsulating a DFS abstraction on top of the distributed RDBMSs, HadoopDB is able to post local selection results on the globally mounted DFS, and performs all the joins in a parallel fashion using MapReduce.

Nevertheless, using HadoopDB is not always beneficial as reported in other experiments in [23]. This disparity in performance gives rise to an interesting and practical question: is it possible to combine the advantages in parallel RDBMS and MapReduce to supplement the deficiencies of each other?

In this work, we exploit the feasibility of building a hybrid system that takes the best features from both MapReduce and shared-nothing parallel RDBMSs, and propose an adaptive query processing engine that incorporates the query execution of traditional parallel databases and MapReduce. In particular, we identify the strategic differences between DBMS query execution and MapReduce, and model the query efficiency for both execution plans. Using the cost model, we devise a hybrid execution engine that adaptively generates the most cost effective plan for queries. The prototype we build approaches parallel databases in performance and efficiency, yet still yields the scalability, fault tolerance, and flexibility of MapReduce-based systems.

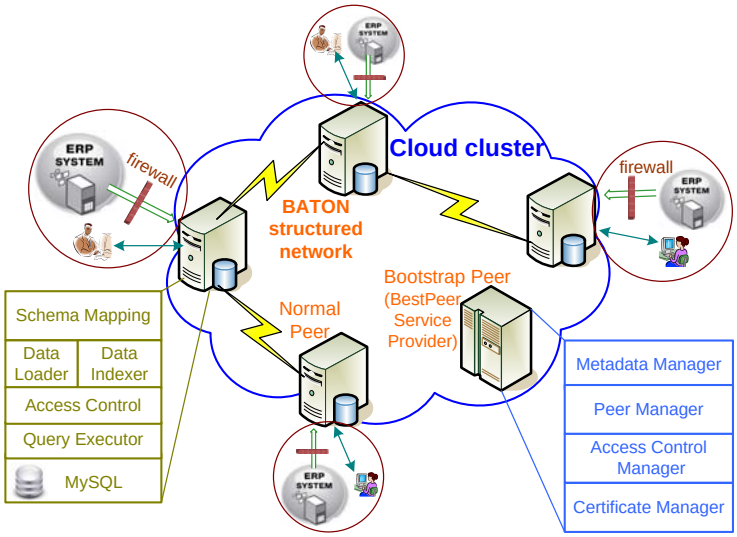



Figure 5.1: The BestPeer++ network deployed on Amazon Cloud offering

## 5.2 The BestPeer++ Core

The BestPeer++ core contains all platform-independent logic, including query processing and P2P overlay. It runs on top of the Cloud adapter and consists of two software components: *bootstrap peer* and *normal peer*. A BestPeer++ network can only have a single bootstrap peer instance which is always launched and maintained by the BestPeer++ service provider, and a set of normal peer instances. The architecture is depicted in Figure 5.1. This section briefly describes the functionalities of these two kinds of peer. Individual components and data flows inside these peers are presented in the subsequent sections.

The bootstrap peer is the entry point of the whole network. It has several responsibilities. First, the bootstrap peer serves for various administration purposes, including monitoring and managing normal peers and also scheduling various network management events. Second, the bootstrap peer acts as a central repository for meta data of corporate network applications, including shared global schema, participant normal peer list, and role definitions. In addition, BestPeer++ employs the standard PKI encryption scheme to encrypt/decrypt data transmitted between normal peers in order to further increase the security of the system. Thus, the bootstrap peer also acts as a Certificate Authority (CA) center for certifying the identities of normal peers.

Normal peers are the BestPeer++ instances launched by businesses. Each normal peer is owned and managed by an individual business and serves the data retrieval requests issued by the users of the owning business. To meet the high throughput requirement, BestPeer++ does not rely on a centralized server to locate which normal peer hold which tables. Instead, the normal peers are organized as a balanced tree peer-to-peer overlay based on BATON [51]. The query processing is, thus, performed in entirely a distributed

manner. Details of query processing is presented in Section 5.3.

### 5.2.1 Bootstrap Peer

The bootstrap peer is run by the BestPeer++ service provider, and its main functionality is to manage the BestPeer++ network. This section presents how bootstrap peer performs various administrative tasks.

#### Managing Normal Peer Join/Departure

Each normal peer intends to join an existing corporate network must first connect to the bootstrap peer. If the join request is permitted by the service provider, the bootstrap peer will put the newly joined peer into the peer list of the corporate network. At the same time, the joined peer will receive the corporate network information including the current participants, global schema, role definitions, and an issued certificate. When a normal peer needs to leave the network, it also notifies the bootstrap peer first. The bootstrap peer will move the departure peer to the black list and mark the certificate of the departing peer invalid. The bootstrap peer will the reclaim all resources allocated to the departing peer and finally remove the departing peer from the peer list.

#### Auto Fail-over and Auto-Scaling

In addition to managing peer join and peer departure, the bootstrap peer spends most of its running-time on monitoring the healthy of normal peers and scheduling fail-over and auto-scaling events. Algorithm 8 shows how the daemon service of the bootstrap works.

The bootstrap periodically collects performance metrics of each normal peer (line 2). If some peers are malfunctioned or crashed, the bootstrap peer will trigger an automatic fail-over event for each failed normal peer (line 6-10). The automatic fail-over is performed by first launching a new instance from cloud. Then, the bootstrap peer asks the newly launched instance to perform database recovery from the latest database backup stored in Amazon EBS. Finally, the failed peer is put into the blacklist. Similarly, if any normal peer is overloaded (e.g., CPU is over-utilized or free storage space is low), the bootstrap peer triggers an auto-scaling event (line 12-17) to either upgrade the normal peer to a larger instance or allocate more storage spaces.

At the end of each maintenance epoch, the bootstrap releases the resources in the blacklist (line 18) and notifies the changes to all participants (line 20).

As discussed above, cloud services provide the required reliability of a single node, i.e., its data can be safely recovered in cases of crashes within a recovery time constraint

**Algorithm 8:** BootStrapDaemon()

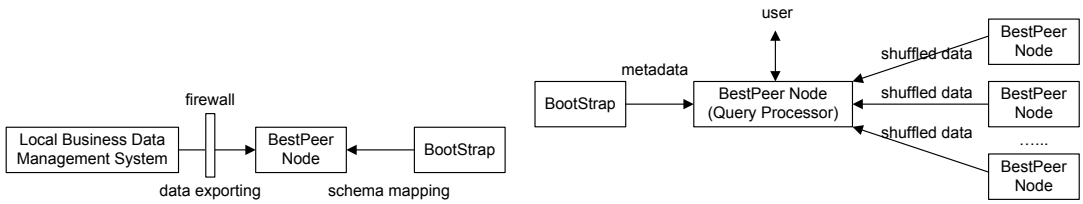
```
1:  while true do
2:      Status S = invokeCloudWatch()
3:      ArrayList peerList = BootStrap.getAllPeer()
4:      ArrayList newPeer= new ArrayList()
5:      for i=0 to peerList.size() do
6:          if peerList.get(i).fails() then
7:              Peer peer = new Peer()
8:              peer.loadMySQLBackUpFromRDS(peerList.get(i))
9:              newPeer.add(peer)
10:             BootStrap.setBlackList(peerList.get(i))
11:         else
12:             if peerList.get(i).overloaded() then
13:                 Peer peer = new Peer()
14:                 peer.upScale(peerList.get(i))
15:                 peer.clone(peerList.get(i).getDB())
16:                 BootStrap.setBlackList(peerList.get(i))
17:                 newPeer.add(peer)
18:     BootStrap.removeAllPeersInBlackList()
19:     BootStrap.addAllNewPeer(newPeer)
20:     BootStrap.broadcastNetworkStatus()
21:     sleep T seconds
```

guaranteed by the service level agreements (SLAs) offered by the cloud services. In a data sharing platform like BestPeer++, enforcing system's consistency guarantee is a crucial but difficult task. An important issue is the consistency of the whole system when there are node failures, more specifically how queries can be executed in these situations. Business applications rely on accurate summarization of data, and thus may suffer from any form of data inconsistency. Therefore, the widely used eventual consistency model [13] or other weakened consistency models do not fit in our case. In BestPeer++, we opt to enforce strong consistency by guaranteeing that all necessary data in a business scope is online at query time. When a node crashes, all affected queries need to be blocked until the auto fail-over process is completed. We are able to provide correctness and consistency guarantee in this way at the expense of some latency. However, given that the recovery time complies with SLA's constraint, this latency is restrained within an acceptable range.

### 5.2.2 Normal Peer

The normal peer software consists of five components: schema mapping, data loader, data indexer, access control, and query executor. We present the first four components in this section. Query processing in BestPeer++ will be presented in the next section.

As shown in Figure 5.2, there are two data flows inside the normal peer: an offline data flow and an online data flow. In the offline data flow, the data are extracted periodically by a data loader from the business production system to the normal peer instance. In partic-




(a) Offline Data Flow (b) Online Data Flow

Figure 5.2: Data Flow in BestPeer++

ular, the data loader extracts the data from the business production system, transforms the data format from its local schema to the shared global schema of the corporate network according to the schema mapping, and finally stores the results in the MySQL databases hosted in the normal peer.

In the online data flow, user queries are submitted to the normal peer and then processed by the query processor. The query processor performs user queries using a fetch and process strategy. The query processor first parses the query and then employs the BATON search algorithm to identify the peers that hold the data related to the query. Then, the query executor employs a pay-as-you-go query processing strategy, which will be described in Section 5.3 in detail, to process those data and return the results to the user.

### Schema Mapping

Schema mapping [12] is a component that defines the mapping between the local schema of each production system and the global shared schema employed by the corporate network. Currently, BestPeer++ only supports relational schema mapping, namely both local schema and the global schema are relational. The mapping consists of metadata mappings (i.e., mapping local table definitions to global table definitions) and value mappings (i.e., mapping local terms to global terms). Besides schema-level mapping, BestPeer++ can also support instance-level mapping [81], which complements the mapping process when there is not sufficient schema information. In general, the schema mapping process requires human to be involved and is rather time consuming. However, it only needs to perform once. Furthermore, BestPeer++ adopts templates to facilitate the mapping process. Specifically, for each popular production system (i.e., SAP or PeopleSoft), we provide a mapping template which defines the transformation of local schemas of those systems to a global schema. What the business only needs is to modify the mapping template to meet its own needs. We found that this mapping template approach works well in practice and significantly reduces the service setup efforts.

| | |
|---|---|
| `join(P)` | Join the network |
| `leave(P)` | Leave the network |
| `put(k, v)` | Insert a key-value pair into the network |
| `remove(k, v)` | Delete the value with the key |
| `get(k)` | Retrieve the value with the single key |
| `get(begin, end)` | Retrieve values with the key range |

Table 5.1: BATON Interface

### Data Loader

Data Loader is a component that extracts data from production systems to normal peer instances according to the result of schema mapping. While the process of extracting and transforming data is straightforward, the main challenge comes from maintaining consistency between raw data stored in the production systems and extracted data stored in the normal peer instance (and subsequently data indices created from these extracted data) while the raw data being updated inside the production systems.

We solve the consistency problem by the following approach. When the data loader first extracts data from the production system, besides storing the results in the normal peer instance, the data loader also creates a snapshot of the newly inserted data [1]. After that, at interval times, the data loader re-extracts data from the production system to create a new snapshot. This snapshot is then compared to the previously stored one to detect data changes. Finally, the changes are used to update the MySQL database hosted in the normal peer.

