DESIGN OF EFFICIENT AND ELASTIC STORAGE IN THE CLOUD

VO HOANG TAM

M.Eng. in Computer Science
Ho Chi Minh City University of Technology

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2012

Acknowledgements

I would like to reserve this section to express my sincere gratitude to many people who have provided me invaluable support and encouragement without which I could not have completed this thesis.

Firstly, I am very grateful to my supervisor, Professor Beng Chin Ooi, for taking care of me through my Ph.D. research and teaching me important lessons to be successful in life. Without his excellent guidance in research, I could not have developed a professional way of working and conducting research. I believe the training I received from Professor Ooi as well as School of Computing has placed an important background for my future career and life. I am also privileged to get RAship under his various research projects, which funded me throughout my five years of studies. Besides of being an excellent academic supervisor, he also had a very personal touch with his students. I was happy to be invited to visit his family for every Lunar New Year dinner and we also went to the temple together.

Secondly, I would like to thank Professor Kian-Lee Tan at National University of Singapore, Professor Divyakant Agrawal at University of California, Santa Barbara and Professor M. Tamer Ozsu at University of Waterloo for providing insightful comments on my research works. I have been fortunate to collaborate with them on various works and have learnt precious skills in writing research papers from their guidance. I would also like to thank A/P Chee Yong Chan, A/P Stephane Bressan and the external examiner for participating in my thesis committee and providing helpful comments for me to improve this thesis in terms of both organization and writing.

Thirdly, I would like to thank friendly lab mates in the Database Research Lab at School of Computing – NUS, especially Sai Wu and Dawei Jiang among others. They are technically smart and always willing to help in system hacking and research discussion. In retrospect on my Ph.D. life, it brings back to me lots of good memories for various fun and enjoyable parties we had together to celebrate someone having published a paper in top-tier conferences or achieved an award.

Last but not least, I am very much grateful to my beloved families for their constant encouragement and support throughout my life. I am especially indebted to my mother and my wife for their understanding, care and love through the duration of my studies. I would like to dedicate this thesis to them.

Design of Efficient and Elastic Storage in the Cloud

by

Vo Hoang Tam

Submitted to the School of Computing
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science

ABSTRACT

The cloud simplifies the deployment of large-scale applications by shielding users from the underlying infrastructure and implementation details. It also provides other promising features such as low startup cost, elasticity and pay-as-you-go pricing model. Recently, there have been substantial interests in cloud deployment of data-centric applications, and storage services form a critical component in the software stack provided in the cloud.

Nevertheless, the emerging cloud platforms also present unique challenges for deploying databases and applications in the cloud. Given the large number of end-users and huge amounts of data being generated by applications, coupled with frequent changes in data access pattern, the backend storage system for these applications must be elastically scalable and deployable on clusters of commodity machines while still being able to guarantee data durability and provide highly available data service as well as other important functionalities of a database management system (DBMS) such as transactional semantics for bundled operations, efficient indexes of multiple types and effective support of a variety of workloads.

The ultimate goal of this thesis is to address the aforementioned challenges and propose an efficient and elastic cloud storage service with similar capabilities as centralized database systems. The research in this thesis shows that with careful choices of design, it is possible to develop such an efficient and elastic storage service that provides important DBMS-like features for database applications in the cloud. Specifically, our research advances the current state-of-the-art by introducing three fundamental techniques for cloud data management.

Firstly, we propose `ecStore` – an **e**lastic **c**loud **stor**ag**e** system that can be dynamically deployed on top of cloud virtual infrastructures and support both OLTP and OLAP workloads that run simultaneously and interactively within the same storage. Secondly, we propose a simple but extensible and efficient distributed indexing framework that enables users to define their own indexes without knowing the structure of the underlying network or having to tune the performance by themselves. Thirdly, we propose a load-adaptive replication mechanism to provide both data availability and load balancing functionalities for the system. We also provide transactional semantics for bundled read-modify-write operations spanning across multiple records.

The proposed techniques are evaluated in various cloud environments, including an in-house cluster serving as private cloud, the commercial public cloud Amazon's EC2, and PlanetLab – a testbed representing distributed clouds where machines are geographically located. The experimental results confirm the efficiency, effectiveness and robustness of the system.

Thesis Supervisor: Prof. Ooi Beng Chin

Title: Professor of Computer Science at NUS

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Cloud computing is a step towards the notion that all aspects of computation and IT resources can be organized and provided as a public utility. As industry has started to transit from traditional to cloud-hosted data management, cloud data storage has become one of the most widely acceptable infrastructures [30]. In this chapter, we first start with an introduction of how database applications can benefit from cloud computing model and look especially at challenges of deploying databases in the cloud. Next, we discuss the motivation of our research which aims to provide advanced features missing from current cloud data serving systems and address challenges arising from the convergence of real-time and analytic workloads. Then, we present specific goals and scope of our research. Finally, we give an overview of our solution to the research questions and summarize main contributions of the thesis.

1.1 Database Applications in the Cloud

Figure 1-1 provides an illustration of traditional architecture of web-based database applications. In this architecture, clients work with the applications via web browser interfaces. The web server is responsible to handle requests from the clients, and commonly integrated with an application server which realizes application logics and enforces business constraints. They rely on the underlying database and possibly a file

system to provide data service. This architecture, though offers high flexibility for system development, still suffers from some disadvantages such as single point of failure of the servers at each layer, i.e., application and database/file servers, and limited scalability when the request load from clients exceeds the capacity of the servers. Therefore, the servers are commonly over-provisioned to accommodate the "peak" workload, resulting in high investment and maintenance cost.

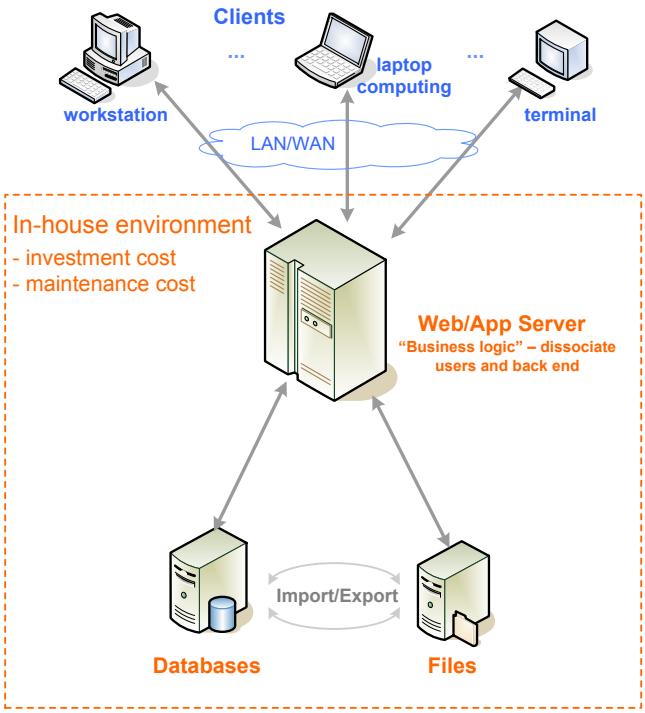


Figure 1-1: Traditional deployment of database applications.

With this conventional deployment of database applications, as the company's business grows it needs to upgrade its hardware capacity on a frequent basis in order to accommodate the increasing workload, which presents many challenges in terms of technical support and cost. Consequently, the revolution of "cloud computing", in which large clusters of commodity processors are exploited to perform various computing tasks with a "pay-as-you-go" model, has become a feasible solution that mitigates the pain. Figure 1-2 depicts the best practice for cloud deployment of database applications. While the web, application, and especially database servers are

the bottleneck in the traditional in-house deployment, these servers now can be deployed on multiple virtual machines leased from the cloud, e.g., Amazon or Rackspace cloud providers [1, 18], and therefore enables the application to elastically scale on demand.

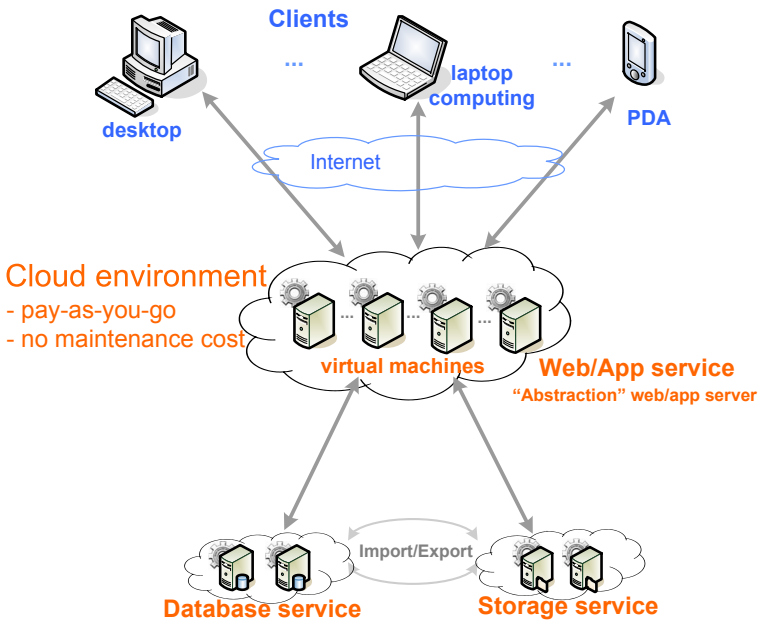


Figure 1-2: Cloud deployment of database applications.

With the fast popularity of cloud computing model, it heralds a new wave of information technology transformation by enabling enterprises to utilize computing power as a service. The cloud is designed to deliver unlimited compute capacity on demand and distinguishes itself from the other system architectures and computing models in the aspect of scalability and elasticity. For many social networking sites, e.g., Foursquare[1] and Quora[2], the cloud is an ideal platform for accommodating their rapid increase in terms of data size, end-users, and applications.

Similarly, it is also ideal for database centric applications where occasional surge in demand for processing capacity is encountered. One good example application is Customer Relationship Management (CRM)[3], which is used to monitor sales activities,

[1]https://foursquare.com/
[2]http://www.quora.com/
[3]http://www.salesforce.com/

and improve sales and customer relationships. While there are daily account maintenance and sales activities, there are certain periods when sales quota must be met, forecasting and analysis are required, etc., and these activities require more resources at peak periods, and the cloud is able to meet such dynamism of resource requirements.

1.1.1 Challenges of Deploying Databases in the Cloud

There have been two advocated approaches to the deployment of database systems in the cloud as of now:

- Install a clustered database system on the virtual machines, e.g., MySQL used in Amazon's RDS [3] and SQL Server used in Microsoft SQL Azure [41, 45].

- Employ a NoSQL storage system [16] that is specially designed for cloud environments and specific applications.

The former approach provides full functionalities of a traditional database management system in the cloud, but these systems are hard to scale and not designed to run on low-end machines [22, 90, 51]. The technologies adopted by most traditional parallel databases cannot be applied directly to cloud data management systems due to the elasticity characteristic of the new environment.

Specifically, unlike traditional distributed environments which commonly comprise of a fairly static and small number of high-end machines, in the cloud a dynamically large number of low-end machines are deployed to process massive datasets, and more importantly, the demand for resources may vary drastically from time to time due to changes in the application workload. Since traditional parallel database systems are mainly designed and optimized for fairly static clusters, they cannot take full advantages of the cloud as users desire to economically and elastically allocate resources from the cloud based on load characteristics.

On the contrary, NoSQL storage systems [16] developed following the latter approach provide the essential elastic scalability for systems to be deployed in the cloud. However, while it is desirable to provide efficient and elastic cloud storage services with similar functionalities offered by traditional centralized database systems, current cloud data serving systems, as surveyed in [47], still lack of important features such as smart replication, transactional semantics and especially DBMS-like index mechanism, which motivates our research.

1.2 Motivation

Our research is motivated by the facts that there is an emerging trend of the convergence of real-time and analytic workloads as observed in [129, 42, 21, 78], and while current data serving systems provide the needed scalability for specific applications they still lack important features for database applications in the cloud [47].

1.2.1 Convergence of Real-time and Analytic Workload

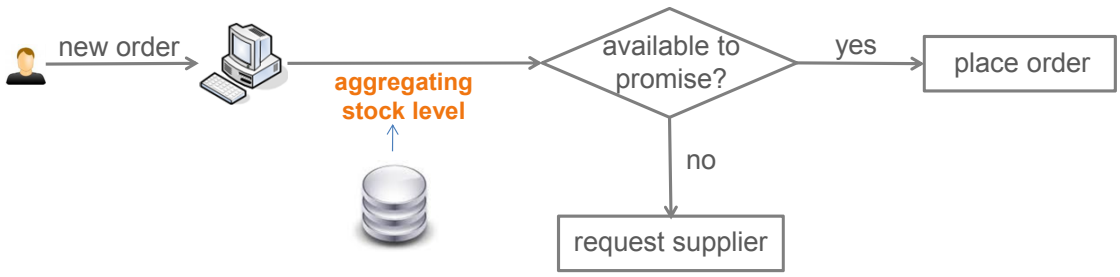


Figure 1-3: Convergence of OLTP and OLAP: real-time analysis application.

From the application point-of-view. The convergence of real-time and analytic workloads, commonly referred to as online transaction processing (OLTP) and online analytical processing (OLAP), arises in many application scenarios. For example, in online business applications, most transactional decisions will be preceded by a detailed analysis. Figure 1-3 illustrates that the decision whether to promise a new purchase

order from a customer is dependent on a real-time aggregating of stock levels. Therefore, it is preferable to perform analysis queries directly on the transactional data for up-to-date results.

The convergence of real-time and analytic workload is also observed in the scenario of financial and capital markets, where the application maintains a large amount of real-time event streams and needs to perform analytics on historical data and feed the analytical model back into the application for end-users' information. Experiences from Yahoo! also show that many interesting web applications do not fit neatly into either data serving or batch processing paradigm [129]. Application scenarios that benefit from the combination of OLTP and OLAP include Web 2.0 applications, social network sites, etc. To better support search and data sharing, large-scale ad-hoc analytical processing on the data collected from those web applications is becoming increasingly valuable to improving the quality and efficiency of existing services, and supporting new functional features.