Given two consecutive data snapshots, we employ a similar algorithm as the one proposed in [41]. In our algorithm, the system first fingerprints every tuple of the tables in the two snapshots to a unique integer. We use 32Bits Rabin fingerprinting method [80]. Then, each table is sorted by the fingerprint values. Finally, the algorithm executes the sort merge algorithm on the tables in both snapshots. The resultant table after sorting reveals changes in the data.

### Data Indexer

In the BestPeer++, the data are stored in the local MySQL database hosted by each normal peer. Thus, to process a query, we need to locate which normal peers host the tables involved in the query. For example, to process a simple query like `select R.a from R where R.b=x`, we need to know the location of the peers store tuples belonging to the global table `R`.

We adopt the peer-to-peer technology to solve the data locating problem and only send

[1] The snapshot is also stored in the normal peer instance but in a separate database.

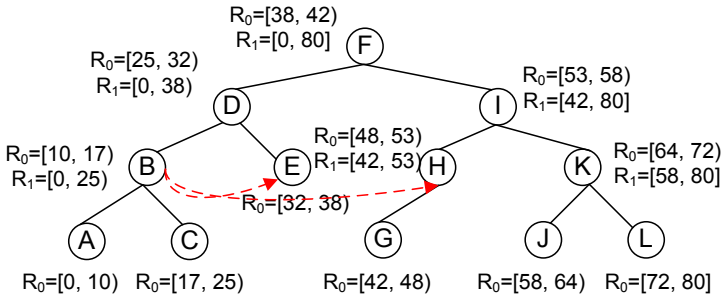



Figure 5.3: BATON Overlay

queries to normal peers which host related data. In particular, we employ BATON [51], a balanced binary tree overlay protocol to organize all normal peers. Figure 5.3 shows the structure of BATON. Given a value domain [L, U], each node in BATON is responsible for two ranges. The first range, $R_0$, is the sub-domain maintained by the node. The second range, $R_1$, is the domain of the subtree rooted at the node. For example, $R_0$ and $R_1$ are set to $[25, 32)$ and $[0, 38)$ for node $D$, respectively. For a key $k$, there is one unique peer $p$ responsible for $k$ and $k$ is contained by $p.R_0$. For a range $[l, u]$, there is also one unique peer $\bar{p}$ for it and 1) $[l, u]$ is a sub-range of $\bar{p}.R_1$; and 2) if $[l, u]$ is contained by $\widehat{p}.R_1$, $\widehat{p}$ must be $\bar{p}$'s ancestor node.

If we traverse the tree via in-order, we can access the values in consecutive domains. In BATON, each node maintains $\log_2 N$ routing neighbors in the same level, which are used to facilitate the search process in this index structure. To achieve a balanced structure, BATON employs two flexible load balancing schemes [51]. A node can balance its load with adjacent nodes when there exists under-loaded ones. However, in the case that there is no adjacent node available for load balancing, BATON performs a global adjustment by moving a non-adjacent leaf node from its original position to the overloaded region to share load. Since BATON organizes nodes as a balanced tree, such a scheme could incur network restructuring. However, this amortized cost is just $O(\log_2 N)$ per insertion or deletion [49], which is negligible. For details about BATON, readers are referred to [49, 51].

In BestPeer++, the interface of BATON is abstracted as Table 5.1. We provide three ways to locate data required for query evaluation: table index, column index, and range index. Each of them is designed for a separate purpose.

**Table Index.** Given a table name, a table index is designed for searching the normal peers hosting the related table. A table index is of the form $I_T(key, value)$ where the $key$ is the table name and $value$ is a list of normal peers which store data of the table.

**Column Index.** Column index is a supplementary index to table index. This index type is designed to support queries over columns. A column index $I_C(key, value)$ includes a key, which is the column name in the global shared schema, and a value, which

| Type | Key | Indexed Value |
|---|---|---|
| Table Index | Table Name | A normal peer list |
| Column Index | Column Name | A list of peer-table pairs |
| Range Index | Table Name | A list of column-range pairs |

Table 5.2: Index Format Summaries

consists of the identifier of the owner normal peer and a list of tables containing the column in the peer.

**Range Index.** Range indices are built on specific columns of the global shared tables. One range index is built for one column. A range index is of the form $I_D(key, value)$ where $key$ is the table name and the value is a list. Each item in the list consists of the column name denoting which column the range index is built on, a min-max value which encodes the minimum and maximum value in the column being indexed, and the normal peer which stores the table.

Table 5.2 summarizes the index formats in BestPeer++. In query processing, the priorities of indices are (Range Index>Column Index>Table Index). We will use the more accurate index whenever possible. Consider $Q1$ of the TPC-H benchmark:

```
SELECT l_orderkey, l_receiptdate
   FROM LineItem
   WHERE l_shipdate > Date(1998-11-05) AND
         l_commitedate > Date(1998-09-29)
```

If the range index has been built for `l_shipdate`, the query processor can know which peers have the tuples with `l_shipdate` > `Date(1998-11-05)`. Otherwise, if column index is available, the query processor only knows which peers have the `LineItem` table and their `l_shipdate` columns have valid values.[2] In the worst case, when only table index is available, the query processor needs to communicate with every peer that has part of the lineitem table.

Since machine failures in cloud environment are not uncommon, BestPeer++ employs replication of index data in the BATON structure to ensure the correct retrieval of index data in the presence of failures. Specifically, we use the two-tier partial replication strategy to provide both data availability and load balancing, as proposed in our recent study [95]. BestPeer++ couples its inherent load balancing scheme with the one proposed in [95] to achieve a better performance. The complete method for system recovery from various types of node failures is also studied in this work.

[2] In multi-tenant scenario, even the companies share the same schema, they may have different set of columns.

**Distributed Access Control**

The access to multi-businesses data shared in a corporate network needs to be controlled properly. The challenge is for BestPeer++ to provide a flexible and easy-to-use access control scheme for the whole system; at the same time, it should enable each business to decide the users that can access its shared data in the inherent distributed environment of corporate networks. BestPeer++ develops a distributed role-based access control scheme. The basic idea is to use roles as templates to capture common data access privileges and allow businesses to override these privileges to meet their specific needs.

**Definition 5.1. Access Role**
*The access role is defined as* $Role = \{(c_i, p_j, \delta) | c_i \in S_c \wedge p_j \in S_p \wedge \delta \in S_v\}$*, where* $S_c$ *is the set of columns,* $S_p$ *is the set of privileges and* $S_v$ *is the range conditions.*

For example, suppose we have created a role $Role_{sales}$={*(lineitem.extendedprice, read∧write, [0, 100]), (lineitem.shipdate, read, null)* } and a user is assigned the role $Role_{sales}$. He can only access two columns. For the shipdate column, he can access all values, but cannot update them. For the extendedprice column, the user can read and modify the values in the range of $[0, 100]$.

When setting up a new corporate network, the service provider defines a standard set of roles. The local administrator at each normal peer can assign the new user with an existing role if the access privilege of that role is applicable to the new user. If none of the existing roles satisfies the new user, the local administrator can create new roles by three operators: $\vdash$, $-$ and $+$.

- $Role_i \vdash Role_j$: $Role_j$ inherits all privileges defined by $Role_i$.
- $Role_j = Role_i - (c_i, p_j, \delta)$: $Role_j$ gets all privileges of $Role_i$ with the exception of $(c_i, p_j, \delta)$.
- $Role_j = Role_i + (c_i, p_j, \delta)$: $Role_j$ gets all privileges of $Role_i$ and a new access rule $(c_i, p_j, \delta)$.

The roles are maintained locally and used in the query processing to rewrite the queries. Specifically, given a query $Q$ submitted by user $u$, the query processor will send the data retrieval request to the involved peers. The peer, upon receiving the request, will transform it based on $u$'s access role. The data that cannot be accessed by $u$ will not be returned. For example, if a user assigned to $Role_{sale}$ tries to retrieve all tuples from `lineitem`, the peer will only return values from two columns: `extendedprice` and `shipdate`. For `extendedprice`, only values in $[0, 100]$ are shown, the rest are marked as "NULL".

Note that BestPeer++ does not collect the information of existing users in the collaborating ERP databases, since it will lead to potential security issues. Instead, the user management module of BestPeer++ provides interfaces for the local administrator at each participating organization to create new accounts for users who desire to access BestPeer++ service. The information of the users created at one peer is forwarded to the bootstrap peer and then broadcasted to other normal peers also. In this manner, each normal peer will eventually have enough user information of the whole network, and therefore the local administrator at this peer can easily define the role-based access control for any user.

## 5.3 Pay-As-You-Go Query Processing

BestPeer++ provides two services for the participants: the storage service and search service, both of which are charged in a pay-as-you-go model. This section presents the pay-as-you-go query processing module which offers an optimal performance within the user's budget. We begin with the presentation of histogram generation, a building block for estimating intermediate result size. Then, we present the query processing strategy.

Before discussing the details of query processing, we first define the semantics of query processing in the BestPeer++. After data are exported from the local business system into a BestPeer++ instance, we apply the schema mapping rules to transform them into the predefined formats. In this way, given a table $T$ in the global schema, each peer essentially maintains a horizontal partition of it. The semantics of queries is defined as

**Definition 5.2. Query Semantic**
*For a query $Q$ submitted at time $t$, let $\mathcal{T}$ denote the tables involved in $Q$. The result of $Q$ is computed on $\bigcup_{\forall T_i \in \mathcal{T}} S_t(T_i)$, where $S_t(T_i)$ is the snapshot of table $T_i$ at time $t$.*

When a peer receives a query, it compares the timestamp ($t'$) of its database with the query's timestamp ($t$). If $t' \leq t$, the peer processes the query and returns the result. Otherwise, it rejects the query and notifies the query processor, which will terminate the query and resubmit it.

### 5.3.1 The Histogram

In BestPeer++, histograms are used to maintain the statistics of column values for query optimization. Since attributes in a relation are correlated, single-dimensional histograms are not sufficient for maintaining the statistics. Instead, multi-dimensional histograms are employed. BestPeer++ adopts MHIST [78] to build multi-dimensional histograms adaptively. Each normal peer invokes MHIST to iteratively split the attribute

which is most valuable for building histograms until enough histogram buckets are generated. Then, the buckets (multi-dimensional hypercube) are mapped into one dimensional ranges using iDistance [50] and we index the buckets in BATON based on their ranges.

Once the histograms have been established, we can estimate the size of a relation and the result size of joining two relations as follows.

**Estimation of a Relation Size.** Given a relation $R$ and its corresponding histogram $H(R)$, $ES(R) = \sum_i H(R)_i$, where $H(R)_i$ denotes the value of the $i^{th}$ bucket in $H(R)$.

**Estimation of Pairwise Joining Result Size.** Given two relations $R_x$, $R_y$, their corresponding histograms $H(R_x)$, $H(R_y)$ and a query $q = \sigma_p(R_x \bowtie_{R_x.a=R_y.b} R_y)$, where $p = R_x.a_1 \wedge \cdots \wedge R_x.a_{n-1} \wedge R_y.b_1 \wedge \cdots \wedge R_y.b_{n-1}$, to estimate the joining result size of a query, we first estimate the number of data in each histogram belonging to the queried region $(Q_R)$ defined by the predicate $p$ as follows.

$$EC(H(R_x)) = \sum_i H(R_x)_i \times \frac{Area_o(H(R_x)_i, Q_R)}{Area(H(R_x)_i)}$$
$$EC(H(R_y)) = \sum_i H(R_y)_i \times \frac{Area_o(H(R_y)_i, Q_R)}{Area(H(R_y)_i)}$$

Where $Area(H(R_x)_i)$ and $Area_o(H(R_x)_i, Q_R)$ denote the region covered by the $i^{th}$ buckets of $H(R_x)$ and the overlapping region between this region and $Q_R$. A similar explanation is applied for $Area(H(R_y)_i)$ and $Area_o(H(R_y)_i, Q_R)$.

Based on $EC(H(R_x))$ and $EC(H(R_y))$, the estimated result size of $q$ is calculated as follows.

$$ES(q) = \frac{EC(H(R_x)) \times EC(H(R_y))}{\prod_i W_i}$$

where $W_i$ is the width of the queried region at dimension $i$.

### 5.3.2 Basic Processing Approach

BestPeer++ employs two query processing approaches: basic processing and adaptive processing. The basic query processing strategy is similar to the one adopted in the distributed databases domain. Overall, the query submitted to a normal peer $P$ is evaluated in two steps: *fetching* and *processing*. In the fetching step, the query is decomposed into a set of subqueries which are then sent to the remote normal peers that host the data involved in the query (the list of these normal peers is determined by searching the indices stored in BATON, cf. Section 5.2.2). The subquery is then processed by each remote normal peer and the intermediate results are shuffled to the query submitting peer $P$.

In the processing step, the normal peer $P$ first collects all the required data from the other participating normal peers. To reduce I/O, the peer $P$ creates a set of `MemTables` to hold the data retrieved from other peers and bulk inserts these data into the local MySQL when the `MemTable` is full. After receiving all the necessary data, the peer $P$ finally evaluates the submitted query.

The system also adopts two additional optimizations to speed up the query processing. First, each normal peer caches sufficient table index, column index, and range index entries in memory to speed up the search for data owner peers, instead of traversing the BATON structure. Second, for equi-join queries, the system employs bloom join algorithm to reduce the volume of data transmitted through the network.

During the query processing, BestPeer++ charges the user for data retrieval, network bandwidth usages and query processing. Suppose $N$ bytes of data are processed and the query consumes $t$ seconds, the cost is represented as:

$$C_{basic} = (\alpha + \beta)N + \gamma t \tag{5.1}$$

where $\alpha$ and $\beta$ denote the cost ratio of local disk and network usages respectively and $\gamma$ is the cost ratio for using a processing node for a second. Suppose one processing node can handle $\theta$ bytes data per second, the above equation becomes

$$C_{basic} = (\alpha + \beta)N + \gamma \frac{N}{\theta} \tag{5.2}$$

One problem of the basic approach is the inefficiency of query processing. The performance is bounded by $\frac{N}{\theta}$, as only one node is used. We can easily address this problem by employing more nodes to process the query in parallel.

### 5.3.3 Adaptive Processing Approach

The lesson learnt from BestPeer++ in Section 5.1.1 has led us to the proposal of the adaptive engine. In this section and its subsequent sections, we present the design and models for implementing the adaptive mechanism.

#### Implementing MapReduce for BestPeer++

Besides its generic P2P processing strategy, a MapReduce engine is implemented for BestPeer++. To facilitate MapReduce processing, a Hadoop Distributed File System (HDFS) is mounted at system start time to serve as the intermediate storage for MapReduce jobs. In general, in our MapReduce engine, the *mappers* read data directly from the BestPeer++ instances and the output of *reducers* are written back to HDFS, which

is similar to HadoopDB [7]. Specifically, a submitted SQL query is packed and sent to participant nodes using BestPeer++'s messaging substrate by the query dispatcher. Each local MySQL is connected via its JDBC interface using the query parameters given in the package and returns the intermediate results. We extend Hadoop's InputFormat class to handle data transmission between BestPeer++ nodes and HDFS. The InputFormat library provides all necessary parameters such as: database name, query fetch size and other query tuning parameters. It capsulizes each MySQL output tuple in a MapReduce Readable format and implements RecordReader interface for MapReduce job to fetch the intermediate tuple from HDFS. The intermediate results are then fed to the MapReduce job doing join and aggregation. We illustrate the integrated engine of MapReduce and P2P in Figure 5.4.

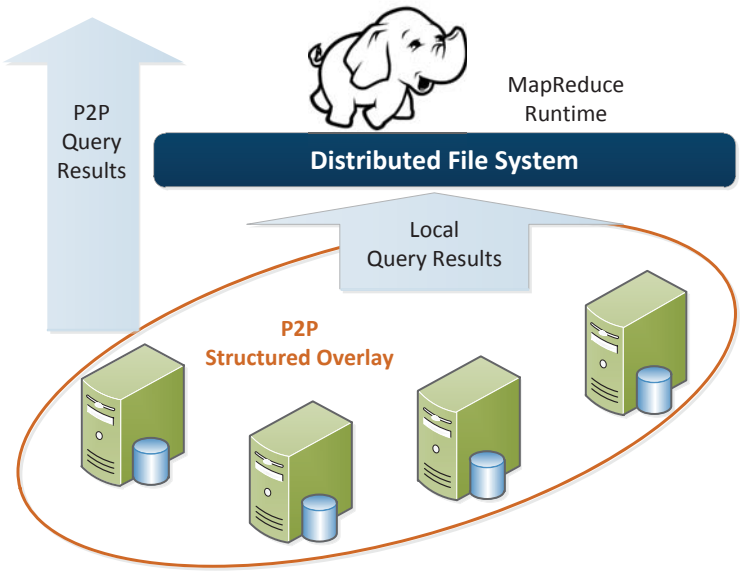



Figure 5.4: MapReduce Integration. A MapReduce layer is mounted in parallel with the P2P overlay. Query can be executed using either P2P engine or MapReduce.

The integrated design allows a query be executed either by the P2P engine or by MapReduce, under full control of the runtime optimizer.

### Modeling the Cost for Execution Plans

Since MapReduce and the P2P engine are both adopted in BestPeer++ for query processing, it is crucial to understand the performance metrics of these methods in order for the runtime to optimize its execution strategy. In this section, we present our models to

evaluate the two execution plans. We first define a processing graph in which the execution flow is expressed in a graph layered by *JOIN* and *GROUPBY* operators.

**Definition 5.3. Processing Graph**
*Given a query $Q$, the processing Graph $G = (V, E)$ is generated as follows:*

1. *For each node $v_i \in V$, we assign a level ID to $v_i$, denoted as $f(v_i)$.*
2. *Root node $v_0$ represents the peer that accepts the query, which is responsible for collecting the results for the user. $f(v_0) = 0$.*
3. *Suppose $Q$ involves $x$ JOINs and $y$ GROUPBY attributes, the maximal level of the graph $L$ satisfies $L \leq x + f(y)$ ($f(y) = 1$, if $y \geq 1$. Otherwise $f(y) = 0$). In this way, we generate a level of nodes for each JOIN operator and GROUPBY operator.*
4. *Except for the root node, all other nodes only process one JOIN operator or GROUPBY operator.*
5. *Nodes of level $L$ accept input data from the BestPeer++'s storage system (e.g. local databases). After completing its processing, node $v_i$ sends its data to the nodes in level $f(v_i) - 1$.*
6. *All of operators that are not evaluated in the non-root node are processed by the root.*

And the parameters used in the model are summarized in Table 5.3

| Notations | |
|---|---|
| $S(T)$ | Size of table T |
| $\tau(T)$ | Number of partitions of table T |
| $W_i$ | Workload of shuffling between level $i$ and $i+1$ |
| $\varphi$ | Overhead of MapReduce job |
| $\alpha$ | Cost ratio of network |
| $\beta_{BP}$ | CPU cost ratio of P2P engine |
| $\beta_{MR}$ | CPU cost ratio of MapReduce engine |
| $g(i)$ | Selectivity of $op_i$ |
| $\sigma(i)$ | Size of intermediate data after level $i$ |
| $C_{BP}$ | Total cost of P2P method |
| $C_{MR}$ | Total cost of MapReduce method |

Table 5.3: Notations for Cost Modeling

### The Parallel P2P Processing Approach

The idea of parallel processing is shown in Figure 5.5. For each join, instead of forwarding all tuples into a single processing node, we disseminate them into a set of nodes, which will process the join in parallel. We adopt the conventional replicated join approach. Namely, the small table will be replicated to all processing nodes and joined with a partition of the large table. For example, in Figure 5.5, table $S$ is replicated to two nodes and joined with the partitions of $R$ ($R1$ and $R2$). When a query involves multiple joins and group by, the query plan can be expressed as a processing graph:

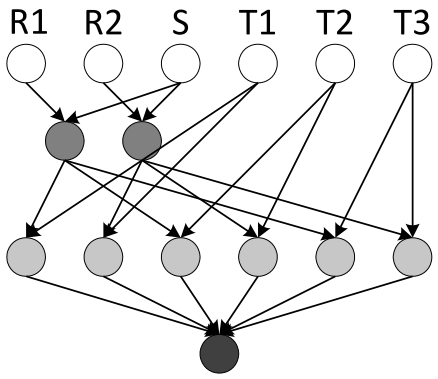



Figure 5.5: Parallel P2P Processing

In the replicated join, we trade off the network cost (a table is replicated to multiple nodes) for the parallelism. The benefit may be neutralized when a large number of tuples are re-partitioned in the P2P network. Therefore, we propose a model to estimate the cost.

The intermediate result from level $i+1$ needs to be broadcasted to all of the $\tau(T_i)$ partitions of table $T_i$ involving in level $i$'s join. In this cost model, we assume that the I/O (local and network communication) and the CPU time dominate the overall cost. First, we define the workload of $ith$ replicated join as the product of last step's workload and the number of partition of Table $T_i$:

$$W(i) = \tau(T_i) \times \sigma(i+1) \tag{5.3}$$

Meanwhile, $\sigma(i)$ obeys the recurrence relation:

$$\sigma(i) = \sigma(i+1) \times S(T_i) \times g(i) \tag{5.4}$$

Therefore, the size of intermediate result involving in level $i$'s join can be expressed as:

$$\sigma(i) = \prod_{j=L}^{i} S(T_j)g(j) \tag{5.5}$$

Combining Equation (5.3) and Equation (5.5), we obtain the workload of level $i$:

$$W(i) = \tau(T_i) \times \prod_{j=L}^{i} S(T_j)g(j) \tag{5.6}$$

The cost of level $i$ is consisted of both network and CPU cost on its workload:

$$C(i) = W(i) \times (\alpha + \beta_{BP}) \tag{5.7}$$

Since there are L levels in a processing graph, the total cost is inferred as:

$$\begin{aligned} C_{BP} &= (\alpha + \beta_{BP}) \times \sum_{i=L}^{1} W(i) \\ &= (\alpha + \beta_{BP}) \times \sum_{i=L}^{1} [\tau(T_i) \times \prod_{j=L}^{i} S(T_j)g(j)] \end{aligned} \tag{5.8}$$

**The MapReduce Approach**

The major difference between MapReduce method and generic P2P method comes from the way they process join.