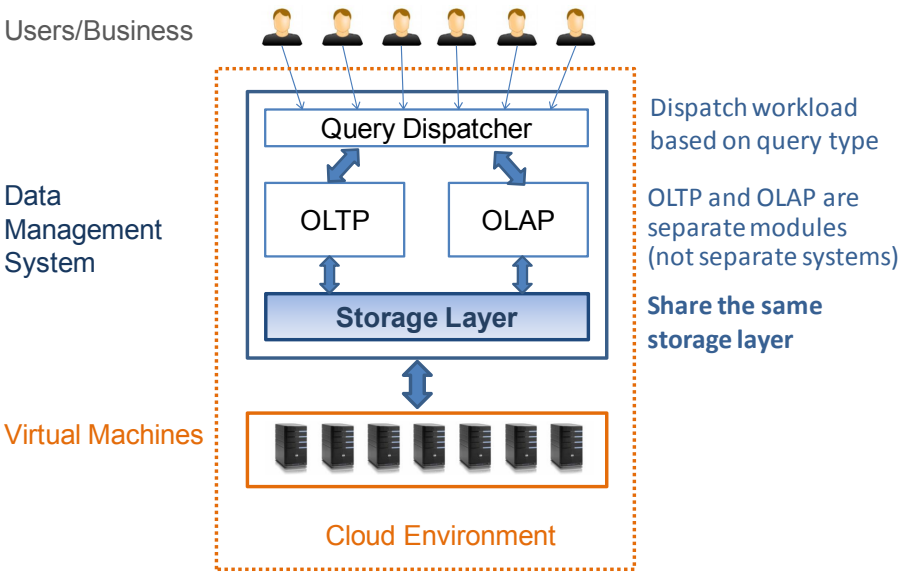


Figure 1-4: Convergence of OLTP and OLAP: from infrastructure point-of-view.

From the infrastructure point-of-view. Traditionally, real-time and analytic workloads are often handled independently by separate systems with different

architectures, namely relational database management system (RDBMS) for OLTP and data warehousing system for OLAP. To maintain the data freshness between these two systems, a data extraction process (a.k.a. ETL) is periodically performed to transform and load the data from the RDBMS into the data warehouse for further analysis. This system-level separation, though provides flexibility and the required efficiency, introduces several limitations such as lack of up-to-date data freshness for OLAP, redundancy of data storage as well as high startup and maintenance cost.

The need to dynamically provide for capacity in terms of storage and computation, and to support OLTP and OLAP in the cloud demands the re-examination of existing data servers and architecting possibly "new" elastic and efficient data servers for cloud data management service. In other words, with the fast popularity of cloud infrastructures, it is timely and desirable to have an integrated system that provides both high-performance OLTP and OLAP capabilities. In this architecture, as depicted in Figure 1-4, OLTP and OLAP are now separate modules of a single system instead of being separate systems traditionally. Since these two modules share the same storage layer, it is possible for OLAP to perform on the latest data that are being manipulated by OLTP operations and provide timely analytic insights on the data. This architecture therefore enables new breed of real-time analysis applications.

Not surprisingly, main-memory resident database systems that handle both OLTP and OLAP have recently been proposed [115, 78, 89]. For cloud environments, DataStax, an IT company for cloud technology, has proposed to unify Hadoop MapReduce [14] and Cassandra [93] for supporting both real-time and analytic workloads [21].

1.2.2 Missing Features of Cloud Data Serving Systems

The design and development of our proposed cloud storage system is also motivated by the fact that current closed-source data serving systems (such as Dynamo [61] and Pnuts [54]) and open-source data serving systems (such as HBase [6] and Cassandra [93]) do not support *transactional semantics* for a collection of reads and writes spanning across

multiple records. More recently, systems such as MegaStore [37] and ElasTraS [57] have started to provide transaction support for cloud storages.

It is also noteworthy that most of these systems such as Cassandra and Pnuts employ data migration to balance the storage load of the servers. However, under skewed query distributions, it is critical to balance the query execution load across servers as well, which drives the design of a *load-adaptive replication* technique used in our proposed storage system.

More importantly, while it is desirable that the cloud should provide efficient and scalable storage services with similar functionalities offered by centralized database systems for better support of data-centric applications, the provisioning of *DBMS-like index functionality* is a missing feature in current cloud data serving systems. One obvious requirement for this functionality is to locate some specific records among millions of distributed candidates in real-time, preferably within a few milliseconds.

It is also important that the system supports multiple indexes over the distributed data, including primary and secondary indexes, which is a common service in any DBMS. The last but not least requirement is extensibility by which users can define new indexes without knowing the structure of the underlying network or having to tune the system performance by themselves. Currently no cloud data serving system satisfies these requirements.

1.3 Research Goals and Scope

Given the call for integrating OLTP and OLAP from both infrastructure and application point-of-view, coupled with the aforementioned missing features of current cloud data serving systems, our ultimate research goal is to build an efficient and elastic storage system that can be dynamically deployed on cloud virtual infrastructures and provide advanced features for database applications in the cloud, including the ability to support a variety of workloads, automatic load balancing, transactional semantics, and efficient

indexing, as its intrinsic properties in order to deal with the scale, elasticity and load dynamism that characterize the cloud environment and its applications.

The thesis focuses on the following research lines:

1. Hybrid Storage – the design of storage-level support of a combined OLTP and OLAP workload.

2. Load Balancing – the capability of automatic load balancing in the presence of workload dynamism.

3. Consistency Management – the management of replica consistency and transaction consistency, and the interplay between the two.

4. Distributed Indexing – the design of a comprehensive and efficient framework for providing DBMS-like indexes in the cloud.

In this thesis, we mainly describe the design and implementation of `ecStore`, the storage manager of a bigger cloud data management system named `epiC` [12, 51], and provide fundamental results and initial work towards the building of an efficient and elastic cloud storage system. The main features of `ecStore` include flexible hybrid data partitioning for supporting both OLTP and OLAP workloads, smart replication for data availability and automatic load balancing, transactional semantics and distributed indexing. As will be presented in more depth in Section 4.1, the processing and optimization of OLAP and OLTP queries – which is handled by upper layer query processing engines of `epiC`, i.e., the OLAP and OLTP controller [51, 146] – will ride on the basic functionalities provided by `ecStore`, and consequently is beyond the scope of this research.

1.4 Solution Overview

In this research, we develop `ecStore` – an **e**lastic **c**loud **s**t**o**r**a**g**e** system that can be dynamically deployed in clusters of commodity machines located in the cloud while still

being able to guarantee data durability and provide highly available data service as well as other important functionalities of a centralized database system.

`ecStore` is designed as a stratum architecture. At the lowest level, it develops a *generalized partitioned data structure* to decluster data records across storage nodes in order to facilitate parallelism and improve system performance in terms of both throughput and response time. In particular, it employs a generic peer-to-peer (P2P) overlay network based on Cayley graph model [34] to efficiently support multiple distributed data structures of different types such as DHT-based structures (e.g., Chord [130]), tree-based structures (e.g., BATON [86]) and multi-dimensional structures (e.g., CAN [122]).

These distributed data structures could automatically repartition and redistribute the data when machines are added into or removed from the system via online migration of data between adjacent storage nodes. This property is desirable since an *elastic* cloud storage should allow users to scale out and scale back on the fly based on load characteristics. Furthermore, in order to support the combined OLTP and OLAP workload, `ecStore` exploits the trace of queries in the workload and devises a *hybrid data partitioning* scheme that favors both workloads with a careful design of vertical and horizontal partitioning.

In the middle tier, we leverage on the underlying generalized partitioned data structure to support smart replication and provide both data availability and load balancing for the system. Here, we extend the Cayley graph-based data structures to effectively support *load-adaptive replication* for large-scale environments. The idea of replicating hot data to resolve skewed access patterns is common; however, previous works on replication for load balancing in conventional distributed systems [83, 144, 143] as well as P2P systems [73, 138] maintain the query access statistics on the granularity of data objects. This approach is impractical when the amount of data in the system is large, especially for cloud-scale databases. By the use of self-tuning range histograms, `ecStore` can efficiently deal with skewed access patterns while creating

only a small number of replicas (thus reducing storage cost and replica consistency management cost) and keeping the cost of histogram maintenance minimal. In addition, we develop a *simple but extensible and efficient indexing framework* that enables users to define their own indexes without knowing the structure of the underlying network. The indexing framework is also designed to ensure the efficiency of hopping between cluster nodes during index traversal, and reduce the maintenance cost of indexes.

Finally, in the topmost tier, we develop a multi-version optimistic concurrency control scheme. While multi-versioning enhances the performance of read-dominant applications, the use of optimistic concurrency control takes advantage of emerging applications where users typically access mutually exclusive data. Further, a complete method for system recovery in `ecStore` guarantees the requirement of data durability, which is an essential service level agreement (SLA) of cloud storages when deployed on virtual infrastructures. Additionally, the data access optimizer of `ecStore`, which also stays in this tier, dynamically chooses the best data access plan, namely parallel sequential scan or index scan, for a specific data access request by the use of a cost-based optimization algorithm that utilizes the statistics information maintained in the metadata catalog of the system.

1.5 Contributions

The research in this thesis makes several fundamental contributions towards providing scalable "database as a service" in the cloud. Particularly, we design and develop an elastic storage system that provides important features for supporting database applications in the cloud, including storage-level support for both OLTP and OLAP workloads [46], a load-adaptive replication scheme and transactional semantics for bundled reads and writes spanning across multiple records [139], and a comprehensive framework for supporting indexes in the cloud [53]. Figure 1-5 summarizes these contributions into three major areas of the thesis. We now highlight these contributions and their impact in the following.

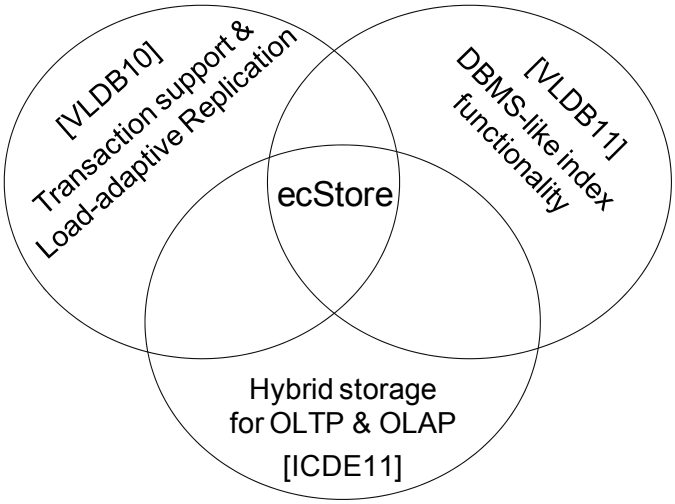


Figure 1-5: Overview of contributions.

A Hybrid Storage for Supporting both OLTP and OLAP Workload [46]. We propose a new system architecture for supporting database operations in cloud systems spanning clusters of commodity servers where machines can be dynamically added into or removed from the system based on load characteristics. `ecStore` – our proposed **e**lastic **c**loud **stor**ag**e** system – is designed to support a combined OLTP and OLAP workload efficiently with a flexible data partitioning scheme to favor both workloads and an effective cost-based data access optimizer to choose near optimal data access plans. The system also provides load-adaptive replication, efficient distributed indexes and transactional access across multiple records, which are important features but missing from most cloud data serving systems.

Generalized Distributed Indexing in the Cloud [53]. As in conventional DBMSes, indexes incur maintenance overhead and the problem is more complex in distributed environments since the data are typically partitioned and distributed based on a subset of attributes. Furthermore, the distribution of indexes is not straight forward, and there is therefore always the question of scalability, in terms of data volume, network size, and number of indexes. `ecStore` pioneers the

provision of DBMS-like index functionality in the cloud. We propose a simple but extensible and efficient indexing framework that enables users to define their own indexes without knowing the structure of the underlying network or having to tune the index performance by themselves while ensuring the efficiency of hopping between cluster nodes during index traversal and reducing the maintenance cost of indexes.

Load-adaptive Replication and Transactional Support for Cloud Storages [139].
We provide transactional semantics for bundled read-modify-write operations spanning across multiple records in `ecStore`. We also provide high resilience capability with smart data replication and a complete method for system recovery in order to meet the data durability requirement, an essential service level agreement (SLA) of cloud storages when deployed on virtual infrastructures. In addition, we propose a two-tier partial replication strategy, which is adaptive with the database workload at runtime, in order to guarantee effective load balancing in the system under skewed data access patterns.

1.6 Outline of the Thesis

The thesis is organized as follows.

- Chapter 2 gives background information that forms the basis of our research.
- Chapter 3 presents a literature review on related works in the field.
- Chapter 4 describes the design and implementation of `ecStore` – our proposed elastic cloud storage system that supports both OLTP and OLAP workloads.
- Chapter 5 presents the generalized distributed indexing framework developed in `ecStore` to provide DBMS-like index functionality in the cloud.

- Chapter 6 describes `ecStore`'s load-adaptive replication scheme and transactional support for bundled read-modify-write operations.
- Chapter 7 provides an extensive performance study of `ecStore`.
- Chapter 8 summarizes the research contributions of this thesis and indicates our future work.

Chapter 2

Background

In this chapter, we present background information for our research. In order to gain a better understanding of cloud systems, we examine various concepts of cloud computing and look especially at cloud computing model from data management perspective. We also discuss basic techniques for replication management and review peer-to-peer (P2P) overlay networks that are commonly used to facilitate distributed search.

2.1 Cloud Computing Concepts

While cloud computing has gained fast popularity, users might get overwhelmed with a variety of taxonomy such as cloud platform, software as a service (SaaS), 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. We also present an overview of the transition from traditional to cloud platform.

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

2.1.1 Cloud Computing: Definition & Characteristics

Definition of Cloud Computing

Cloud computing is gaining fast popularity and technology providers tend to have different definitions of cloud computing. In response to this situation, some standard organizations, such as the U.S. Government's National Institute of Standards and Technology (NIST), have proposed to standardize the definition of cloud computing as "a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" [15].