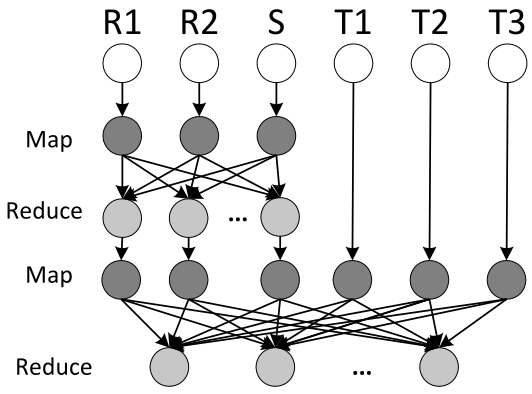



Figure 5.6: MapReduce Processing

As shown in Figure 5.6, in MapReduce method, instead of doing replicate joins, the symmetric-hash join approach is adopted. Each mapper reads in its local data and shuffles the intermediate tuple according to the hash value of the join key. Therefore, each tuple

only needs to be shuffled once on each level. Note that the configuration and launch of a MapReduce job also incurs certain overhead, which, can be measured in the runtime, is a constant value. The workload of $ith$ level can be inferred as:

$$W(i) = \sigma(i+1) + S(T_i) + \varphi \tag{5.9}$$

Similar to P2P method, the size of intermediate data in level $i$ can be derived from a recursion formula:

$$\sigma(i) = \prod_{j=L}^{i} S(T_j)g(j) \tag{5.10}$$

Therefore, the total cost for the MapReduce method is:

$$\begin{aligned} C_{MR} &= (\alpha + \beta_{MR}) \times \sum_{i=L}^{1} W(i) \\ &= (\alpha + \beta_{MR}) \times \\ &\left[ \sum_{i=L}^{1} \prod_{j=L}^{i} S(T_j)g(j) + \sum_{i=L}^{1} S(T_i) + \varphi(L-1) \right] \end{aligned} \tag{5.11}$$

### 5.3.4 Adaptive Query Processing in BestPeer++

For small jobs, the P2P engine performs better than the MapReduce engine, as it does not incur initialization cost and database join algorithms have been well optimized. However, for large-scale data analytic jobs, the MapReduce engine is more scalable, as it does not incur recursive data replications.

Based on the above-mentioned cost models, we propose our adaptive query processing approach. When a query is submitted, the query planner retrieves related histogram and index information from the bootstrap node, analyzes the query and constructs a processing graph for the query. Then the costs of both the P2P engine and MapReduce engine are predicted based on the histograms and runtime parameters of the cost models. The query planner compares the costs between two methods and executes the one with lower cost. The detailed algorithm description is shown in Algorithm 9.

Comparing between two cost models, we can observe that table size and query complexity are the key factors that affect the query planner's decision. With more levels of join, and larger size of tables, the query planner tends to choose the MapReduce method, while on the contrary, simple queries involving smaller data size and fewer joins are taken care of by the P2P method.

**Algorithm 9:** Adaptive Query Processing

**Input**: Query Q
**Output**: Query configuration on a specific query engine
**TableSet** $S \longleftarrow TableParser(Q)$ ;
**Cost** $C_{min} \longleftarrow MAX\_VALUE$ ;
**QueryPlan** $Target \longleftarrow null$ ;
**QueryPlanSet** $QS \longleftarrow \emptyset$ ;
**foreach Table** $T \in S$ **do**
    `// Generate Processing Graphs rooted on T`
    **GraphSet** $GS = GraphGen(T)$;
    `// Iterate through all Processing Graph rooted on T`
    **foreach Graph** $G \in GS$ **do**
        **QueryPlan** $P_1 = P2PPlanGen(G)$;
        **QueryPlan** $P_2 = MapredPlanGen(G)$;
        $QS = QS \cap \{P_1\}$;
        $QS = QS \cap \{P_2\}$;
**foreach QueryPlan** $P \in QS$ **do**
    **if** $CostEst(P) < C_{min}$ **then**
        $C_{min} = CostEst(P)$;
        $Target = P$;
**return** $Target$;

In order to make smart and accurate decision about which method to use, the query planner requires query statistics (such as $S(T)$, $g(i)$, $\alpha$, $\beta_{BP}$, $\beta_{MR}$, $\varphi$). These parameters are determined using a statistics module built in between the storage engine and the bootstrap node, which communicates with both to collect necessary statistics. Additionally, the statistics module is extended with a feedback-loop mechanism capable of adjusting the query parameter based on recently measured values.

## 5.4 Performance Evaluation

This section evaluates the performance and throughput of BestPeer++ on Amazon cloud platform. For the performance benchmark, we compare the query latency of BestPeer++ with HadoopDB using five queries selected from typical corporate network applications workloads. For the throughput benchmark, we create a simple supply-chain network consisting of suppliers and retailers and study the query throughput of the system.

### 5.4.1 Performance Benchmarking

This benchmark compares the performance of BestPeer++ with HadoopDB. We choose HadoopDB as our benchmark target since it is an alternative promising solution for our problem and adopts an architecture similar to ours. Comparing the two systems (i.e., HadoopDB and BestPeer++) reveals the performance gap between a general data warehousing system and a data sharing system specially designed for corporate network applications.

#### Benchmark Environment

We run our experiments on Amazon `m1.small` DB instances. Each DB small instance has 1.7GB memory, 1 EC2 Compute Unit (1 CPU virtual core). We attach each instance with 50GB storage space. We observe that the I/O performance of Amazon cloud is not stable. The `hdparm` reports that the buffered read performance of each instance ranges from 30MB/sec to 120MB/sec. To produce a consistent benchmark result, we run our experiments at the weekend when most of the instances are idle. Overall, the buffered read performance of each small instance is about 90MB/sec during our benchmark. The end-to-end network bandwidth between DB small instances, measured by `iperf`, is approximately 100MB/sec. We execute each benchmark query three times and report the average execution time. The benchmark is performed on cluster sizes of 10, 20, 50 nodes. For the BestPeer++ system, these nodes are normal peers. We launch an additional dedicated node as the bootstrap peer. For HadoopDB system, each launched cluster node acts as a worker node which hosts a Hadoop task tracker node and a PostgreSQL database server instance. We also use a dedicated node as the Hadoop job tracker node and HDFS name node.

**BestPeer++ Settings**

The configuration of a BestPeer++ normal peer consists of two parts: the underlying MySQL database server and the BestPeer++ software. For MySQL database, we use the default MyISAM storage engine which is optimized for read-only queries since no transactional processing overhead is introduced. We set up a large index memory buffer (500MB) and the maximum number of tables to be concurrently opened (50 tables). For BestPeer++ software stack, we set the maximum memory consumed by the `MemTable` to be 100MB. We also configure each normal peer to use 20 concurrent threads for fetching data from remote peers. Finally, we configure each normal peer to use the basic query processing strategy.

**HadoopDB Settings**

We carefully follow the instructions presented in the original HadoopDB paper to configure HadoopDB. The setting consists of the setup of a Hadoop cluster and the PostgreSQL database server hosted at each worker node. We use Hadoop version 0.19.2 running on Java 1.6.0_20. The block size of HDFS is set to be 256MB. The replication factor is set to 3. For each task tracker node, we run one map task and one reduce task. The maximum Java heap size consumed by the map task or the reduce task is 1024MB. The buffer size of read/write operations is set to 128KB. We also set the sort buffer of the map task to 512MB with 200 concurrent streams for merging. For reduce task, we set the number of threads used for parallel file copying in the shuffle phase to be 50. We also enable the buffer reuse between the shuffling phase and the merging phase. As a final optimization, we enable JVM reuse.

For the PostgreSQL instance, we run PostgreSQL version 8.2.5 on each worker node. The shared buffers used by PostgreSQL is set to 512MB. The working memory size is 1GB. We only present the results for SMS-coded HadoopDB, i.e., the query plan is generated by HadoopDB's SMS planner.

**Datasets**

Our benchmark consists of five queries, denoted as Q1, Q2, Q3, Q4, and Q5 which are executed on the TPC-H datasets. We implement the benchmark queries by ourselves since the TPC-H queries are complex and time-consuming queries which are not suitable for benchmarking corporate network applications.

The TPC-H benchmark dataset consists of eight tables. We use the original TPC-H schema as the shared global schema. HadoopDB does not support schema mapping and access control. To benchmark the two systems in the same environment, we perform

| | |
|---|---|
| LineItem | *l_shipdate, l_commitdate, l_receiptdate* |
| Orders | *o_custkey, o_orderpriority, o_orderdate* |
| Customer | *c_mktsegment* |
| PartSupp | *ps_supplycost* |
| Part | *p_brand, p_type, p_size, p_mfgr* |

Table 5.4: Secondary Indexes for TPC-H Tables

additional configurations on BestPeer++ as follows. First, we set the local schema of each normal peer to be identical to the global schema. Therefore, the schema mapping is trivial and can be bypassed. We, thus, let the data loader directly load the raw data into the global table without any transformations. Second, we create a unique role $R$ at bootstrap peer. The unique role is granted full access to all eight tables. A benchmark user is created at one normal peer for query submitting. All normal peers are configured to assign the role $R$ to the benchmark user. In summary, in the performance benchmark, each normal peer contributes data to all eight tables. As a result, to evaluate a query, the query submitting peer will retrieve data from every normal peer. Finally, we generate the datasets using TPC-H `dbgen` tool and distribute 1GB data per node. Totally, we generate datasets of 10GB, 20GB, and 50GB for cluster sizes of 10, 20, 50 nodes.

### Data Loading

The data loading process of BestPeer++ is performed by all normal peers in parallel and is consisted of two steps. In the first step, each normal peer invokes the data loader to load raw TPC-H data into the local MySQL databases. In addition to copying raw data, we also build indices to speedup query processing. First, a primary index is built for each TPC-H table on the primary key. Second, some additional secondary indices are built on selected columns of TPC-H tables. Table 5.4 summarizes the secondary indices that we built. After the data is loaded into the local MySQL database, each normal peer invokes the data indexer to publish index entries to the BestPeer++ network. For each table, the data indexer publishes a table index entry and a column index entry for each column. Since the values in TPC-H datasets follow uniform distribution, each normal peer holds approximately the same data range for each column of the table, therefore, we do not configure normal peer to publish range index.

For HadoopDB, data loading process is straightforward. For each worker node, we load only raw data into the local PostgreSQL database instance using SQL COPY command and build corresponding primary and secondary indices for each table. We did not employ the Global Hasher and Local Hasher to further co-partition tables. HadoopDB co-partitions tables among worker nodes on join key in order to speed up join processing

[3]. However, in a corporate network, data is fully controlled by each business. It is undesirable for a certain business to move data to normal peers managed by other businesses due to privacy and safety concern. Therefore, we disabled this co-partition function for HadoopDB.

### The Q1 Query Results

The first benchmark query Q1 evaluates a simple selection predicate on the `l_shipdate` and `l_commitdate` attributes from the `LineItem` table. The predicates yields approximately 3,000 tuples per normal peer.

```
SELECT l_orderkey, l_receiptdate
   FROM LineItem
   WHERE l_shipdate > Date(1998-11-05) AND
         l_commitedate > Date(1998-09-29)
```

The BestPeer++ system evaluates the query by fetching and processing strategy described in Section 5.3. The query executor first searches for those peers that hold the `LineItem` table. In our settings, the search will return all normal peers since each normal peer hosts all eight TPC-H tables. Then, the query executor generates a subquery for each normal peer by pushing the selection and projection clause into that peer. The final results are produced by merging partial results returned from data owner peers.