Characteristics of Cloud Computing

As of now, there exists no consensus on the exact definition of cloud computing; however, it possesses several characteristics that are commonly agreed by the industry and users community. In an attempt to standardize the cloud computing concepts [15], NIST provides a description of five essential characteristics of cloud computing.

Rapid Elasticity: The elasticity aspect of cloud computing represents its most promising feature for the ability to scale out and scale back the resources based on needs. From the consumers' point of view, the cloud provides infinite resources, and they can purchase the computing power from the cloud like other utility services and are billed on a pay-per-use model. Elasticity is the characteristic that differentiates cloud computing from grid computing most [11].

Measured Service: The cloud service provider must constantly monitor all aspects of its service in order to guarantee service level agreements (SLA) with customers. This characteristic is also important for various tasks in the cloud such as capacity planning, resource optimization, billing service, and access control.

On-Demand Self-Service: This characteristic allows customers to acquire their needed resources from cloud services in an automated fashion, without having to go through tedious interaction with the cloud provider to perform necessary configuration.

Ubiquitous Network Access: The cloud resources such as storage capacity and computation are provisioned over the network, either on in-house infrastructures (private cloud) or remotely on the internet (public cloud). End-users access these resources through standard methods such as web service interfaces regardless of the type of network.

Location-Independent Resource Pooling: This characteristic allows for *multi-tenant* model, i.e., supporting a large number of customers while ensuring efficient resource utilization. Resources are assigned to consumers based on load and need. The consumers are shielded from implementation details of the underneath cloud infrastructure and do not know the location of the physical resources.

2.1.2 Cloud Architectural Service Layers

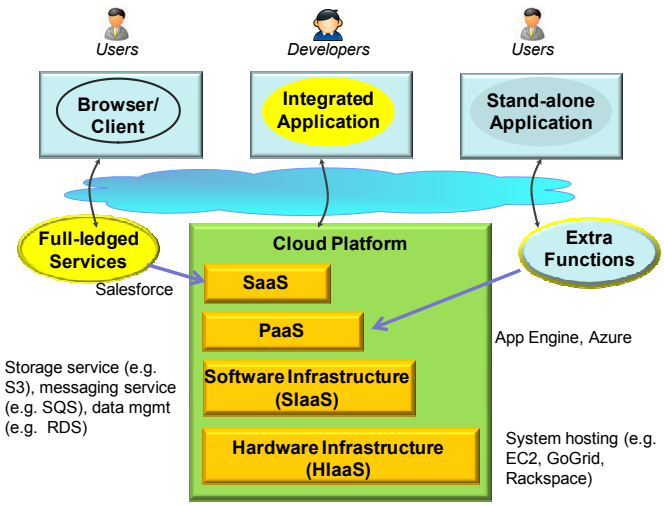


Figure 2-1: Architectural service layer in the cloud.

As discussed above, cloud computing represents a new way of delivering IT resources as utility services in that these resources, for examples, packaged

applications, computational power and storage capacities are provisioned as a remote billed service. Figure 2-1 provides an illustration of the architectural service layer in the cloud consisting of three major categories, namely Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS).

IaaS, which provides users with the access to hardware infrastructure (HIaaS) such as virtual machines and persistent data stores, or software infrastructure (SIaaS) such as messaging services, is the most general form of cloud services. The services are typically billed with the pay-as-you-go model, i.e., based on the amount of consumed resources. Compared to IaaS, PaaS provides a higher-level platform, such as storage and database services, for developers to write applications, and thus hiding the low-level infrastructure from the users. SaaS, the highest form in the cloud service stack, delivers special-purpose software through the Internet. The software offered by SaaS are completely maintained by the service provider, and therefore the customers of SaaS are free from the burden of managing servers, maintaining and upgrading software.

2.1.3 Transition from Traditional to Cloud Platform

In [50], the author provides an overview on the transition from traditional to cloud platform and presents in detail about components of a cloud platform, which is one of the three major categories of cloud services (see above) and provides platform as a service (PaaS).

A platform for developing application typically consists of three main parts including the foundation, infrastructure services and application services. In the context of *traditional platform*, the foundation could be operating system and local support such as .Net framework and J2EE. The conventional infrastructure services could be database technologies (such as MySQL, PostgreSQL, and InterBase), and identity service for distributed applications. The traditional application services vary from packaged applications (such as SAP and Oracle suite) to customized applications developed in-house.

When it comes to the context of *cloud platform*, the above three components should evolve to its cloud version. More specifically, for the *cloud foundation* component, the provision of customer-specific instances of virtual machines is essential and Amazon Elastic Compute Cloud (EC2) [2] is probably the most well-known operation system in this aspect. For *cloud infrastructure services* component, *cloud storages* are increasingly attractive for applications which require elastically scalable and cost efficient data store. Basic unstructured remote storages, for example, Amazon Simple Storage Service (S3) [4], represent common cloud storage services that are used by the industry and users community. Another example in this aspect is the provision of structured cloud storages such as Microsoft's SQL Server Data Services [45]. Regarding to *cloud application services* component, some utilities provided in the cloud such as search service, mapping and photo galleries have made it easier to create mash-up Web 2.0 applications.

2.2 Cloud Computing: From Data Management Perspective

We now study cloud computing concepts from the data management perspective. Specifically, we first present the desired properties of a cloud data management system. Then, we discuss the gap between relational databases and the cloud, and finally review cloud-based data management solutions bridging the gap.

2.2.1 Desired Properties of a Cloud Data Management System

To utilize the cloud economies effectively, cloud data management systems are desired to provide the following features [51, 27].

Scalability: In today "information explosion era", the amount of data generated by mankind has increased exponentially. To process such huge amount of data within a reasonable time, a large number of compute nodes are required.

Consequently, a cloud data management system must be able to deploy on very large clusters (hundreds or even thousands of nodes) without much problems.

Elasticity: Elasticity is an invaluable feature provided by the cloud. The ability of scaling resource requirements on demand results in a huge cost saving and is extremely attractive to any operations when the cost is a concern. To unleash the power of the cloud, a data management system should be able to transparently manage and utilize the elastic computing resources. That is, the system should allow users to add and remove compute nodes on the fly. Ideally, to speed up the data processing, one can simply add more nodes to the cluster and the newly added nodes can be utilized by the data processing system immediately (i.e., the startup cost is negligible). In contrast, when the workload is light, one can release some nodes back to the cloud and the cluster shrinking process will not affect other running jobs such as causing them to abort.

Fault-tolerance: The cloud is often built on a large number of low-end machines. As a result, hardware failures are fairly common rather than exceptional. A cloud data management system should be highly resilient to node failures. Single or even a number of node failures should not affect data availability and data reliability, or cause the data processing to restart the running jobs.

Self-manageability: In principle, more machines leased from the cloud can be allocated to improve the performance of a cloud data management system. However, this solution is not cost effective in a pay-as-you-go environment and may potentially offset the benefit of elasticity. In order to maximize cost savings, a cloud data management system should be able to self-tune and optimize its performance given the allocated resources rather than running a large number of light-loaded machines.

2.2.2 Bridging the Gap between Parallel and Cloud Databases

The Gap between Parallel and Cloud Databases. Cloud computing model provides an abstraction of traditional server hosting solutions, where users can lease virtual machines from service providers and deploy applications on these machines which could be organized into a cluster following a shared-storage or shared-nothing architecture [131]. Consequently, conventional distributed and parallel database technologies form the basis of the design and implementation of cloud-based data management systems.

DeWitt and Gray [63] present a thorough review on the techniques used by various research and commercial parallel database systems. Parallel database systems have their roots from the middle of 1980s with pioneer Gamma [62] and Grace [67] projects. The parallel database technologies offered by vendors such as Teradata, Netezza and Vertica, are typically small or medium-size clustered deployment of a database management system that provides an environment for users to perform an analytical query via internal support of parallel query processing.

Most parallel database systems employ two-phase locking for concurrency control and write-ahead logging scheme for recovery control. However, traditional parallel database systems are initially designed and optimized for stable systems with a fairly static number of machines, and hence fall short of scaling dynamically with load and need. They are not 100% fit for a scalable storage which needs to elastically scale on demand with minimal overheads.

That is, although parallel database systems can be deployed in cloud environment, they are not able to exploit the built-in elasticity feature of the cloud which is important for startups, small and medium sized businesses. Since parallel database systems are mainly designed for static clusters of high-end servers, the inflexibility of dynamically growing up and shrinking down the clusters of commodity machines based on load characteristics limits their elasticity and suitability for the pay-as-you-go model in cloud environments.

Fault tolerance is another issue of parallel database systems when deployed in the new environment. Historically, it is assumed that node failures are uncommon in small clusters, and therefore fault tolerance is often provided for transactions only. The entire query must be restarted when a node fails during the query execution. This strategy may cause parallel database systems not being able to process long running queries on clusters with thousands of nodes, since in these clusters hardware failures are common rather than exceptional.

Nevertheless, it is noteworthy that many design principles of parallel database systems such as indexing techniques, horizontal data partitioning, partitioned execution, cost-based query optimization and declarative query support, could form the foundation for the design of systems to be deployed in the cloud.

Bridging the Gap between Parallel and Cloud Databases. As discussed above, traditional parallel database systems are initially designed for stable systems with a fairly static number of machines, and therefore fall short of scaling dynamically with load and need. MapReduce, a state-of-the-art processing model for dynamic cluster environments, is first introduced by Dean and Ghemawat [60] to simplify the building of web-scale inverted indexes. The framework has been employed to process filtering-aggregation data analysis tasks as well [114]. It is also possible to evaluate more complex data analytical tasks, by executing a chain of MapReduce jobs [113].

MapReduce systems have several advantages over parallel database systems. First, MapReduce is a pure data processing engine, enabling MapReduce and the underlying storage system to scale independently and match well with the pay-as-you-go model. Second, map tasks and reduce tasks are assigned to available nodes on demand and users can dynamically increase or decrease the size of the cluster without interrupting the running jobs. Third, map tasks and reduce tasks are independently executed from each other, enabling MapReduce to be highly resilient to node failures. When a single node fails during the execution of a job, only map tasks and/or reduce tasks on the failed node need to be restarted, but not the entire job.

Nevertheless, Hadoop [14], an open-source equivalent of MapReduce, has been noted to suffer from sub-optimal performance in the database context [113, 132]. For example, Jiang et al. [87] have identified five design factors that affect the performance of Hadoop MapReduce including block-level scheduling, grouping functions, record parsing, indexing utilization, and I/O modes. Another research proposal, Hadoop++ [64], aims to optimize the performance of Hadoop via exploitation of indexes. It reduces overall I/O cost by utilizing local indexes on the inputs of map tasks, but lacks a global index to reduce the number of map tasks.

More recently, some commercial parallel database systems have started to integrate the MapReduce framework into their execution engines in order to implement user-defined functions which lack efficient support in conventional parallel database systems. Aster [66] and Greenplum [13] have showed that the combination of MapReduce and other relational operators can improve the performance of processing analytic queries in the system. In the another approach, HadoopDB [24] combines single node databases and Hadoop, i.e., data are loaded from Hadoop distributed file system (HDFS) [7] into a cluster of local databases for processing in Hadoop, to utilize sophisticated technologies from database community while leveraging dynamic scalability and fault-tolerance of MapReduce.

2.3 Replication Management

Replication is important in distributed environments since it ensures data availability. To guarantee the transparency of replication to users, conventional replication techniques follow pessimistic approach to provide single-copy consistency, i.e., the users work on replicated data as if there were only one copy of data. A typical method of this approach is voting scheme [71]. Under this scheme, to read the replicated data a read quorum of r votes must be collected to ensure that the latest copy of the replicated data is included. Similarly, for write operations, the system needs to collect at least w votes for the write

quorum to guarantee that the new value of the replicated data has been updated at a minimum number of replicas.

Furthermore, to ensure that the latest version of the replicated data can always be returned for any read operation, there should be an overlap between the two quorums. Therefore, the sizes of read and write quorums are commonly chosen in such a way that their summation, i.e., $(r + w)$, is greater than the total number of copies of the replicated data in the system. The performance of a quorum system can be controlled by appropriately configuring the sizes of read and write quorums. For example, in a read-dominated environment, we can set r to 1 and w to the total number of replicas. This setting is often referred to as the ROWA (Read One Write All) scheme. Under this scheme, read operations can be served quickly by just returning any replica while write operations will suffer from a high latency due to the waiting time for all write votes.

Pessimistic replication provides acceptable throughput and availability in tightly-coupled environments such as local-area networks with low communication latency and low rate of failures. However, problems will arise when pessimistic replication is applied to wide-area environments or application contexts in which high throughput is desirable and replica consistency could be compromised to some certain level. Moreover, the system throughput and data availability of a replicated system which guarantees strict consistency between replicas are considerably affected when the number of replicas in the system increases.

On the contrary, optimistic replication, which propagates updates to replicas asynchronously, is more promising for systems that are deployed in large-scale environments while demanding high throughput and low latency. Optimistic replication techniques are used in many real-world scenarios such as replication in domain name systems (DNS), wide-area information exchange (e.g., Usenet), and software version control (e.g., CVS).

Nonetheless, building an optimistic replication system faces the following key challenges: how to order update operations, which replicas can submit updates, how to

exchange update between sites, how to define and handle conflicts, and how the system guarantees the upper bound of divergence of replicas. The survey paper [123] discusses various techniques that have been developed to address the above challenges. Note that the advantages of optimistic replication come with the cost of complicated implementation, and therefore unsatisfactory design may result in system ill-performance.

Between the line of pessimistic and optimistic replication is adaptive consistency guarantee, which is proposed in [105]. In this system, version vectors are exchanged between replicas so that replica inconsistencies can be detected. Extended version vectors are proposed to make it possible for the system to quantify the current consistency level of the replicated data and trigger a reconciliation process when necessary. Users could specify their desirable consistency level in advance and the system guarantees that the obtained consistency is always above the desired consistency level. The users could also adjust the consistency requirement at runtime when their needs change.

2.4 P2P Overlays for Distributed Search

Structured peer-to-peer (P2P) overlays are designed for efficient support of distributed search, and hence naturally make themselves good candidates for underlying network structures of cloud environments. For example, both Amazon's Dynamo [61] and Facebook's Cassandra [93] exploit a P2P-based architecture, specifically a distributed hash table similar to Chord structure [130], to build their highly available storages with no single point of failure. Consequently, in this section, we review common overlay networks that will be employed in our system, including Chord [130] for distributed hash indexes, CAN [122] for distributed multi-dimensional indexes and BATON [86] for distributed range indexes.

2.4.1 Chord

Chord [130] is built on a ring structure that stores key/value pairs for distributed data items and provides decentralized P2P lookup protocol on the structure. Figure 2-2 provides an example of the Chord ring. In Chord, each node and data item is mapped to a m-bit identifier by applying a universal hashing function. Each node with identifier p has close relationships with two adjacent nodes on the ring, namely its successor, denoted as $successor(p)$, which is the node next to it in the clockwise direction, and its predecessor, denoted as $predecessor(p)$, which is the node next to it in the anti-clockwise direction. The node p is responsible to manage all data with hashed k between the identifier of its predecessor and its identifier, i.e., $k \in [predecessor(p), p]$. For routing purpose, each node p maintains a list of adjacent nodes with identifier p_i such that $p_i = successor(p + 2^{i-1})$ in its finger table.

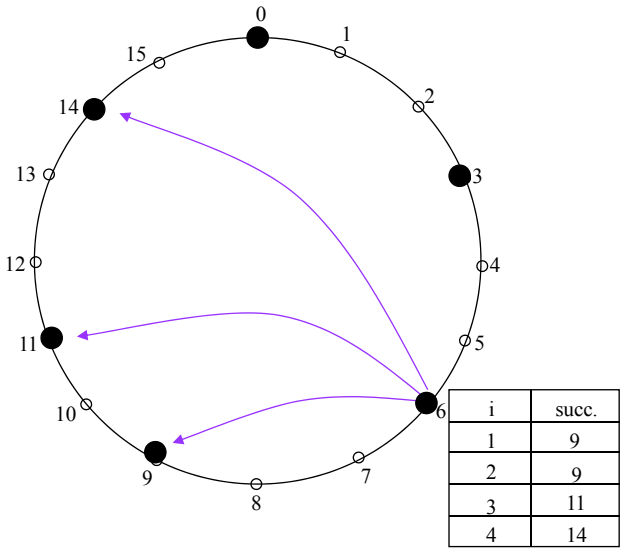


Figure 2-2: The structure of Chord.

Users can start query for a record from any node in the system, say n. The search key in the query is hashed into key k, which is used by Chord protocol to perform the search process. Node n checks its finger table to determine node j which has an identifier most immediately preceding k. The query will be routed to the node j which will in its turn to identify the next node having an identifier that is closest to k. The process is repeated

until key k is located. For each hop, the distance between the target and the current nodes in the Chord ring will decrease by half. Consequently, the average routing complexity of Chord in a network of N nodes is $O(logN)$ search hops.

2.4.2 CAN – Content Addressable Network

CAN [122] is a structured overlays based on a virtual d-dimensional Cartesian coordinate space. Each node in CAN keeps track of the information of its neighbors in each dimension, and hence its routing table consists of $2d$ neighbors. The routing information includes IP address of the neighbor nodes and their responsible zones in the network. Figure 2-3 shows an example of a CAN system with 8 nodes in a two dimensional space.

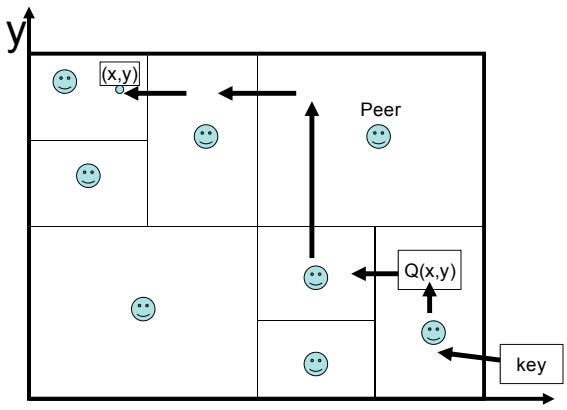


Figure 2-3: The structure of CAN.

CAN uses a hash function for each dimension and maps each data item (key, $value$) into a point p in the coordinate space. The node whose zone includes point p is responsible to maintain the corresponding (key, $value$) data item. In the search process given a key, the starting node applies the same hash functions to generate the destination point p in the coordinate space. At each step of the routing process, the query is forwarded to the neighbor that is closest to the target until it finally reaches the node responsible for the zone containing the data. The average routing complexity of a d-dimensional CAN network of N nodes is $O(d \cdot N^{1/d})$ search hops.

2.4.3 BATON – BAlanced Tree Overlay Network

BATON [86] organizes storage nodes into a binary tree structure in which each node of the tree represents one storage node. The tree is always kept height-balanced so that the height difference of any directed sub-trees of a tree node is at most one. Each node in the tree is responsible to manage a range of values, which are smaller than that of its right adjacent and greater than that of its left adjacent. Consequently, the data maintained in a BATON structure are in increasing order if we traverse the tree starting from the left-most node and following adjacent links.

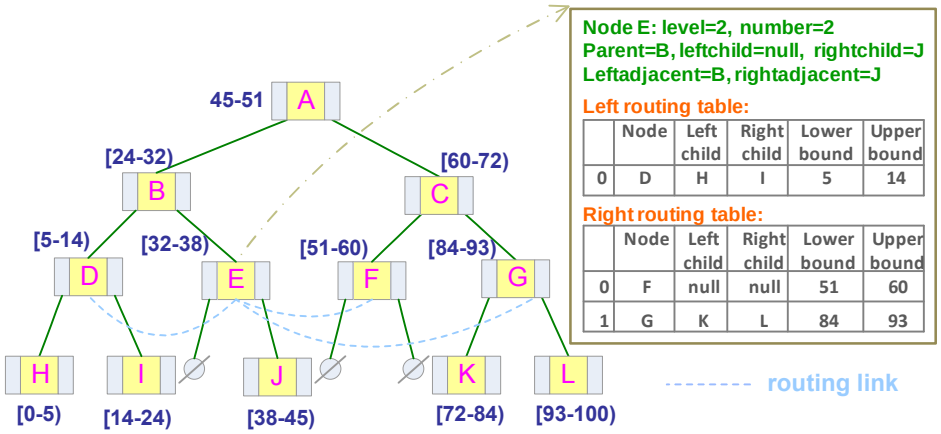


Figure 2-4: The structure of BATON.

Although BATON is a tree-based structure, the search can be initiated at any node without having to search from the root node. To support searching over the overlay, each BATON node keeps the pointer (IP address) and the range index of its parent node, child nodes and adjacent nodes (the left and right adjacent nodes in key order). Additionally, it also maintains the routing pointers to the nodes at a distance of power of two on the same level. Figure 2-4 depicts an example of BATON overlay and the routing table maintained at each node. The height-balance characteristics and the routing links of BATON guarantee that an exact match query can be performed in $O(\log N)$ search hops in a network of N storage nodes.

One major advantage of BATON over other existing DHT-based overlays (distributed hash tables) is that it can support range queries efficiently while being able

to keep storage loads balanced across nodes in the system. To process a range query, BATON first issues an exact match query for the lower bound of the query range to locate the node managing the start of the range. Then the search process continually follows the right adjacent links to retrieve the data objects qualifying the range until meeting the upper bound of range query.

Moreover, BATON is able to balance the system load by redistributing the load via data migration. Each node is required to continuously monitor its local workload and estimate the average load of the whole network. If its local load exceeds the average load with respect to a certain threshold, a balancing process is invoked. To share the load of a heavily-loaded node, a lightly-loaded node in the system is forced to leave the network and rejoin as a child of that overloaded node. The height of the tree may become imbalanced due to this process, and a network restructuring operation similar to rotation in AVL tree is performed to keep the tree height-balanced.

2.4.4 Providing $O(1)$ Search Hop Latency

It is challenging to deploy a P2P structured overlay such as CAN [122], Chord [130] and BATON [86] as a distributed data structure in a cloud cluster because the latency of multiple search hops might affect the performance of query processing. Therefore, we propose to cache direct routing information at each node in the system. The search process can try these auxiliary pointers before using standard routing tables. If the cached pointers are correct, the search process will succeed with only one search hop. Since cluster environments are typically more stable, i.e., experience lower churn rate than pure P2P systems on wide-area networks, maintaining the consistency of the routing cache is not expensive. In particular, we use ping messages to periodically validate and update these auxiliary cached pointers. It is noteworthy that P2P-based cloud storages such as Dynamo [61] and Cassandra [93] also use gossip-based protocol to disseminate the routing information of all nodes in the system in order to support $O(1)$ search hop latency.

2.5 Summary

In this chapter, we have given background information that forms the basis of our research. Specifically, we have presented various concepts of cloud computing model and data management perspective of cloud computing. In addition, basic techniques for replication management and P2P overlay networks commonly used for distributed search have also been discussed. In the next chapter, we shall review related work and current state-of-the-art of our research.

Chapter 3

Literature Review

In this chapter, we review several related topics and highlight the advancement of our research to the current state-of-the-art. Particularly, we start with an examination of system load balancing issues in parallel and distributed databases. Then, we make a brief survey on techniques for distributed transaction management. Next, we review current solutions for hybrid OLTP and OLAP systems and look especially at cloud data serving systems and their missing features. Finally, we study related works on transaction and index support in the cloud.

3.1 System Load Balancing

To facilitate parallel processing and provide the important query performance for end-users, a large data table is commonly partitioned and distributed across storage nodes in the system to balance the workload. Three basic partitioning schemes exist, namely hashed, range, and round-robin partitioning [63].

In hashed partitioning, the storage node responsible for maintaining a tuple is determined based on a hash function. Range partitioning, on the contrary, keeps the order of tuples in a relation by assigning contiguous attribute ranges to storage nodes in the system. As the simplest strategy, round-robin partitioning assigns tuples of a relation to storage nodes in a round-robin fashion.

Among the three schemes, round-robin partitioning is rarely used in practice and only suited for applications that typically access all the data in the relation via parallel sequential scanning on each query. On the contrary, hash partitioning is commonly used in applications that desire to find tuples having a particular attribute value, a.k.a. associative access, because tuples are originally placed to a specific partition based on the hashed value on that attribute. Nevertheless, hashing tends to randomize data across partitions due to the hash function, and hence is not suited for applications that require access to clustered data, for example, range queries on a particular attribute. For this type of applications, range partitioning is more preferable since each partition manages tuples with similar attribute values.

Unfortunately, range partitioning is much sensitive to skewed data distribution, where most of the data are assigned to a certain partition, and skewed query distribution in which most queries in the workload tend to access data in a certain partition. We handle the problem of skewed data distribution with a sampling-based data mapping function (cf. Section 5.3.2 in Chapter 5). In particular, a sampling process is performed during data bulk loading phase. The system therefore is able to draw an approximate distribution of the data based on the collected samples. This information enables the system to map the initial skewed data domain back to a uniform data domain.

It is noteworthy that static data placement schemes, e.g., via range partitioning, may be not optimal due to skewed workloads and changes in data access patterns at runtime. In [97], a self-tuning approach is proposed to reorganize the data in shared-nothing systems at runtime in order to correct any degradation in system performance when the access pattern changes dynamically. Other related works include online load balancing in range-partitioned systems using data migration [69]. To achieve storage balance across nodes in the system, the partitions could be adjusted via a data migration process that moves the data between adjacent partitions at runtime based on query workloads.

We note that data migration alone is not sufficient to guarantee the balance of query execution load across storage nodes in the system under skewed workloads. More

specifically, when the system faces a "flash crowd" query, i.e., a sudden increase of query workload targeting to some "hot" data, migrating these "hot" data between partitions only shuffles the bottleneck throughout the system without really balancing the workload across nodes. As a consequence, in this research we propose a smart and load-adaptive replication scheme to provide effective load balancing in shared-nothing systems (cf. Section 6.1 in Chapter 6). While the idea of tuning replication process based on data popularity is common, previous works on replication for load balancing in conventional distributed systems [83, 144, 143] as well as P2P systems [73, 138] maintain the query access statistics on per data object basis. This approach is impractical when the amount of data in the system is large, especially for cloud scale databases, since maintaining such access statistics per record will incur considerable storage and update overhead. Our proposed system employs self-tuning range histogram in order to reduce such cost.

Furthermore, in these previous works, the replication decision to balance work load in the system is based on the local information of each storage node, i.e., an overloaded node will replicate hot data to the nodes on the query routing path. This approach is not necessarily effective as there could be other lightly-loaded nodes in the system that are more suitable to share the workload. Instead, in our proposed system, each storage node can roughly know the load of other nodes since the load information of storage nodes are piggy-backed on the periodical heart-beat messages sent between these nodes. Our experiments show the quick convergence rate of the load statistics information and this information is much useful for the system to determine where to replicate the hot query ranges to effectively reduce the imbalance in the workload.

3.2 Distributed Transaction Management

Distributed and parallel database systems commonly use two-phase locking (2PL) for concurrency control [63, 62, 67, 133]. Possible deadlocks are resolved through standard

techniques that use timeouts and abort one of the deadlocked transactions. Since the cost of locking is expensive in distributed environments with low degree of update operations, there have been research work such as transactional distributed B-tree [32] and [25, 124] that apply optimistic scheme to speed up the concurrency control with the assumption that data conflicts between transactions only happen in the worst case.

However, a simple validation algorithm in optimistic concurrency control might result in unnecessary high number of transaction restarts and there is a high probability that long transactions suffer from starvation. Further, a read-only transaction may also have to abort due to data conflicts with other update transactions committed during its execution time. Therefore, a combination of multiversion and optimistic concurrency control scheme has been shown to be promising in a query-dominant environment [28, 135]. It is important to note that most distributed optimistic concurrency control algorithms, including [28] and [135], are designed to enforce strict serializability and guarantee that there is no inconsistent read or write in the system.