HadoopDB's SMS planner generates a single MapReduce job to evaluate the query. The MapReduce job only consists of a map function which takes the SQL query, generated by SMS planner, as input, executes the query on local PostgreSQL instance and writes the results into a HDFS file. Similar to BestPeer++, HadoopDB's SMS planner also pushes projection and selection clause to remote worker nodes.

The performance of each system is presented in Figure 5.7. Both systems (HadoopDB and BestPeer++) perform this query within a short time. This is because both systems benefit from the secondary indices built on `l_shipdate` and `l_commitdate` columns. However, the performance of BestPeer++ is significantly better than HadoopDB. The performance gap between HadoopDB and BestPeer++ is attributed to the startup costs of MapReduce job introduced by the Hadoop layer, including the cost of scheduling map tasks on available task tracker nodes and the cost of launching a fresh new Java process on each task tracker node to perform the map task. We note that independent of the cluster size, Hadoop requires approximately 10∼15 seconds to launch all map tasks. This startup

[3] If two tables are co-partitioned on the join column, the join over the two tables can be performed locally without shuffling.

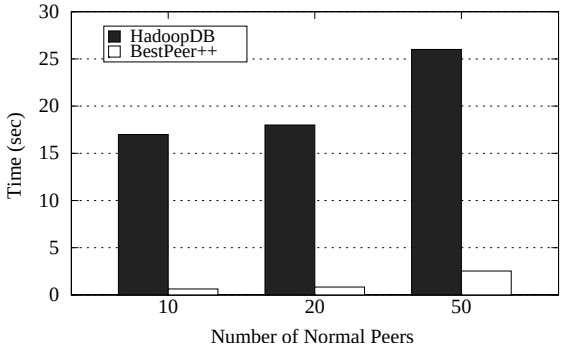



Figure 5.7: Results for Q1.

cost, therefore, dominates the query processing. BestPeer++, on the other hand, has no such startup cost since it does not require a job tracker node to schedule tasks among normal peers. Moreover, to execute a subquery, the remote normal peer does not launch a separate Java process. Instead, the remote normal peer just forwards that subquery to the local MySQL instance for execution.

### The Q2 Query Results

The second benchmark query Q2 involves computing the total prices over the qualified tuples stored in `LineItem` table. This simple aggregation query represents another kind of typical workload in a corporate network.

```
SELECT l_returnflag, l_linestatus
   SUM(l_extendedprice)
   FROM LineItem
   WHERE l_shipdate > Date(1998-09-01) AND
         l_discount < 0.06 AND
         l_discount > 0.01
   GROUP BY
         l_returnflag, l_linestatus
```

The query executor of BestPeer++ first searches for peers that host the `LineItem` table. Then, it sends the entire SQL query to each data owner peer for execution. The partial aggregation results are then sent back to the query submitting peer where the final aggregation is performed.

The query plan generated by the SMS planner of HadoopDB is identical to the query plan employed by BestPeer++'s query executor described above. The SMS compiles this query into one MapReduce and pushes the SQL query to the map tasks. Each map task,

then, performs the query over its local PostgreSQL instance and shuffles the results to the reducer side for final aggregation.

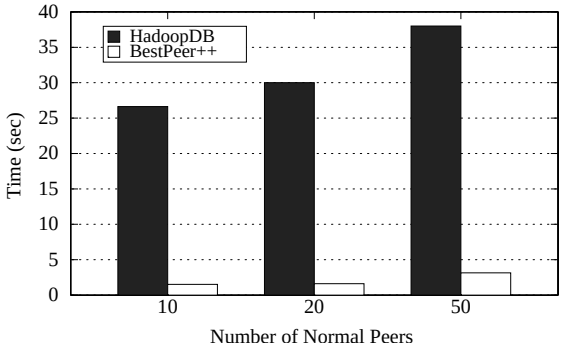



Figure 5.8: Results for Q2.

The performance of each benchmarked system is presented in Figure 5.8. BestPeer++ still outperforms HadoopDB by a factor of ten. The performance gap between HadoopDB and BestPeer++ comes from two factors. First, the startup costs introduced by Hadoop layer still dominates the execution time of HadoopDB. Second, Hadoop (and generally MapReduce) employs a pull based method to transfer intermediate data between map tasks and reduce tasks. The reduce task must periodically queries the job tracker for the map completion events and start to pull data after it has retrieved these completion events. We observe that, in Hadoop, there is a noticeable delay between the time point of map completion and the time point of those completion events being retrieved by the reduce task. Such delay slows down the query processing. The BestPeer++ system, on the other hand, has no such delay. When a remote normal peer completes its subquery, it directly sends the results back to the query submitting peer for final processing. That is, BestPeer++ adopts a push based method to transfer intermediate data between remote normal peers and the query submitting peer. We observe that, for short queries, the push approach is better than pull approach since the push approach significantly reduces the latency between the data consumer (query submitting peer) and the data producer (remote normal peer).

### The Q3 Query Results

The third benchmark query Q3 involves retrieving qualified tuples from joining two tables, i.e., `LineItem` and `Orders`.

```
SELECT l_orderkey, l_shipdate
   FROM LineItem, Orders
   WHERE l_orderkey = o_orderkey
         AND l_receiptdate < Date(1994-02-07)
```

```
AND l_receiptdate > Date(1994-01-01)
AND o_orderdate < Date(1994-01-31)
AND o_orderdate > Date(1994-01-01)
```

To evaluate this query, BestPeer++ first identifies the peers that host `LineItem` and `Orders` tables. Then, the normal peer retrieves qualified tuples from those peers and performs the join.

The query plan produced by SMS planner of HadoopDB is similar to the one adopted by BestPeer++. The map tasks retrieve qualified tuples of `LineItem` and `Orders` tables and sort those intermediate results based on `l_orderkey` (for `LineItem` tuples) and `o_orderkey` (for `Orders` tuples). The sorted tuples are joined at reducer side using a merge-join algorithm. By default, the SMS planner only launches one reducer to process this query. We found that the default setting yields poor performance. Therefore, we manually set the number of the reducers to be equal to the number of worker nodes and only report results with this manual setting.

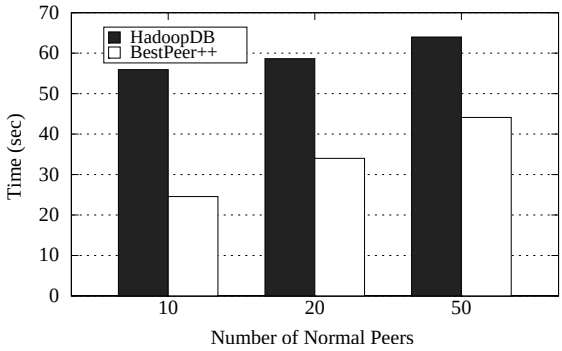



Figure 5.9: Results for Q3.

Figure 5.9 presents the performance of both systems. We can observe from this figure that, the performance gap between BestPeer++ and HadoopDB becomes smaller. This is because this query requires to process more tuples than previous queries. Therefore, the Hadoop startup costs is amortized by the increased workload. We also see that as the number of nodes grows, the scalability of HadoopDB is slightly better than BestPeer++. This is because BestPeer++ performs the final join processing at the query submitting peer. Therefore, the data which are required to process at the query submitting peer grows linearly with the number of normal peers, resulting in performance degradation. HadoopDB, however, can distribute the final join processing to all worker nodes and thus insensitive to data volume needed to be processed. We should note that, in real deployment, we can boost the performance of BestPeer++ by scaling-up the normal peer instance.

### The Q4 Query Results

The fourth benchmark query Q4 is as follows.

```
SELECT p_brand, p_size, SUM(ps_avialqty),
   SUM(ps_supplycost)
   FROM PartSupp, Part
   WHERE p_partkey = ps_partkey
         AND p_size < 10
         AND ps_supplycost < 50
         AND p_mfgr = 'Manufacturer#3'
   GROUP BY p_brand, p_size
```

The BestPeer++ system evaluates this query by first fetching qualified tuples from remote peers to query submitting peer and storing those tuples in `MemTables`. The BestPeer++, then, joins tuples stored in the `MemTables` and produces the final aggregation results.

The SMS planner of HadoopDB compiles the query into two MapReduce jobs. The first job joins `PartSupp` and `Part` tables. The SMS planner pushes selection conditions to the map tasks in order to efficiently retrieve qualified tuples by using indices. The join results are then written to HDFS. The second MapReduce job is launched to process the joined tuples and produce the final aggregation results.

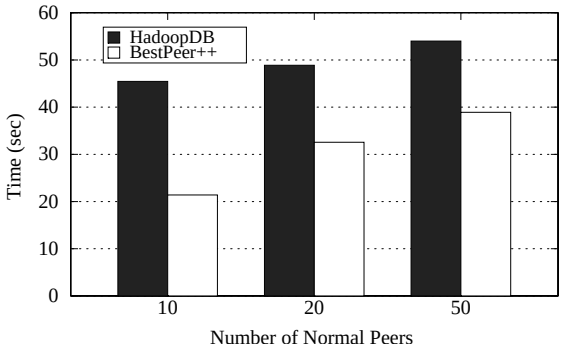



Figure 5.10: Results for Q4.

Figure 5.10 presents the performance of both system. We can see that BestPeer++ still outperforms HadoopDB. But the performance gap between the two systems are much smaller. Also, HadoopDB achieves better scalability than BestPeer++. This is because HadoopDB can benefit from parallelism by distributing the join and aggregation processing among worker nodes. However, to achieve that, we must manually set the number of reducers to be equal to the number of worker nodes. BestPeer++, on the other hand, only performs the join and the final aggregation at the query submitting peer. As more nodes

are involved, more data need to be processed at the query submitting peer, resulting in that peer to be over-loaded. Again, the performance problem of BestPeer++ can be mitigated by upgrading the normal peer to a larger instance.

### The Q5 Query Results

The final benchmark query Q5 involves a muti-tables join and is defined as follows.

```
SELECT c_custkey, c_name,
   SUM(l_extendedprice*(1-l_discount)) AS R
   FROM Customer, Orders, LineItem, Nation
   WHERE c_custkey = o_custkey
         AND l_orderkey = o_orderkey
         AND c_nationkey = n_nationkey
         AND l_returnflag = 'R'
         AND o_orderdate >= date(1993-10-01)
         AND o_orderdate < date(1993-12-01)
   GROUP BY c_custkey, c_name
```

Figure 5.11 presents the results of this benchmark. Overall, HadoopDB performs better than BestPeer++ in evaluating this query. This distinction comes from the query execution strategies the two systems use.

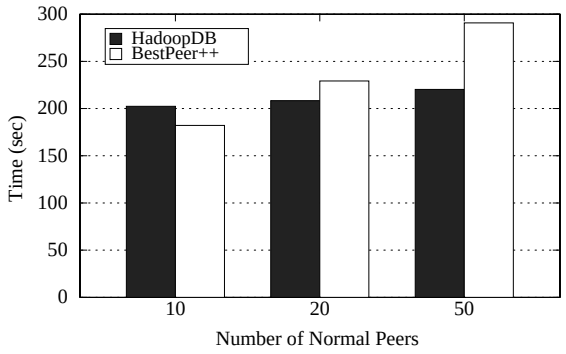



Figure 5.11: Results for Q5.