Enforcing strict serializability is costly, especially in distributed and cloud environments, since the systems need to verify read-write conflicts of concurrently executing transactions. Given the fact that snapshot isolation [40] is a widely-accepted correctness criterion and adopted in many open-source and commercial database systems, such as PostgreSQL, MySQL, InterBase, Oracle, and Microsoft SQL Server, we hypothesize that snapshot isolation is also useful for large-scale environments. Consequently, in our research we develop a hybrid scheme of multiversion optimistic concurrency control that provides snapshot isolation for our proposed elastic cloud storage system (cf. Section 6.2 in Chapter 6).

3.3 OLTP and OLAP Systems

Traditionally, online transactional processing (OLTP) and online analytical processing (OLAP) workloads are handled separately by two systems with different architectures –

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. Although this design provides the required flexibility and efficiency, it also results in several limitations, for examples, lack of data freshness for OLAP, redundancy of data storage, as well as high capital and maintenance cost. Not surprisingly, several main-memory resident database systems that handle both OLTP and OLAP have recently been proposed [115, 78, 89].

Given continuous growth of data generated by Web 2.0 and enterprise applications, coupled with advancement in broadband connectivity, virtualization, and other technologies, the cloud computing model, with its capability to dynamically provide for computation and storage, has emerged as an ideal choice for data-intensive and database-as-a-service computing infrastructures. The need to provide for capacity in terms of computation and storage, and to support the combined OLTP and OLAP workload, has given rise to major challenges in architecting elastic and efficient storage systems for supporting database operations in the cloud.

Web 2.0 applications provided by Internet companies such as emailing, online shopping and social networking, are all based on online transactions that are essentially similar to those in traditional OLTP systems. In such web applications, system scalability, service response time and service availability are the foremost requirements. Several proprietary cloud data serving systems for hosting various web applications have been designed and built, including BigTable [49], Pnuts [54], Dynamo [61] and Cassandra [93].

To better support search and data sharing, large-scale ad-hoc analytical processing of data collected from those web services is becoming increasingly valuable to improving the quality and efficiency of existing services, and supporting new functional

features. However, traditional OLAP solutions such as parallel database systems and data warehouses fail to scale dynamically to meet the demand given the massive size of web data. Therefore, both commercial companies and open-source communities have proposed new large-scale data processing systems such as MapReduce [60], Hadoop [5], Hive [137], Pig [8] and Dryad [85].

The divergence between web data hosting and web data analysis is mainly by design. The storage layer and processing layer are loosely coupled so that the processing layer can read data in any format in bulk and perform the necessary processing to produce the indexes or views required by the applications. The frequency at which an analytical or bulk-processing task is invoked is a business decision, and its data freshness is therefore determined based on needs. However, such design causes applications to rely heavily on periodically generated metadata due to its lack of transaction management. Further, due to design by choice, these systems do not support indexing mechanisms that facilitate ad-hoc query processing.

In our research, we architect an elastic storage system that supports the combined OLTP and OLAP workload (cf. Chapter 4). The system employs a hybrid data partitioning scheme that favors both workloads with a careful design of vertical and horizontal partitioning based on the trace of query workload. Furthermore, the system is also designed to provide load-adaptive replication, transactional semantics and index functionality for database applications in the cloud.

3.4 Cloud Data Serving Systems

A thorough survey and feature comparison of cloud data management systems, including scalable SQL and NoSQL data stores [16], is presented in [47]. Here we briefly review some well-known systems and then discuss their missing features at the end of this section. Amazon has built a highly available key-value store called Dynamo [61] for supporting its e-commerce applications which require high reliability and fast

response to customers' operations on the web. Dynamo chooses to guarantee data reliability and availability as a trade-off for relaxed data consistency in the system. Storage nodes in Dynamo are organized into a ring like distributed hash tables (DHT) [130]. Every data object is asynchronously replicated to three nodes. Any replica of an object is "always writable" to users, which may result in divergence of replicas in the system. The inconsistency between replicas of a data object is reconciled at later time, thereby ensuring eventual consistency [140].

Compared to Dynamo, Pnuts cloud data platform of Yahoo! [54] provides per-record timeline consistency and supports more expressive queries. Pnuts uses asynchronous replication to ensure low latency for update operations and only provides per-record timeline consistency. Bigtable [49] and its open-source HBase [6] also provide record oriented access to very large tables which are distributed in commodity clusters consisting of thousands of machines. Bigtable employs column family model to support sparse tables and maps a composite key of three components (including row key, column key, and timestamp) to an associated record value. Cassandra [93] is a hybrid system that employs peer-to-peer network model from Dynamo and column family data model from Bigtable to provide highly available service with no single point of failure for managing large amounts of structured data spread out across commodity servers.

Following this trend, Ambrust et al. [36] propose a cost-effective scalable storage architecture with declarative consistency for social computing applications. However, this perspective paper only provides a general description about the architecture without considerations of the underlying data storage and data model. As implementation plan, they intend to build the system based on Cassandra.

It is noteworthy that most cloud data serving systems do not support transactional (read-modify-write) semantics for operations spanning across multiple data records [47]. Recently, some systems such as MegaStore [38] and ElasTraS [57] have started to support transactions (see below section for further details). In addition, these systems

mainly employ data migration to balance the storage load across the servers in the system. However, under skewed query distributions, it is also critical to balance the query execution load across machines. `ecStore`, our proposed cloud storage system, is designed to support these two features, namely transactional semantics and smart replication, as its intrinsic features, in addition to its novel capability to provide DBMS-like index functionality in the cloud. We present a detailed feature comparison of `ecStore` with other cloud data serving systems in Section 7.4.5 (cf. Chapter 7).

3.5 Transaction Support in the Cloud

Transactional semantics is a desirable functionality when providing scalable database services in the cloud since it allows for bundling read-modify-write operations spanning across multiple data records, which is a common feature required in most database applications. Consistency rationing has been proposed for transaction management in cloud storages [43, 91]. This approach categorizes the application data into three types and provides a different consistency treatment for each category. Consistency rationing at data level instead of at transaction level might incur much overhead of metadata management (for categorizing each data item) when the database size is large.

As alternative approaches, a full support of ACID properties is only guaranteed for data records that reside within a partition of the database (e.g., ElasTraS [57] and SQL Azure [45]), or in a key group (e.g., G-Store [58]), or in an entity group (e.g., Megastore [38]). In these approaches, partitions and entity groups are statically predefined, whereas key groups can be formed dynamically at runtime via a key grouping protocol.

Recently, Lomet and Mokbel [103, 102, 99] put forward that a modern transactional storage can be designed as a system consisting of transactional components and data components, which are not tightly coupled together as in traditional storages. This flexible model could be beneficial for cloud database deployment. When the transaction service is separated from the underlying data service, concurrency and recovery control

become more challenging. To ensure the global correctness of transactions spanning across data components, this approach requires that there is only one transaction component in the system.

Intrigued by inconsistency management techniques recently used in several large-scale distributed storages, e.g., Dynamo [61] and Pnuts [54], and own experiences in developing SAP enterprise applications, the authors of [65] summarize principles for managing inconsistency in data management systems. The mainstream of these principles come from the fact that data inconsistency could be tolerated and compromised for responsiveness and availability of the system.

Besides eventual consistency model, the experience paper [65] also discusses other techniques such as: execute transactions based on local view of data, update a single object within a transaction and use a reliable queue for multiple-object update. Although these principles may not be universal for all applications, it is common that developers choose to compromise strict data consistency in some internet-scale scenarios.

As discussed in Section 3.2, it is costly to enforce strict serializability for transactions in large-scale environments. However, the wide-spread use of snapshot isolation [40] in many open-source and commercial database systems, for examples, InterBase, Oracle, Microsoft SQL Server, PostgreSQL and MySQL, leads us to hypothesize that snapshot isolation is also useful for large-scale environments such as cloud. Therefore, we design our elastic storage system to provide snapshot isolation (cf. Section 6.2 in Chapter 6).

Snapshot isolation is first formalized by Berenson et al. [40], which shows that any multiversion concurrency control scheme that ensures "first-committer-wins" rule will provide the standard snapshot isolation level. Instead of proposing a new isolation level, which may be not useful and accepted by users, our proposed storage system (`ecStore`) extends that technique in dynamic distributed (cloud) environments in several ways. First, in `ecStore`, distributed write locks during the write phase are managed by a separate service called Zookeeper, which is widely used in distributed environments for providing efficient distributed synchronization. Second, since it is complicated to detect

and resolve deadlocks at runtime, and the problem is even more challenging in large-scale distributed environments such as cloud, `ecStore` chooses to avoid deadlocks by enforcing each transaction to always request the locks in the same sequence, e.g., based on the order of records' key. Third, `ecStore` does not solve the problem of transaction consistency alone, but also considers the interplay between transaction consistency and replication consistency.

3.6 Index Support in the Cloud

To support data-centric applications, the cloud must provide an efficient and elastic database service with similar functionalities as centralized databases. The provision of indexes is an important feature among these functionalities. One obvious requirement for this functionality is the ability to locate some specific records among millions of distributed candidates in real-time. A second requirement is to support multiple indexes over the data – a common service in any DBMS – including primary and secondary indexes. Another important requirement is extensibility by which users can define new indexes without knowing the structure of the underlying network or having to tune the system performance by themselves.

Currently no cloud data serving system satisfies these requirements. Most popular cloud storage systems are key-value based, which, given a key, can efficiently locate the value associated to the key. Examples of these systems include Dynamo [61] and Cassandra [93]. These systems build a hash index over the underlying storage layer, partitioning the data by keys (e.g., primary index). For supporting primary range index, a scalable distributed B-tree has been proposed [32] for cluster environments. While these proposals are efficient in retrieving data based on primary index, they are not useful when a query does not use the key as the search condition. In these cases, sequential (or even parallel) scanning of the entire (large) table is required to retrieve only a few records, and this is obviously inefficient.

Recently, Cassandra [93] has started to support native distributed indexes on non-key attributes. However, the secondary indexes in Cassandra are restricted to hash indexes. Although Megastore [37] provides global indexes spanning across multiple data entity groups, the support of distributed multi-dimensional indexes has not been reported in this literature. A second line of research has been view materialization in the cloud to support ad hoc queries. In [29], a view selection strategy is proposed to achieve a balance between query performance and view maintenance cost.

Secondary indexes can be implemented as a specific type of materialized views. However, this approach might not be scalable when providing indexes in the cloud since the system needs to build a separate materialized view for every specific query in the workload. Instead, in our system, the indexes could be optionally declared as covering indexes, i.e., the index entries of these indexes contain a portion of the base records, and therefore a single index could facilitate the processing of multiple queries that access different columns contained in the index entries.

Two secondary indexes have been proposed recently for cloud systems including a distributed B^+-tree-like index to support single-dimensional range queries [145], and a distributed R-tree-like index to support multi-dimensional range and kNN (k Nearest Neighbor) queries [141]. The main idea of both indexes is to use P2P routing overlays as global indexes and combine with local disk-resident indexes at each index node. The overlays are used for distributing the system workload, by partitioning data across storage nodes and routing queries to appropriate nodes.

On the contrary, IR based strategies in integrating distributed independent databases over unstructured network, without any global index, are proposed in [109]. To the best of our knowledge, none of the existing works, including [145, 141, 109], has addressed the scalability and performance issues of supporting a large number of indexes of different types (e.g., hash, range, and multi-dimensional indexes) in the cloud. Consequently, we propose a generalized distributed indexing framework to provide DBMS-like index functionality for cloud environments in Chapter 5.

3.7 Summary

In this chapter, we have reviewed several research areas related to our work and highlighted specific advancement of our research. In particular, we propose to handle the problem of skewed data distribution in partitioned storage systems with a sampling-based data mapping function. We also propose a smart replication scheme which is adaptive with the database workload to deal with skewed query distributions. Furthermore, since strict enforcement of serializability for transactions in large-scale environments is costly, our proposed cloud storage system provides snapshot isolation, which is a widely-accepted correctness criterion, and uses the key ordering to sequence writes in transactional access in order to avoid deadlocks in distributed environments. The system employs a hybrid data partitioning scheme with a careful design of vertical and horizontal partitioning to facilitate both real-time and analytic workloads. In addition, a comprehensive and efficient distributed indexing framework, which is limited in other cloud data serving systems, is proposed by our research. In the next chapter, we shall present detailed design and implementation of our proposed elastic cloud storage system.

Chapter 4

A Hybrid Cloud Storage for Supporting Both OLTP and OLAP

As cloud computing has gained fast popularity, it is timely and desirable to have an integrated system with both high-performance OLTP and OLAP capabilities. In this chapter, we present a new system architecture for supporting database operations in the cloud spanning clusters of commodity machines. `ecStore` – our proposed **e**lastic **c**loud **stor**ag**e** system – supports both OLTP and OLAP workloads which run simultaneously and interactively within the same storage. `ecStore` is also designed to provide other essential capabilities for database applications in the cloud such as load-adaptive replication, comprehensive distributed indexing framework and transactional semantics for read-modify-write operations across multiple records, which are important features but limited in most cloud data serving systems.

The remainder of this chapter is organized as follows. In Section 4.1, we introduce `ecStore` as the storage manager of a bigger cloud data management system. In Section 4.2, we present the data model used in `ecStore`. We give the overall architecture of `ecStore` in Section 4.3. Detailed design and implementation of `ecStore` is presented in Section 4.4. We conclude the chapter in Section 4.5.

4.1 Elastic Storage in the epiC

ecStore is part of a bigger system named epiC – an elastic power-aware data-intensive Cloud computing platform – for providing scalable database services in the cloud. In epiC, two typical workloads including data intensive analytical jobs (OLAP) and online transactions (OLTP) are supported to simultaneously and interactively run within the same storage and processing system. The overall system architecture of the epiC cloud data management system, as illustrated in Figure 4-1, consists of the following main modules: Query Interface, OLAP/OLTP Controller, the Elastic Execution Engine (E^3) and the Elastic Storage System (ecStore). Here, we briefly introduce these modules and how they work together in a cohesive system. More details of the epiC system are described elsewhere [51].