The original P2P strategy executes this query by first fetching all qualified tuples to the query submitting peer and then do the final join and aggregation processing. HadoopDB compiles this query into four MapReduce jobs with the first three jobs performing the joins and the final job performing the final aggregation.

In BestPeer++, the query submitting peer joins all qualified tuples, thus at a large scale (20 and 50 nodes), the query submitting peer becomes the bottleneck, impacting system's performance. HadoopDB, on the contrary, utilizes all nodes to perform joins in parallel and hence has a better scalability, which can be seen in Figure 5.11.

We identified this problem and proposed our adaptive query processing strategy, which will be evaluated in the next experiment.

### Adaptive Query Processing Results

To demonstrate our adaptive query processing strategy, we further evaluate Q5 using three different engines separately alone, namely the P2P engine, the MapReduce engine and the adaptive query engine for BestPeer++. To start with, we compile and execute Q5 on either the P2P engine or the MapReduce engine. In each case, we enforce our query planner to invoke either the P2P engine or the MapReduce engine alone, regardless of the possible cost. As described in the previous experiment, the execution strategies of these two engines differ from each other in the way that they shuffles intermediate data and organize the joins, which leads to a considerable performance gap. We then use our adaptive processing engine to make comparison.

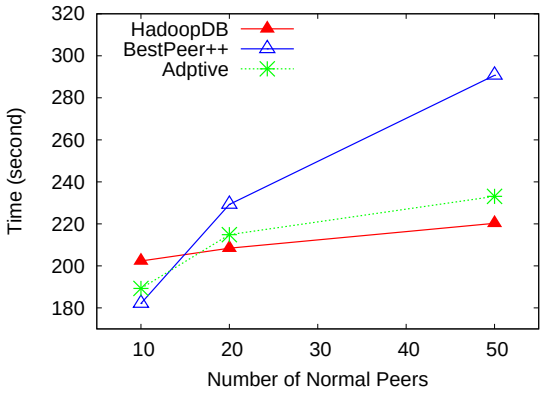



Figure 5.12: Adaptive Query Processing.

Figure 5.12 presents the performance of these three processing strategy. The P2P engine works better in a smaller scale (10 data nodes). With the increase of data scale, we witness a decent performance gain from the MapReduce engine, who then outperforms the P2P engine at the scale of 20 and 50 data nodes. Such a trend complies the prediction of our cost model in the sense that the P2P engine handles lighter workload nicely, while on the contrary, the MapReduce scales better with more complex queries.

Taking use of the insight that our cost model gives, the adaptive engine switches between the P2P engine and the MapReduce engine to accommodate itself to a vaster variety of queries in a cost efficient way. The results from figure 5.12 shows the effectiveness end efficiency of the adaptive engine. With a negligible overhead for constructing plans for both engine and evaluating the cost, the performance of the adaptive engine approaches whatever the better one under different workload setups.

### 5.4.2 Throughput Benchmarking

This Section studies the query throughput of BestPeer++. HadoopDB is not designed for high query throughput, therefore, we intentionally omit the results of HadoopDB and only present the results of BestPeer++. We conduct two tiers of benchmark evaluation for the performance and scalability of BestPeer++, respectively.

#### Benchmark Settings

We establish a simple supply-chain network to benchmark the query throughput of the BestPeer++ system. The supply-chain network consists of a group of *suppliers* and a group of *retailers* which query data from each other. Each normal peer either acts as a supplier or a retailer. We set the number of suppliers to be equal to the number of retailers. Thus, in the cluster with 10, 20, and 50 normal peers, there are 5, 10, and 25 suppliers and retailers respectively.

We still use the TPC-H schema as the global shared schema, but partition the schema into two sub-schema, one for suppliers and the other for retailers. The supplier schema consists of the following tables: `Supplier`, `PartSupp`, and `Part`. The retailer schema involves `LineItem`, `Orders`, and `Customer` tables. The `Nation` and `Region` tables are commonly owned by both supplier peers and retailers peers. We partition the TPC-H datasets into 25 datasets, one dataset for each nation, and configure each normal peer to only host data from a unique nation. The data partition is performed by first partitioning `Customer` and `Supplier` tables according to their nation keys. Then, joining each `Supplier` and `Customer` dataset with the other four tables (i.e., `Part`, `PartSupp`, `Orders`, `LineItem` respectively, the joined tuples in those tables finally form the corresponding partitioned datasets. To reflect the fact that each table is partitioned based on nations, we modify the original TPC-H schema and add a nation key column in each table.

For scalability evaluation, we scale-up the amount of data and the number of normal peer proportionally. Eventually, we generate a 50GB raw TPC-H dataset on 50 normal peers, which consists of 25 suppliers and 25 retailers, and measure the absolute system throughput for the two types of peers respectively. In the performance evaluation, we retain the data size and peer scale (50 normal peers and 50GB data in our setup), and increase the throughput, until the point at which the system is saturated and throughput stops increasing. We report the average latency versus throughput curve, as in the YCSB [29] tool's terminology.

We configure the access control module as follows. We set up two roles: supplier and retailer. The supplier role is granted full access to tables hosted by retailer peers. The retailer role is granted full access to tables hosted by supplier peers. We should not

be confused with the supplier role and the supplier peer. The supplier peer is a BestPeer++ normal peer which hosts tables belonged to a supplier (i.e., `Supplier`, `Part`, `PartSupp` tables). The supplier role is an entity in the access control policy which will be used by a local administrator of a retailer peer to grant users of supplier peers to access tables (i.e., `LineItem`, `Orders`, `Customer`) hosted at the local MySQL instance. We also create a throughput test user at each normal peer (either supplier peer or retailer peer) for query submission. Each retailer peer is tasked to assign the supplier role to users from supplier peers. We also let each supplier peer assign the retailer role to users of retailer peers. In this setting, users in retailer peers can access data stored in supplier peers but cannot access data stored in other retailers.

### Data Loading

The data loading process is similar to the loading process described in Section 5.4.1. The only difference is that in addition to publishing the table indices and column indices, we also build a range index on the nation key column of each table in order to avoid accessing suppliers or retailers which do not host data of interest.

### Results for Throughput Benchmark

The throughput benchmark queries of suppliers and retailers are as follows:

```
SELECT s_name, s_address
   FROM Supplier, PartSupp, Part
   WHERE p_type like 'MEDIUM POLISHED%' AND
         p_size < 10 AND p_availqty < 300 AND
         s_suppkey = ps_suppkey AND
         p_partkey = ps_partkey


SELECT l_orderkey, o_orderdate,
        o_shippriority, SUM(l_extendedprice)
   FROM Customer, Orders, LineItem
   WHERE c_mktsegment = 'BUILDING' AND
         o_orderdate < Date(1995-03-15) AND
         l_shipdate > Date(1995-03-15) AND
         c_custkey = o_custkey AND
         l_orderkey = o_orderkey
   GROUP BY l_orderkey, o_orderdate,
         o_shippriotiy
```

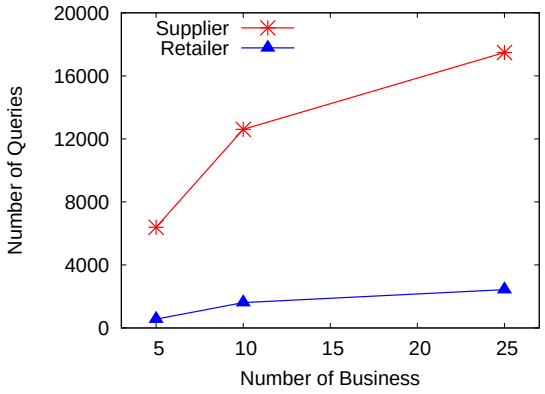
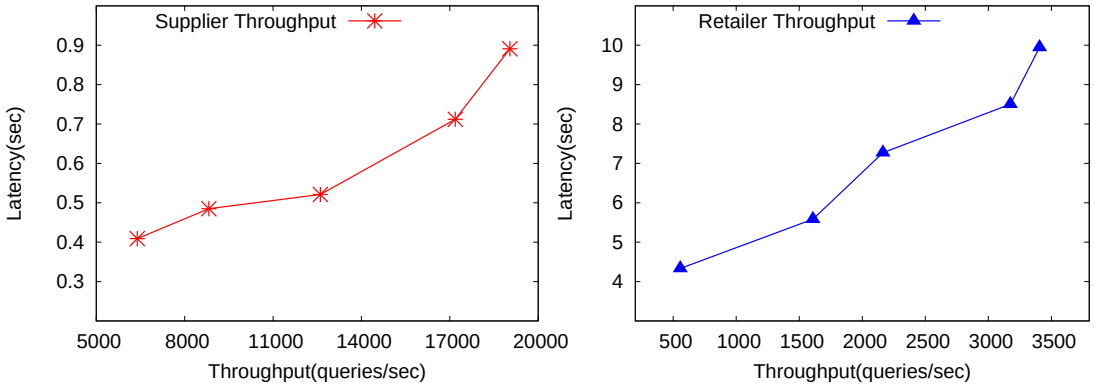



Figure 5.13: Scalability Evaluation.

The above queries omit the selection clauses on nation key column to save space. In the actual benchmark, we append a nation key clause (e.g., `s_nationkey = 0`) for each table to restrict the data access on a single supplier or retailer. The benchmark is performed in two rounds: supplier round and retailer round. In the supplier round, the throughput test user at each supplier peer sends retailer benchmark queries to retailer peers. In the retailer round, the throughput test user at each retailer peer sends supplier benchmark queries to supplier peers. In each round, the nation key is randomly chosen among available nations. Each benchmark query only queries data stored in one retailer or supplier's database. Each round begins with a 20 minutes warming up. The throughput, namely the number of queries being processed, are collected from the next 20 minutes benchmark.

Figure 5.13 presents the results of scalability evaluation. We can see that BestPeer++ achieves near linear scalability in both heavy-weight workload (i.e., retailer queries) and light-weight workload(i.e., supplier queries). The main reason for this is that BestPeer++ adopts a single peer optimization. In our benchmark, all queries will only touch just one normal peer. In the peer searching phase, if the query executor finds that a single normal peer hosts all required data, the query executor employs the single peer optimization and sends the entire SQL to that normal peer for execution. The results returned by that normal peer are directly sent back to the user. The final processing phase is entirely skipped.

Figure 5.14a and figure 5.14b plot the system performance by showing the average query latency versus throughput curves for both business participants. The heavy-weight retailer workload suffers from higher latency because of its higher computational demand. In contrast, the supplier workload incurs lower latency at higher throughput. Overall, the BestPeer++ system achieves relatively high throughput with acceptable response time to analytical queries. The heavy-weight retailer queries are finished within 10 seconds at maximum throughput (3400 queries/sec), while the light-weight supplier queries achieve




(a) Supplier Performance (b) Retailer Performance

Figure 5.14: System Throughput.

better performance with less than 1 second latency when throughput peaks at 19000 queries/sec. This is because BestPeer++ employs BATON overlay [51] to organize normal peers as a balanced tree, and thus avoiding performance bottleneck at high throughput.

## 5.5 Summary and Contributions

In this work, we first studied the query performance of parallel database systems and MapReduce based on the observations from our previous work [23], and identified the influencing factors with respect to query's complexity. We then proposed a cost model to evaluate the execution efficiency of a given query when using parallel database and MapReduce. This cost model takes into account data distribution and query parameters, and gives a quantitative guideline for runtime optimization.