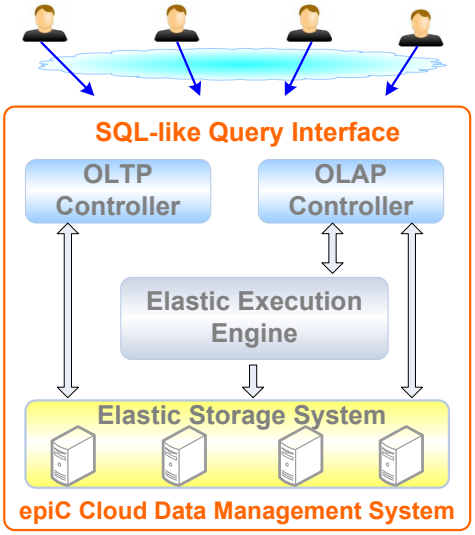


Figure 4-1: The epiC cloud ecosystem.

The Query Interface of epiC provides a SQL-like language for client applications and compiles the SQL query into a set of analytical jobs (for OLAP query) or a series of read and write operations (for OLTP query), which will be handled by the OLAP/OLTP Controller respectively. E^3 [52] is a sub-system of epiC that is designed to efficiently perform large-scale analytical jobs in the cloud. ecStore, the underlying cloud data storage system for supporting both OLAP and OLTP workloads, provides data access interfaces for upper-layer query processing engines, i.e., the OLAP/OLTP Controller.

ecStore embeds *load-adaptive replication* mechanism in order to provide important features such as data availability and system load balancing. ecStore also develops a *comprehensive indexing framework* for declaration of various types of distributed indexes, e.g., hash, B^+-tree-like and R-tree-like indexes [130, 145, 141], over the cloud data in order to facilitate efficient processing of ad-hoc queries. The essence of these indexes is to use P2P structured overlays such as Chord [130], BATON [86] and CAN [122] as global indexes and combine with local disk-resident indexes at each index node. This strategy is more efficient than IR-based strategies in integrating distributed independent databases over unstructured network proposed in [109].

The two query processing engines, namely OLAP controller and OLTP controller, are implemented to handle different types of queries and monitor the processing status. After a query is submitted to epiC, the Query Interface first checks whether the query is an OLAP query, i.e., an analytical query performing aggregation and join across multiple tables, or an OLTP query, i.e., a simple select or update query on a single table. In the former case, the query is forwarded to the OLAP controller which transforms the query into a set of E^3 jobs. For each job, the OLAP controller defines the input and output, both of which are tables in ecStore.

A specific processing order of E^3 jobs constitutes a query plan. The OLAP controller employs a cost-based optimizer to generate a low cost plan. Specifically, histograms are built and maintained in the underlying ecStore by running a built-in E^3 job periodically. The OLAP controller queries the metadata catalog of the ecStore to retrieve the histograms, which can be used to estimate the cost of a specific E^3 job. It then iteratively permutes the processing order of the jobs and estimates the cost of each permutation. The one with lowest cost is then selected as the query plan. Based on the selected plan, the OLAP controller submits jobs to E^3. After E^3 has completed the jobs, the controller collects the result and returns it to the user.

If the query is a simple select query on a single table, the OLTP controller will take over the query. It first checks with the metadata catalog of ecStore to get histogram

and index information. Based on the histograms, it can estimate the number of resulted records. Then, the OLTP controller chooses a proper access method such as either index lookup or parallel sequential scan depending on which method incurs the lower cost in terms of network and disk I/Os. Finally, the OLTP controller invokes the functions provided by the data access interface of `ecStore` to perform the operations. Similarly, data manipulation queries (insert/upddate/delete) on a table are parsed by the OLTP controller of `epiC` and passed to `ecStore` for further execution via the data access interface provided by `ecStore`. Both OLTP and OLAP controller rely on the underlying `ecStore` storage system to provide *transactional support* for bundling read-modify-write operations across multiple records and for isolating OLTP and OLAP queries.

Since OLAP is often more computationally expensive than OLTP, `epiC` utilizes the data replication provided by `ecStore` to guarantee the requirement of *resource isolation* during the processing of OLTP and OLAP as follows. `epiC` divides the storage nodes in the system into two replica groups and only launches OLAP jobs on the nodes of one group while reserving the nodes in the other group for serving OLTP requests. Consequently, the execution of OLAP will not compete for resources with the response-critical OLTP operations. Note that the writes done by OLTP operations will be propagated into the other replica group in real-time so that OLAP can access the up-to-date data. Utilizing data replication for implementing resource isolation is also used in [21], which aims to unify Hadoop MapReduce [14] and Cassandra [93] for supporting both real-time and analytic workloads.

4.2 Data Model

NoSQL storage systems [47] represent a recent evolution in building infrastructures for serving large-scale data. Most of these cloud data serving systems such as Bigtable [49], HBase [6], Dynamo [61] and Cassandra [93], employ key-value model or its

variants (e.g., column family model for supporting sparse tables) and choose to provide scalability for the system as a trade-off for the lack of full functionality of a DBMS. More recently, some systems such as Megastore [38] adopt a variant of the abstracted tuple model of an RDBMS where the data model is declared in a schema and with strongly typed attributes. Pnuts [54] is another large-scale distributed storage system that uses the tuple oriented data model.

It is noteworthy that systems that focus on ad-hoc analysis of massive datasets, i.e., OLAP queries, such as Hive [137], Pig [111] and SCOPE [48], are sticking to the relational data model or its variants. Since `ecStore` is designed to provide effective and efficient supports for both OLTP and OLAP queries and multitenancy in the future, its data model is also based on the widely accepted relational data model where data are stored as tuples in relations, i.e., tables, and a tuple comprises of multiple attributes' values, i.e., columns.

However, `ecStore` further adapts this model to support column oriented storage model in order to exploit the data locality property of queries that frequently access a subset of attributes in the table schema. This adaptation is accomplished by a flexible partitioning strategy. In particular, `ecStore` is essentially designed to operate on a large cluster of shared-nothing commodity machines and therefore it employs both vertical and horizontal data partitioning schemes to facilitate parallelism and provide high performance in terms of throughput and latency.

In this hybrid partitioning scheme, columns in a table schema that are frequently accessed together in the query workload are grouped into a *column group* and stored in a separate physical table. This vertical partitioning strategy facilitates the processing of OLAP queries which often access only a subset of columns within a logical table schema. Further, for each physical table corresponding to a column group, a horizontal partitioning scheme is carefully designed based on the database workload so that cross-partition transactions only happen in the worst case. We shall present details of the data partitioning scheme in `ecStore` in Section 4.4.2.

4.3 Overall Architecture

Figure 4-2 illustrates the overall architecture of `ecStore`, our proposed elastic storage system. The **data access interface** is exposed to developer users, processing engines (e.g., the OLTP and OLAP controller of `epiC`) and other applications/tools for submitting data access requests, while the **data access manager** is in charge of handling basic operations on the data stored in the underlying storage engine, based on the interpreted commands passed by the data access interface.

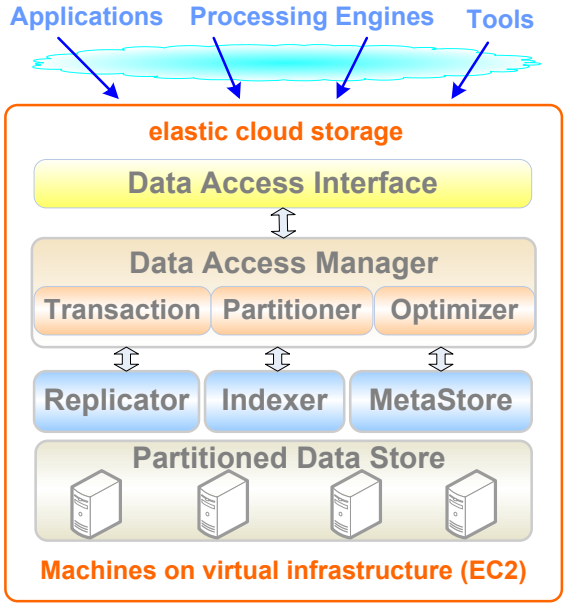


Figure 4-2: Architecture of `ecStore`.

The data access manager is deployed on each storage node in the system to share the workload of incoming data access requests. The data access manager is comprised of three major sub-components: the **transaction manager** for handling OLTP and OLAP isolation, the **partitioner** for dealing with data partitioning issue and the **data access optimizer** for composing a near optimal access plan given a data request.

At the lowest level, the **indexer** maintains a number of distributed indexes for efficient primary and secondary access to the underlying **partitioned data store**. The **replicator** is responsible for realizing load-adaptive replication for both application data and index data in `ecStore`. The general flow to process a data access request is as follows. After receiving requests from clients, the data access interface parses these

requests into the corresponding internal representations that the data access manager operates on and chooses a near optimal data access plan (based on the statistics information stored in the **metastore**) such as parallel sequential scan or index scan or hybrid for locating and operating on the target data stored in the partitioned data store.

4.4 Design and Implementation

In this section, we subsequently present detailed design and implementation of various components of `ecStore`, including data access interface, data partitioner, partitioned storage engine, indexer, metastore, data access optimizer, replicator, and transaction manager.

4.4.1 Data Access Interface

`ecStore` provides two independent data access interfaces, namely OLTP interface and OLAP interface, for OLTP queries and OLAP queries respectively. This is beneficial as these two types of query have diverse data access patterns and thus present different requirements on the data access interface.

OLTP Interface

For the OLTP workload, data is typically accessed via *point* or *small-range* queries. With this type of query, only one or several records within a single table will be located and manipulated. We define three major APIs for this OLTP workload:

- `get(table, key, columns)`
- `put(table, record)`
- `delete(table, key)`

In the above interface, the parameter `table` represents a logical table, whose internal storage consists of one or multiple physical tables through vertical partitioning

(cf. Section 4.2). The parameter `key` is a set of conjunctive or disjunctive selection predicates on some columns, and is used for locating the target records, which are then either retrieved by the `get` operation or eliminated by the `delete` operation. The parameter `columns` is the set of columns to be projected out of the target records. The parameter `record` is a new record to be inserted into the table by the `put` operation. The operation of updating a record is realized as appending a new version of the record to the system, as will be discussed in Section 4.4.8.

OLAP Interface

For the OLAP workload, data is usually accessed via batch processing, which means that not only a large number of records are read, but multiple tables are also accessed within a single query.

The OLAP interface basically provides iterator mode [75], with three major APIs including `open()`, `next()` and `close()`. At the beginning, the `open` function is called to initialize the scan of one logical table, parameterized by record selection and projection predicates. These parameters will be used by the data access optimizer to determine data access plans (e.g., scanning a specific data partition or performing an index scan). The `open` function returns when the data access manager has done preparation jobs for the scan. After that, the `next` function could be repetitively invoked to retrieve a set of records each time. After all qualified records have been retrieved, the `close` function performs some housekeeping tasks such as closing up the physical tables and updating the relevant data access statistics in the metadata store.

It is possible that the execution engine, e.g., the $\mathtt{E}^3$ as discussed in Section 4.1, also deals with other data formats, e.g., key-value pairs, rather than record representation. Therefore, inside the `next` function we implement a data format translator which optionally converts the retrieved records into the desired formats before returning them to the execution engine. It is also important to note that compared to MapReduce-based systems, in which all participating tables are sequentially scanned in parallel and

unqualified records are filtered out by Map tasks, `ecStore` provides an additional index scan functionality, which is especially useful when the set of qualified records is merely a small portion of the entire records that would be touched by the sequential scan.

Therefore, `ecStore` allows the execution engine to push the record selection and projection predicates, along with the table names, through the OLAP interface and into the data access manager which will choose a near optimal data access plan and execute it. As a result, only a small number of records need to be scanned if an appropriate index exists and only the required columns of qualified records will be returned to the execution engine. At the level of the data access manager, we also consider further supporting other relational operations such as aggregation and sorting that could enable additional optimizations like early aggregation [95] and shared scan [110].

4.4.2 Data Partitioning Strategy

The data partitioner module of `ecStore` employs both vertical and horizontal data partitioning schemes, as depicted in Figure 4-3.

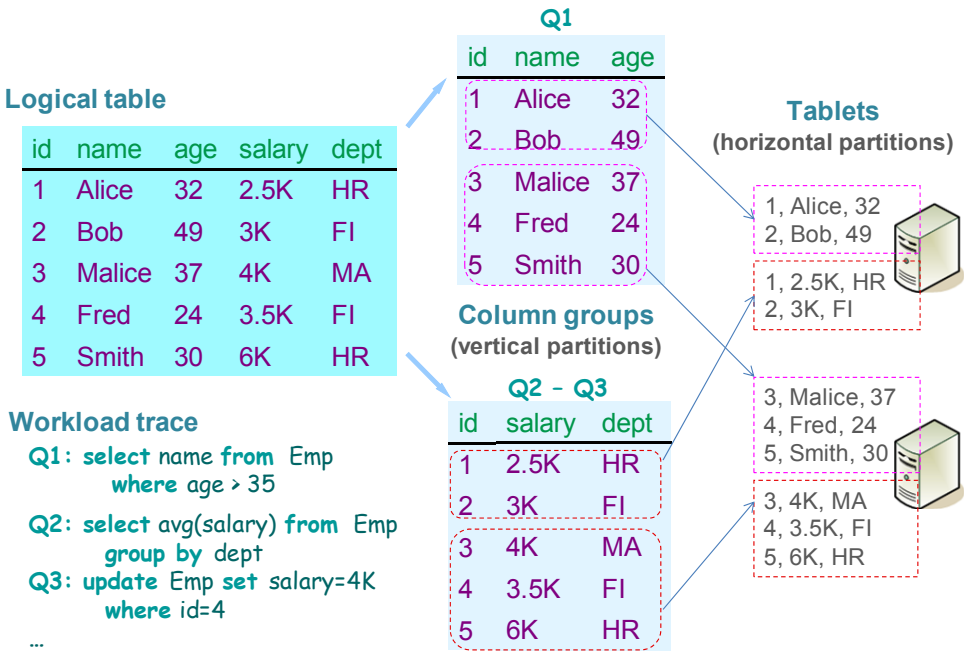


Figure 4-3: Hybrid data partitioning scheme in `ecStore`.

Vertical partitioning. `ecStore` optionally divides columns of a table schema into several *column groups* based on the query workload. Each column group comprises of columns that are frequently accessed together by a set of queries in the workload. Column groups are stored separately in different physical data segments so that the system can exploit data locality when processing queries.

Such vertical partitioning is especially useful for OLAP queries which often require access to only a subset of columns within a table since it saves significant I/O cost compared to the approach that stores all columns of the table schema into a single physical table. Additionally, transactional accesses to data records often update the values of some columns in a column group. Hence, the vertical partitioning technique improves the overall performance of the system significantly by reducing the I/O cost in most cases.

Consider the example in Figure 4-3. The original table has an abstracted schema with five columns $(id, name, age, salary, dept)$. The sample workload trace includes three queries $\{Q_1, Q_2, Q_3\}$ as listed in the figure. It is beneficial to partition the schema into two column groups, namely $CG1(id, name, age)$ and $CG2(id, salary, dept)$, and store them in separate physical tables in order to minimize I/O cost of the given workload since Q_1 mainly accesses data in CG_1 while Q_1 and Q_3 mainly access data in CG_2.

The above partitioning strategy is similar to data morphing technique [80] and vertical partitioning in Hyrise [78] – a main memory database system developed by SAP, which also partition the table schema into column groups. The main difference is that these two approaches aim at designing a CPU cache-efficient column layout while the vertical partitioning strategy in `ecStore` focuses on exploiting data locality for minimizing I/O cost of a query workload.

To obtain such vertical partitioning, `ecStore` adopts an existing technique proposed for automated physical database design [31]. In particular, given a table schema with a set of columns, multiple ways of grouping these columns into different vertical partitions are first enumerated. Then, I/O cost of each configuration is computed based on the

query workload trace and the best way of grouping columns with the lowest I/O cost is selected as vertical partitions for the table schema.

The I/O cost of a column group is measured through its effectiveness in reducing the size of data scanned by queries in the workload. This effectiveness is computed as the fraction of scanned data (aggregate of columns' size multiplied by the number of its occurrences in the workload trace) that are actually needed to answer queries when any column in the column group is referenced in the queries. Note that if the table schema includes many columns, the number of possible ways of grouping columns may result in a combinatorial explosion, and it is therefore not efficient to enumerate all the groupings. In this case, a local search algorithm could be employed to obtain an approximate best candidate for partitions as in the data morphing technique [80].

It is also noteworthy that since we have designed the vertical partitioning scheme based on the trace of query workload, tuple re-construction is only necessary in the worst case. Moreover, each column group still embeds the primary key of data records as one of its componential columns, and hence to reconstruct a tuple, `ecStore` collects the data in all column groups using the primary key as selection predicate.

Horizontal partitioning. To provide scale-out capability, `ecStore` further splits the data in each column group into horizontal partitions when the system actually stores the physical data segment corresponding to this column group.

While there have been works on automating physical database design for parallel database systems such as [108, 120], they focus on data warehousing environments and aim to design horizontal partitioning and replication scheme for tables in order to efficiently support complex long-running queries. In contrast, the main aim of horizontal partitioning in `ecStore` is to distribute data access load across storage nodes while reducing the number of distributed transactions.

A carefully design of horizontal partitioning scheme can help to reduce or even eliminate distributed transactions across machines, and thus simplify the transaction management in the system. Typically, users tend to operate on their own data which

form an entity group as characterized in MegaStore [38]. By cleverly designing the key of data records so that all data related to a user have the same key prefix which is the user's identity. Hence, data accessed by a transaction are usually clustered on a physical machine. In this case, execution of transactions is not costly since a full two-phase commit (2PC) protocol is not needed.

For scenarios where the application data cannot be naturally partitioned into entity groups, we can implement a group formation protocol that enables users to explicitly cluster data records into key groups [58]. An alternative solution is workload-driven approach for data partitioning [56]. This approach models the transaction workload as a graph in which data records are represented as vertices and transactions are represented as edges and uses a graph partitioning algorithm to split the graph into sub-partitions that minimize the number of cross-partition transactions.

4.4.3 Partitioned Storage Engine

We now present the implementation of the underlying partitioned data store and the local persistence at each storage node.

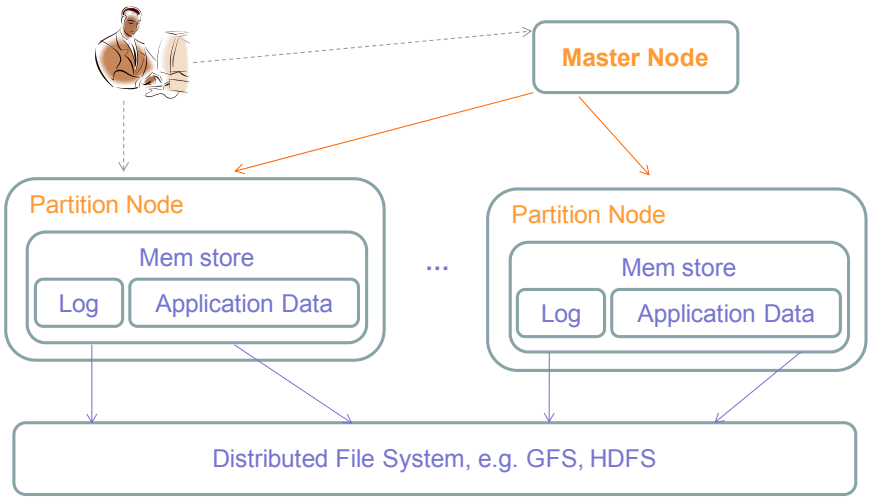


Figure 4-4: Shared-storage architecture with distributed file system.

Shared-nothing vs. Shared-storage Architecture

Typically, there are two common designs for architecting distributed storage engines, namely shared-storage and shared-nothing system [131]. In the former design, as illustrated in Figure 4-4, the system is built on top of a shared distributed file system (DFS). Partition nodes, a.k.a. tablet servers in HBase [6], are responsible for controlling users' data requests. These partition nodes maintain data on memtables temporarily and persist them into the shared DFS when the memtables are full. Relying on a shared DFS simplifies the design, but it also entails some disadvantages. For example, this architecture increases users' perceived latency due to the separation of control layer and shared storage since forcing a log page or data page into the shared DFS incurs fair a bit of overhead. In addition, the downtime of the system also increases when a partition node crashes since there is no notion of hot standby or replication on the control layer. All data serviced by a failed partition node are unavailable until that node is restarted and its log in the shared DFS is fully replayed.

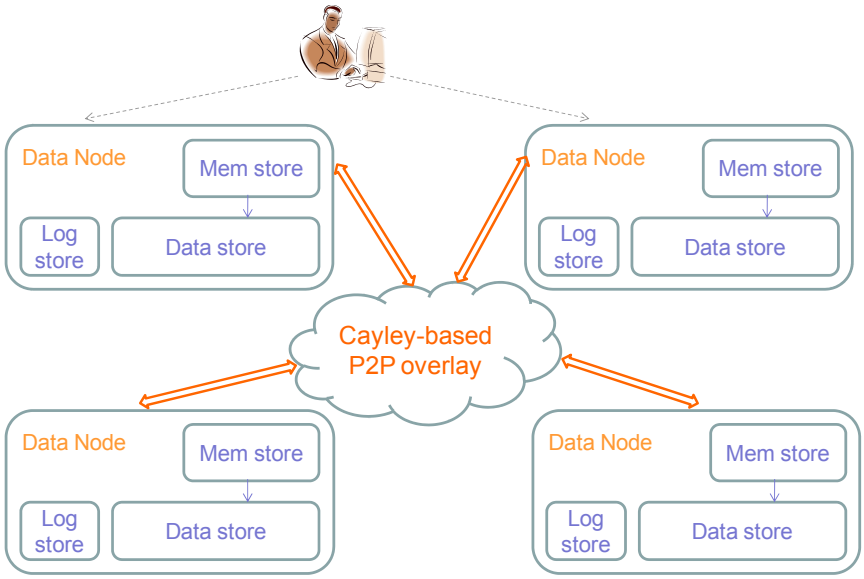


Figure 4-5: Shared-nothing architecture with generalized partitioned data store.

We therefore adopt the latter design, i.e., shared-nothing architecture, as depicted in Figure 4-5. In this design, each data node in the system maintains a partition of the

entire data and persist these data on its local storage. With this design choice, we can devise our own replication and load balancing strategy instead of sticking to a specific shared DFS, and thus being able to reduce the system downtime for the support of replication failover. Furthermore, the additional round trip network latency when interacting with the shared DFS is totally avoided. Also note that in this shared-nothing architecture the data nodes are organized into structured P2P overlays such as CAN [122], Chord [130] and BATON [86], and therefore eliminate the complexity and single-point-of-failure of the master node in the shared DFS. We adopt the Cayley graph model [34, 121] to unify the implementation and deployment of these structured overlays and reduce the maintenance cost.

Generalized Partitioned Elastic Data Store

Following the principle of pay-per-use model or the notion of computing services being organized as a utility, a cloud storage system should be able to provide dynamic scalability and allow users to scale-out and scale-back on the fly based on the load characteristics. This desideratum can only be achieved when storage nodes could be easily added into or removed from the system without having to manually re-partition, replicate and re-distribute the data. Therefore, we do not use the client-server approach as adopted in [43] that builds database systems on top of an existing cloud storage, for example, Amazon S3 [4]. In this approach, data pages are retrieved from S3 for buffering and updating locally at the clients, and modifications to these data pages are finally written back to S3 at transaction commit time, which affects the overall system performance and concurrency control. Instead, we propose to construct a scalable storage system which runs within the cloud cluster to achieve higher performance.

To facilitate parallelism, as discussed in Section 3.1, partitioning strategies such as hash partitioning and range partitioning are commonly used in distributed and parallel databases. Each strategy has its own advantages, for example, hash partitioning provides good load balancing and efficient exact-match queries while range partitioning

favors queries which access clustered data. It is therefore desirable that the system includes these partitioning strategies as intrinsic features. P2P overlays provide good structures for supporting distributed searches on partitioned data, e.g., Chord [130] for distributed hash structures, BATON [86] for distributed range structures, and CAN [122] for distributed multi-dimensional structures. These structures have been shown to be robust in terms of handling node joining and leaving while offering efficient query processing.

However, we cannot afford to implement and maintain multiple overlays in the cluster. Consequently, we have developed a generalized indexing framework, which provides an abstract template overlay based on the Cayley graph model [34, 121]. Based on this framework, the structure and behaviors of different overlays can be customized and mapped onto the template, thereby overloading the overlay with multiple distributed search structures. The partitioned storage engine of `ecStore` employs these structures to realize multiple types (e.g., hash, range, and multi-dimensional) of primary indexes and secondary indexes. Data tables in `ecStore` are stored as clustered indexes on their primary keys while secondary indexes are commonly non-clustered and designed for supporting queries on non-key attributes. Further description of indexes in `ecStore` is presented in Section 4.4.4.

It is noteworthy that in peer-based cloud data serving systems such as Dynamo [61] and Cassandra [93], the query processing only takes $O(1)$ search hop latency. Particularly, the storage nodes in these systems, which are organized as a chord ring [130], use a gossip-based protocol to exchange the membership information. For ecStore (and its Cayley graph-based overlay) to be competitive, we propose to cache routing information at each storage node in order for speeding up the performance of query processing.

In particular, the storage nodes in `ecStore` exchange and update these routing information by piggy-backing the information on heart-beat messages, which is periodically sent between storage nodes. Hence, after a short period of time each

storage node can the cache routing information to all other nodes in the system. Maintaining the routing cache is not expensive since storage nodes do not join or leave frequently in cluster environments.

Local Persistence

Since `ecStore` adopts shared-nothing architecture as we have discussed above, each data node persists its data in a local storage. Different workload such as whole-row access or subsets of columns access maps directly to storage layout, i.e., how we lay out rows and columns on disk, which eventually affects the performance of different disk access patterns such as write-dominant, read-dominant and fast scans. We therefore design the local persistence of each data node as an add-on component, which can be pluggable with various options depending on the application workload. Specifically, `ecStore` provides a generic interface for connecting with various add-on local storage engines such as Berkeley DB [10], log-structured merge trees (LSM) [112], PAX file [33], and columnar files [81, 101]. For evaluation purpose, we employ Berkeley DB Java Edition [10] in the current implementation of `ecStore`.

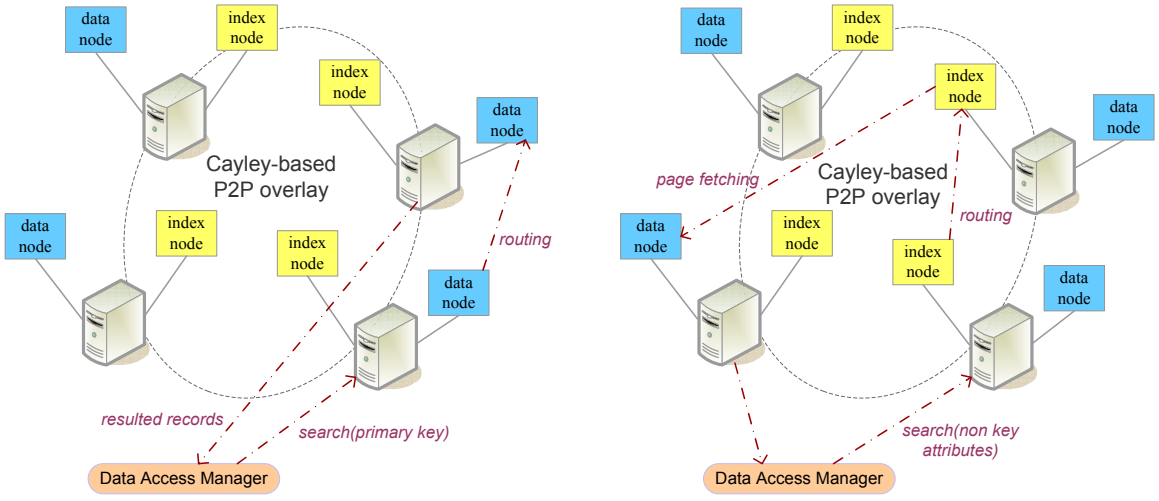
4.4.4 Generalized Distributed Indexes

For OLTP queries and OLAP queries with high selectivity, it is not efficient to perform sequential or parallel scan on the entire table just to retrieve a few records. However, scanning is inevitable if query predicates do not contain attributes that determine the horizontal data partitioning scheme in the system.