Based on the proposed cost model, we presented BestPeer++, an adaptive query processing mechanism in distributed environment. BestPeer++ is a hybrid system incorporating query processing mechanisms from parallel database and MapReduce. We presented the implementation of an adaptive query processing mechanism that is able to provide optimal efficiency for different types of query.

The adaptive query processing scheme is one of the first hybrid proposals to support both MapReduce processing and traditional parallel RDBMS processing in one system. It combines the nice features from RDBMSs (e.g., indexing, query optimization) and MapReduce (e.g., arbitrarily massive parallel processing), and offers a flexible and efficient query execution mechanism by modeling the execution cost at runtime.

# Part III

# Concluding Remarks

# CHAPTER 6

# Conclusion and Future Directions

*"Reasoning draws a conclusion, but does not make the conclusion certain, unless the mind discovers it by the path of experience."*

– Roger Bacon.

## 6.1 Concluding Discussion

Over the past few years, cloud computing has emerged as a multi-billion dollar industry and as a successful paradigm for web application deployment. Irrespective of the cloud provider or the cloud abstraction, data is central to applications deployed in the cloud. The data management layer stores and serves an application's critical data, and it forms a mission critical component in the cloud software stack.

The variety of data management systems deployed in a Cloud infrastructure and supporting diverse applications face unique challenges. First, traditional RDBMSs cannot naturally scale-out and be deployed in the Cloud due to the complex software stack and stringent ACID requirement; and second, as one of the few tools available for large scale data processing, MapReduce is noted to have suboptimal performance because it lacks many of the features that have been proven invaluable for structured data analysis workloads. These two major challenges largely limit MPP system's applicability to a wider variety of data and workload types. Therefore, we believe that the next generation MPP systems should syncretize the merits of existing approaches. The strong features of MapReduce clearly need to be retained; however, they should be combined with efficient data access methods and query optimization techniques present in traditional DBMSs.

The overarching goal of this dissertation was to exploit the opportunity for a better integration of RDBMS technologies and Cloud Computing systems. In particular, we focused on making data access more efficient for MapReduce by designing scalable indexing techniques, and making parallel DBMS's processing more efficient by leveraging MapReduce adaptively. The support for indexes improves the performance of MapReduce, and allows more data analysis applications to benefit from MapReduce. What is more, integrating MapReduce with traditional parallel DBMS in the query processor improves the dynamism of MapReduce, making it more adaptive to workloads. The hybrid solution closes the performance gap between MPP systems and traditional parallel DBMSs.

In the area of exploiting index in MapReduce systems, we proposed the design and implementation of two systems that offer index service in MapReduce. The key insight for both designs was to incorporate indexes in MapReduce's execution stack, thus providing selective access to data other than brute-force scanning, and resulting in better resource allocation and higher scalability.

First, we proposed BIDS, a bitmap-based indexing scheme for large scale distributed data store. BIDS is one of the first index service proposed for MapReduce-based systems to directly work on the underlying index, We presented the design and implementation of the index service in Chapter 3. Given the consideration of size, we firstly proposed to use effective bitmap encoding and partial index schemes to achieve high space efficiency. In addition, we designed a full-fledged mechanism that allows the index be directly processed by MapReduce. The construction and query of index can both leverage the parallelism of MapReduce. Furthermore, we introduced series of runtime optimizations to facilitate efficient query processing in MapReduce.

The next objective of this dissertation was to provide an index framework in MapReduce systems. Towards this goal, we proposed in Chapter 4 *ScalaGiST*, a generalized index framework to extend the indexibility in MapReduce systems. *ScalaGiST* provides extensibility in terms of data and query types, and hence is able to support unconventional queries (e.g., spatial-temporal queries) in MapReduce system. Firstly, we defined the generalized index interface based on which users are able to customize new types of index on their data. We then presented the design and implementation of an index processing mechanism to integrate *ScalaGiST* seamlessly with Hadoop platform. What is more, we proposed a cost-based data access optimizer for improving the performance of MapReduce execution. Indexibility in MapReduce systems is decisive in boosting query performance, and *ScalaGiST* is the first framework that provides support to a wide variety of traditional indexes in distributed environment. In addition, through an extensive experimental study, we showed that *ScalaGiST* offers good scalability and wide support

for various types of indexes. Having the efficiency and flexibility, *ScalaGiST* can be an invaluable tool for index applications in MapReduce systems.

In the area of adaptive query processing using MapReduce and parallel DBMS, we proposed the design and implementation of a hybrid system that adaptively employs MapReduce and P2P based DBMS based on different workloads. The key insight for this hybrid solution was to provide flexibility for general purposed MPP system by integrating MapReduce and parallel DBMS in one query processor in order to adapt to different workloads. The adaptive query processing scheme proposed in Chapter 5 is one of the first hybrid proposals to support both MapReduce processing and traditional parallel RDBMS processing in one system. It combines the nice features from RDBMSs (e.g., indexing, query optimization) and MapReduce (e.g., arbitrarily massive parallel processing), and offers a flexible and efficient query execution mechanism by modeling the execution cost at runtime.

We first studied the query performance of parallel database systems and MapReduce, and identified the influencing factors with respect to query's complexity. We then proposed a cost model to evaluate the execution efficiency of a given query when using parallel database and MapReduce. This cost model takes into account data distribution and query parameters, and gives a quantitative guideline for runtime optimization.

Based on the proposed cost model, we presented BestPeer++, an adaptive query processing mechanism in distributed environment. BestPeer++ is a hybrid system incorporating query processing mechanisms from parallel database and MapReduce. We presented the implementation of an adaptive query processing mechanism that is able to provide optimal efficiency for different types of query.

For each of these proposals, we implemented the components and thoroughly evaluated them using various benchmark queries and datasets. This dissertation makes fundamental contributions in the two thrust areas of indexes in MapReduce and adaptive data processing. These advances are critical to the design of data processing systems for cloud computing infrastructure and significantly advances the state-of-the-art in that field.

## 6.2 Future Directions

The continuing growth of data sizes, advent of novel applications, and evolution of the infrastructure engender new research challenges. While some of these future research directions are direct extensions of the techniques presented in this thesis, others are more radical.

In the area of creating a hybrid MapReduce/parallel database system, one interesting research question that would stem from such a hybrid integration would be how to further

push DBMS features to reinforce MapReduce. DBMS typically employs a storage engine to harness data, while MapReduce directly sits on files stored in DFS whose interface is rather simple without much of the optimizations (e.g., index, materialized view, compression, columnar storage, etc.). Decoupling the storage optimizations in DBMS and applying them to raw data in DFS is promising in further boosting MapReduce's performance. Incremental algorithms are called for, where data can initially be read directly off of the file system (distributed), but each time data is accessed, progress is made towards the many activities surrounding a DBMS load.

A major challenge in data analytics today stems from the sheer volume of data available for processing. The data storage and processing techniques that we presented in this dissertation were aimed at handling such large datasets. This challenge of dealing with very large datasets has been termed the *volume* challenge. There are two other related challenges, namely, those of *velocity* and *variety*.

The velocity challenge refers to the short response-time requirements for collecting, storing, and processing data. The research we conducted in this dissertation are based on batch systems. For latency sensitive applications, such as identifying potential fraud and recommending personalized content, batch data processing is insufficient. The data may need to be processed as it streams into the system in order to extract the maximum utility from the data. Therefore, one interesting challenge is to dynamically index the data while providing fast and accurate query result. We envision two sub-problems towards this challenge: techniques to incrementally update existing indexes, and techniques to accumulatively build and process index on-the-fly for growing data.

The variety challenge refers to the growing list of data types – relational, time series, text, graphs, audio, video, images, genetic codes – as well as the growing list of analysis techniques on such data. New insights could be found while analyzing more than one of these data types together. The storage and processing techniques that we have seen in this dissertation are predominantly aimed at handling data that can be represented using a relational model (rows and columns) and processed by query plan operators like filters, joins, and aggregation. However, the new and emerging data types cannot be captured easily in a relational data model, or analyzed easily by software that depends on running operators like filters, joins, and aggregation. This challenge calls for newly designed indexing scheme to capture the variety types of data while providing indexing scalability and extensibility.

Another interesting research question is how to balance the tradeoffs between fault tolerance and performance. Maximizing fault tolerance typically means carefully checkpointing intermediate results, but this usually comes at a performance cost (e.g., the rate which data can be read off disk in the sort benchmark from the original MapReduce paper

is half of full capacity since the same disks are being used to write out intermediate Map output). A system that can adjust its levels of fault tolerance on the fly given an observed failure rate could be one way to handle the tradeoff.

# Bibliography


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

[2] Apache HBase Homepage. http://hbase.apache.org/.

[3] Apache Storm Homepage. http://storm.incubator.apache.org/.

[4] TPC-H Homepage. http://www.tpc.org/tpch/.

[5] Daniel J. Abadi. Tradeoffs between parallel database systems, hadoop, and hadoopdb as platforms for petabyte-scale analysis. In *Scientific and Statistical Database Management*, pages 1–3. Springer, 2010.

[6] Karl Aberer, Anwitaman Datta, and Manfred Hauswirth. Route maintenance overheads in dht overlays. In *6th Workshop on Distributed Data and Structures (WDAS)*, 2004.

[7] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proceedings of the VLDB Endowment*, 2(1):922–933, 2009.

[8] Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a map-reduce environment. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 99–110. ACM, 2010.

[9] Marcos K. Aguilera, Wojciech Golab, and Mehul A. Shah. A practical scalable distributed b-tree. *Proceedings of the VLDB Endowment*, 1(1):598–609, 2008.

[10] Marcos K. Aguilera, Arif Merchant, Mehul A. Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 159–174. ACM, 2007.

[11] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving relations for cache performance. In *VLDB*, volume 1, pages 169–180, 2001.

[12] Carlo Batini, Maurizio Lenzerini, and Shamkant B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM computing surveys (CSUR)*, 18(4):323–364, 1986.

[13] David Bermbach and Stefan Tai. Eventual consistency: How soon is eventual? an evaluation of amazon s3's consistency behavior. In *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*, page 1. ACM, 2011.

[14] 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 *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 975–986. ACM, 2010.

[15] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D Ernst. Haloop: Efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1-2):285–296, 2010.

[16] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350. USENIX Association, 2006.

[17] Yu Cao, Chun Chen, Fei Guo, Dawei Jiang, Yuting Lin, Beng Chin Ooi, Hoang Tam Vo, Sai Wu, and Quanqing Xu. Es 2: A cloud data storage system for supporting both oltp and olap. In *ICDE*, pages 291–302. IEEE, 2011.

[18] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment*, 1(2):1265–1276, 2008.

[19] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407. ACM, 2007.

[20] Sashikanth Chandrasekaran and Roger Bamford. Shared cache-the future of parallel databases. In *ICDE*, pages 840–840. IEEE Computer Society, 2003.

[21] 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 15–15, 2006.

[22] 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. *Proceedings of the VLDB Endowment*, 4(12):1318–1327, 2011.

[23] Gang Chen, Tianlei Hu, Dawei Jiang, Peng Lu, K-L Tan, Hoang Tam Vo, and Sai Wu. Bestpeer++: A peer-to-peer based large-scale data processing platform. In *ICDE*, pages 582–593. IEEE, 2012.

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

[25] Songting Chen. Cheetah: A high performance, custom data warehouse on top of mapreduce. *Proceedings of the VLDB Endowment*, 3(2), 2010.

[26] Rupesh Choubey, Li Chen, and Elke A Rundensteiner. Gbi: A generalized r-tree bulk-insertion strategy. In *Advances in Spatial Databases*, pages 91–108. Springer, 1999.

[27] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proc. of VLDB*, pages 426–435, 1997.

[28] 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. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[29] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[30] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[31] 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*, volume 7, pages 205–220, 2007.

[32] David J. DeWitt, Robert H. Gerber, Goetz Graefe, Michael L. Heytens, Krishna B. Kumar, and M. Muralikrishna. Gamma-a high performance dataflow database machine. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 228–237. Morgan Kaufmann Publishers Inc., 1986.

[33] David J. DeWitt and Jim Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

[34] David J. DeWitt and Michael Stonebraker. Mapreduce: A major step backwards. *The Database Column*, 1, 2008.

[35] 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). *Proceedings of the VLDB Endowment*, 3(1-2):515–529, 2010.

[36] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Stefan Richter, Stefan Schuh, Alekh Jindal, and Jörg Schad. Only aggressive elephants are fast elephants. *Proceedings of the VLDB Endowment*, 5(11):1591–1602, 2012.

[37] Cédric du Mouza, Witold Litwin, and Philippe Rigaux. Sd-rtree: A scalable distributed rtree. In *ICDE*, pages 296–305. IEEE, 2007.

[38] Ahmed Eldawy. Spatialhadoop: towards flexible and scalable spatial processing using mapreduce. In *Proceedings of the 2014 SIGMOD PhD symposium*, pages 46–50. ACM, 2014.

[39] Francesco Fusco, Marc Ph Stoecklin, and Michail Vlachos. Net-fli: on-the-fly compression, archiving and indexing of streaming network traffic. *Proceedings of the VLDB Endowment*, 3(1-2):1382–1393, 2010.

[40] Shinya Fushimi, Masaru Kitsuregawa, and Hidehiko Tanaka. An overview of the system software of a parallel relational database machine grace. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 209–219. Morgan Kaufmann Publishers Inc., 1986.

[41] Hector Garcia-Molina and Wilburt J. Labio. Efficient snapshot differential algorithms for data warehousing. 1996.

[42] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[43] Yongqiang He, Rubao Lee, Yin Huai, Zheng Shao, Namit Jain, Xiaodong Zhang, and Zhiwei Xu. Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *ICDE*, pages 1199–1208. IEEE, 2011.

[44] Pat Helland. Life beyond distributed transactions: an apostate's opinion. In *CIDR*, pages 132–141, 2007.

[45] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In *Proc. of VLDB*, pages 562–573, 1995.

[46] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, and Ion Stoica. Querying the internet with pier. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 321–332. VLDB Endowment, 2003.

[47] M. Indelicato. Scalability Strategies Primer: Database Sharding. http://goo.gl/51nS0b, December 2008. Retreived: July 2014.

[48] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 59–72. ACM, 2007.

[49] H.V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Quang Hieu Vu, and Rong Zhang. Speeding up search in peer-to-peer networks with a multi-way tree structure. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 1–12. ACM, 2006.

[50] H.V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. idistance: An adaptive b+-tree based indexing method for nearest neighbor search. *ACM Transactions on Database Systems (TODS)*, 30(2):364–397, 2005.

[51] H.V. Jagadish, Beng Chin Ooi, and Quang Hieu Vu. Baton: A balanced tree structure for peer-to-peer networks. In *Proceedings of the 31st international conference on Very large data bases*, pages 661–672. VLDB Endowment, 2005.

[52] Dawei Jiang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Sai Wu. epic: an extensible and scalable system for processing big data. *Proceedings of the VLDB Endowment*, 7(7), 2014.

[53] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. The performance of mapreduce: An in-depth study. *Proceedings of the VLDB Endowment*, 3(1-2):472–483, 2010.

[54] Jeffrey W. Josten, C. Mohan, Inderpal Narang, and James Z. Teng. Db2's use of the coupling facility for data sharing. *IBM Systems Journal*, 36(2):327–351, 1997.

[55] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[56] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[57] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[58] Feng Li, Beng Chin Ooi, M Tamer Özsu, and Sai Wu. Distributed data management using mapreduce. *ACM Computing Surveys (CSUR)*, 46(3):31, 2014.

[59] Yuting Lin, Divyakant Agrawal, Chun Chen, Beng Chin Ooi, and Sai Wu. Llama: leveraging columnar storage for scalable join processing in the mapreduce framework. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 961–972. ACM, 2011.

[60] Bruce G. Lindsay, Laura M Haas, C. Mohan, Paul F. Wilms, and Robert A. Yost. Computation and communication in R*: A distributed database manager. *ACM Transactions on Computer Systems (TOCS)*, 2(1):24–38, 1984.

[61] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[62] Peng Lu, Sai Wu, Lidan Shou, and Kian-Lee Tan. An efficient and compact indexing scheme for large-scale data store. In *ICDE*, pages 326–337. IEEE, 2013.

[63] Samuel Madden, David J. DeWitt, and Michael Stonebraker. Database parallelism choises greatly impact scalability. http://goo.gl/jhQkCn, 2007. Retreived: July 2014.

[64] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[65] MarketsandMarkets. Cloud Analytics Market worth $16.52 Billion by 2018. http://goo.gl/hVeAkw, December 2013. Retreived: July 2014.

[66] Ahmed Metwally and Christos Faloutsos. V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *Proceedings of the VLDB Endowment*, 5(8):704–715, 2012.

[67] Guy M. Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company, 1966.

[68] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, pages 170–177. IEEE Computer Society, 2010.

[69] Wee Siong Ng, Beng Chin Ooi, Kian-Lee Tan, and Aoying Zhou. PeerDB: A P2P-based system for distributed data sharing. In *ICDE*, pages 633–644. IEEE, 2003.

[70] Shoji Nishimura, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Md-hbase: A scalable multi-dimensional data infrastructure for location aware services. In *Proc. of MDM*, 2011.

[71] Alper Okcan and Mirek Riedewald. Processing theta-joins using mapreduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 949–960. ACM, 2011.

[72] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[73] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[74] M. Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems*. Springer, 2011.

[75] Apostolos Papadopoulos and Yannis Manolopoulos. Performance of nearest neighbor queries in r-trees. In *ICDT*, 1997.

[76] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J Abadi, David J DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 165–178. ACM, 2009.

[77] Ali Pinar, Tao Tao, and Hakan Ferhatosmanoglu. Compressing bitmap indices by data reorganization. In *ICDE*, pages 310–321. IEEE, 2005.

[78] Viswanath Poosala and Yannis E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *VLDB*, pages 486–495, 1997.

[79] W. Curtis Preston. *Backup & Recovery: Inexpensive Backup Solutions for Open Systems*. O'Reilly, 2009.

[80] Michael O. Rabin. *Fingerprinting by random polynomials*. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.

[81] E. Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *the VLDB Journal*, 10(4):334–350, 2001.

[82] R. Rawson and J. Gray. HBase at Hadoop World NYC. http://goo.gl/fmvuYk, 2009. Retreived: July 2014.

[83] Denis Rinfret, Patrick O'Neil, and Elizabeth O'Neil. Bit-sliced index arithmetic. In *ACM SIGMOD Record*, volume 30, pages 47–57. ACM, 2001.

[84] Patricia Rodríguez-Gianolli, Anastasios Kementsietsidis, Maddalena Garzetti, Iluju Kiringa, Lei Jiang, Mehedi Masud, Renée J Miller, and John Mylopoulos. Data sharing in the hyperion peer database system. In *Proceedings of the 31st international conference on Very large data bases*, pages 1291–1294. VLDB Endowment, 2005.

[85] James B. Rothnie Jr., Philip A. Bernstein, Stephen Fox, Nathan Goodman, Michael Hammer, Terry A. Landers, Christopher Reeve, David W. Shipman, and Eugene Wong. Introduction to a system for distributed databases (SDD-1). *ACM Transactions on Database Systems (TODS)*, 5(1):1–17, 1980.

[86] Statista. Global Cloud Computing Revenue Worldwide Since 2008. http://goo.gl/1jnZHY, 2014. Retreived: July 2014.

[87] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In

*ACM SIGCOMM Computer Communication Review*, volume 31, pages 149–160. ACM, 2001.

[88] Michael Stonebraker. The case for partial indexes. *ACM SIGMOD Record*, 18(4):4–11, 1989.

[89] Michael Stonebraker, Daniel Abadi, David J. DeWitt, Samuel Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. Mapreduce and parallel dbmss: friends or foes? *Communications of the ACM*, 53(1):64–71, 2010.

[90] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. C-store: a column-oriented dbms. In *Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

[91] Yufei Tao, Jun Zhang, Dimitris Papadias, and Nikos Mamoulis. An efficient cost model for optimization of nearest neighbor search in low and medium dimensional spaces. *IEEE Transactions on Knowledge and Data Engineering*, 16(10):1169–1184, October 2004.

[92] Igor Tatarinov, Zachary Ives, Jayant Madhavan, Alon Halevy, Dan Suciu, Nilesh Dalvi, Xin Luna Dong, Yana Kadiyska, Gerome Miklau, and Peter Mork. The piazza peer data management project. *ACM SIGMOD Record*, 32(3):47–52, 2003.

[93] 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. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[94] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 495–506. ACM, 2010.

[95] Hoang Tam Vo, Chun Chen, and Beng Chin Ooi. Towards elastic transactional cloud storage with range query support. *Proceedings of the VLDB Endowment*, 3(1-2):506–514, 2010.

[96] Werner Vogels. Data access patterns in the amazon.com technology platform. In *Proceedings of the 33rd international conference on Very large data bases*, pages 1–1. VLDB Endowment, 2007.

[97] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[98] Jinbao Wang, Sai Wu, Hong Gao, Jianzhong Li, and Beng Chin Ooi. Indexing multi-dimensional data in a cloud system. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 591–602. ACM, 2010.

[99] Jinbao Wang, Sai Wu, Hong Gao, Jianzhong Li, and Beng Chin Ooi. Indexing multi-dimensional data in a cloud system. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 591–602. ACM, 2010.

[100] Kesheng Wu, Ekow J Otoo, and Arie Shoshani. Compressing bitmap indexes for faster search operations. In *Scientific and Statistical Database Management, 2002. Proceedings. 14th International Conference on*, pages 99–108. IEEE, 2002.

[101] Sai Wu, Dawei Jiang, Beng Chin Ooi, and Kun-Lung Wu. Efficient b-tree based indexing for cloud data processing. *Proceedings of the VLDB Endowment*, 3(1-2):1207–1218, 2010.

[102] Sai Wu, Shouxu Jiang, Beng Chin Ooi, and Kian-Lee Tan. Distributed online aggregations. *Proceedings of the VLDB Endowment*, 2(1):443–454, 2009.

[103] 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 *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 279–290. ACM, 2008.

[104] Sai Wu, Hoang Tam Vo, Kian-Lee Tan, Peng Lu, Dawei Jiang, Tianlei Hu, and Gang Chen. Bestpeer++: A peer-to-peer basedlarge-scale data processing platform. *IEEE Transactions on Knowledge and Data Engineering*, 26(6):1316–1331, 2014.

[105] Sai Wu, Quang Hieu Vu, Jianzhong Li, and Kian-Lee Tan. Adaptive multi-join query processing in pdbms. In *ICDE*, pages 1239–1242. IEEE, 2009.

[106] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.