To deal with this problem, `ecStore` maintains various types of distributed indexes to facilitate different kinds of queries. For examples, distributed hash indexes for supporting single-dimensional exact-match queries, distributed B^+-tree-like indexes for supporting single-dimensional range queries, and distributed R-tree-like indexes for supporting multi-dimensional range and kNN queries. Here, we briefly present the indexing framework while its detailed design and implementation are described in

Chapter 5. In particular, the indexer component in `ecStore` maintains a set of primary and secondary indexes that are distributed over the underlying partitioned data store. It works as a middleware between the data access manager and the partitioned data store, i.e., it interacts with the partitioned data store and provides data retrieval interface for the data access manager.

As discussed in Section 4.4.3, `ecStore` employs Cayley graph model [34, 121] to realize a generalized partitioned data store that can support various types (e.g., hash, range, and multi-dimensional) of primary and secondary distributed indexes. Data tables in `ecStore` are stored as clustered indexes on their primary keys while secondary indexes are commonly non-clustered and designed for supporting queries on non-key attributes. We note that primary and secondary indexes share the same set of machines in the cluster. However, for clarity, we refer to the machines maintaining primary indexes and secondary indexes as data nodes and index nodes, respectively.



(a) Search with primary index

(b) Search with secondary index

Figure 4-6: Index search with primary and secondary indexes in `ecStore`.

The index search with primary and secondary indexes is depicted in Figure 4-6. With the support of primary indexes, the processing of a query that has predicates on primary attributes is straightforward. In particular, the data access manager residing on a data node will forward the query to the appropriate data node that maintains the data

of interest and retrieve the resulted records. The forwarding process follows the routing algorithm of the underlying overlay abstracted by the Cayley graph.

For queries that have predicates on non-key attributes, `ecStore` exploits corresponding secondary indexes to facilitate the search. Particularly, the data access manager forwards the query to appropriate index nodes to retrieve the index entries for the resulted records. Each index entry contains the index key and pointers which can be used to retrieve the corresponding records from the primary index in the data node. If there is no secondary index that can answer the queries' predicates, then a parallel sequential scan on the primary index is unavoidable.

It is notable that the indexes in `ecStore` consist of primary and secondary indexes, and thus index pages are typically stored independently from data pages, which could even be located on different machines. Therefore, `ecStore` also provides another option for collocation of data and index by the use of covering indexes, which embed sufficient data from the base records into the index entries so that the system can answer the queries directly without having to access the based data. This approach improves the performance of query processing significantly for its reduction in network I/Os and disk I/Os. The additional storage cost for maintaining covering indexes is acceptable when the sizes of base records are relatively small.

4.4.5 Metadata Catalog

The metadata catalog, i.e., the metastore of `ecStore`, provides services for maintaining schema information, statistics information such as histograms, and runtime statistics collected via daemon processes. In particular, the information stored in the catalog include (1) table ownerships and definitions such as column names, data types and primary/foreign key(s), (2) partitioning information such as collocated tables, partitioning keys, clustering keys (i.e., sort orders), and (3) table cardinalities, single or multiple dimensional histograms built on columns, available secondary indexes and table access statistics.

Since data access plans are composed based on the information in the metastore, it is important that `ecStore` maintains a consistent view for the metadata. A naive solution is to put the entire catalog in a single node and adopt a simple locking mechanism with fine granularity to enable the data synchronization. The locking mechanism includes sharable read locks and an exclusive write lock. However, in order to improve the scalability and availability of the catalog, in reality we choose to deploy the catalog on a set of distributed nodes and add data replication mechanism with an appropriate data synchronization technique. In particular, we apply different consistent model for different types of metadata, e.g., strict consistency for schema information and relaxed consistency for runtime statistics.

4.4.6 Data Access Optimizer

Essentially, the record retrieval commands passed from the OLTP and OLAP interfaces of `ecStore` are processed by the data access manager. There are two typical data access methods for record retrieval, namely *parallel sequential scan* and *index scan* (random access is a special case of index scan on primary or secondary indexes in which only one record is retrieved).

Note that OLTP and OLAP queries only work with the logical tables via the data access interface and are transparent with the physical organization of these tables. Recall that the columns of a logical table are organized as column groups, each of which is stored in a separate physical table. Therefore, the data access manager may need to assemble projected records for a logical table with corresponding records from componential physical tables. In the case of sequential scan, the data access manager is able to read records belonging to different horizontal partitions of a physical table in a parallel manner (hence parallel sequential scan). Under this situation, an OLAP query will invoke multiple `next` functions simultaneously, one for each partition.

For OLTP queries and OLAP queries with high record selectivity, various distributed secondary indexes maintained by the indexer component of `ecStore` enable index scan

as an alternative option to parallel sequential scan. However, index scan is not always a better choice than parallel sequential scan. First of all, the appropriate indexes may not be available. In addition, the underlying partitioned data store may have a high latency for random access. As a result, even if the index traversal is sufficiently fast, the overall response time that includes the time for retrieving records from the underlying partitioned data store may be large. This means that, in some cases parallel sequential scan would still be preferred. Therefore, the data access manager of `ecStore` should not assume that index scan is always more suitable for OLTP and OLAP queries with high record selectivity.

To address the above problem, we develop a *data access optimizer* within the data access manager component of `ecStore`, which dynamically chooses the best data access scheme for a specific data access request, relying on the statistics stored in the metadata catalog. The workflow of the cost-based optimization algorithm in `ecStore` is illustrated in Figure 4-7.

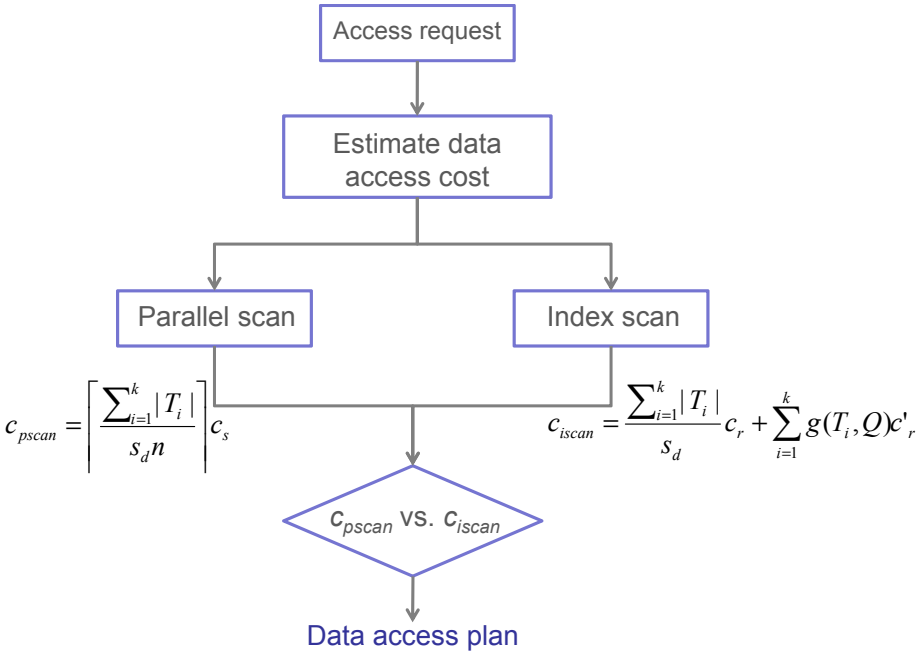


Figure 4-7: Data access optimization algorithm.

The core issue of data access optimization is finding the most efficient access method to read a specified set of records from individual physical tables involved in the query.

The naive solution would be based on a threshold of the query selectivity. That is, for each physical table, if the record selectivity is below some pre-defined threshold, we shall choose index scan; otherwise, we shall choose parallel sequential scan. Clearly, this solution lacks of flexibility and adaptability. Even if we assume that the cluster nodes are static (i.e., with fixed hardware configurations), the possibility of having a sub-optimal access method would be high, not to mention the dynamism of cloud cluster environments as well as the diversity of queries submitted from a large number of users. Therefore, we instead adopt a cost-based approach for better composing near optimal data access plans.

Table 4.1: Parameters for data access optimization algorithm

Parameter	**Definition**
c_s	cost ratio of sequential read
c_r	cost ratio of random read
c'_r	cost ratio of random read with sequential offsets
s_d	size of a data chunk
$f(Q)$	number of I/Os for query Q
$g(T_i, Q)$	number of T_i's tuples that satisfy the selection predicates of Q
n	total number of nodes in the cluster

For simplifying the presentation, we list the parameters used in our cost model in Table 4.1. The data are partitioned into equal-size (s_d) data chunks in the underlying data store. Given a data access request Q, we define function $f(Q)$ to denote the size of data involved in the processing. For parallel sequential scan, if table T_1,...,T_k are involved in Q, $f(Q)$ is computed as $\sum_{i=1}^{k} |T_i|$. In index scan, $f(Q)$ is estimated as $\sum_{i=1}^{k} g(T_i, Q)$, where $g(T_i, Q)$ denotes the number of tuples in T_i that satisfy the selection predicates of Q based on our histograms. Particularly, the costs of different access methods are estimated as follows.

1. The cost of using parallel sequential scan to process Q can be computed as:

$$c_{pscan} = \lceil \frac{\sum_{i=1}^{k} |T_i|}{s_d n} \rceil c_s \tag{4.1}$$

Equation 4.1 is based on the assumption that data are uniformly distributed across the cluster and each node only needs to scan its local data chunks. Since `ecStore` handles the problem of skewed data distribution with sampling-based data mapping functions (cf. Section 5.3.2 in Chapter 5), this assumption is sufficiently satisfied. Consequently, Equation 4.1 provides a good estimate for evaluating the cost of parallel sequential scan.

2. In index scan, we group the requests to the same data chunk and perform the random access in sequential offsets because the cost of random access via sequential offsets (c'_r) is far less than the cost of random access via random offsets (c_r). Suppose the retrieved tuples are uniformly distributed over the data chunks, and therefore the number of data chunks is:

$$\frac{\sum_{i=1}^{k} |T_i|}{s_d}$$

The cost of index scan is estimated as:

$$c_{iscan} = \frac{\sum_{i=1}^{k} |T_i|}{s_d} c_r + \sum_{i=1}^{k} g(T_i, Q) c'_r \tag{4.2}$$

In above equation, we discard the cost of accessing indexes, as such cost is negligible compared to the data retrieval cost.

Given a data access request, the optimizer estimates the cost of the above two access strategies. If the parallel sequential scan has lower cost, i.e., $c_{pscan} < c_{iscan}$, it is used to process the query. Otherwise, the index scan is used. Periodically, the system runs a background micro-benchmark on the underlying partitioned data store to measure the performance of raw random and sequential I/Os and update the statistic values of c_s, c_r and c'_r, respectively.

4.4.7 Load-adaptive Replication

In large-scale commodity environments, machine failures are not uncommon and hence it is important to provide always-on data service for end-users. Another desirable feature is the ability to adapt to changes in data access pattern. The replicator component in `ecStore` addresses these requirements by using a smart data replication scheme that is able to provide high data availability and load balancing while keeping the cost of replication minimal.

In particular, we propose a two-tier load-adaptive replication strategy. The first tier of replication consists of the primary copy and its secondary replicas which are essential to guarantee data reliability requirement. At the second tier, frequently accessed records are associated with additional replicas, called slave replicas, as a way to re-distribute the heavy load of the "hot" data. These slave replicas will be deleted at later time when the "flash crowd" queries have passed and the cost of maintaining consistency for these replicas has started to increase.

This load-adaptive replication strategy incurs much less replication cost, including storage cost and consistency maintenance cost, than the approach replicating all data records at high replication level, while it can facilitate load balancing. It is noteworthy that the proposed load-adaptive replication is a general scheme which is applied to both base data (managed by primary indexes) and index data (managed by secondary indexes) as will be presented in Section 6.1 (cf. Chapter 6).

4.4.8 OLTP and OLAP Isolation

In `ecStore`, with the support of both OLTP and OLAP workloads, short update transactions run simultaneously with long running ad-hoc analytic queries. The classical locking approach is known to suffer from performance degradation due to blocking and substantial read-write conflicts, and therefore is not suited for this combined workload.

Therefore, the transaction manager in ecStore employs snapshot isolation [40] – a widely accepted correctness criterion and adopted in many commercial database systems – to handle the two workloads simultaneously. Particularly, OLAP queries run in historical mode by accessing the recent consistent snapshot of the data while OLTP transactions work on the current version of the data. ecStore keeps multiple versions of the data in the system. Each version is assigned with a version number. When updating a record, ecStore actually appends a new version of the record to the system. When the total number of versions maintained for a record exceeds the threshold, dead (obsolete) versions will be discarded.

The multiversion strategy in ecStore is made possible by a timestamp-based approach. A loosely synchronized clock in the system is implemented as follows. We discretize the time dimension into epochs. A storage node in the cluster plays as the timestamp authority (TA) and increases the epoch after every period of time (which is configurable by the user). The TA then messages the new epoch to all other nodes in the cluster, and the whole system will move to this new snapshot. Possible failures of the TA can be handled by a standby node. Note that in order to reduce the overhead of handling timestamp generation on the storage node that is selected as the TA, we can delegate this coordination task to a separate service, e.g., ZooKeeper [9, 84], which is widely used in cloud data serving systems like Cassandra [93] and HBase [6] for maintaining configuration information and providing distributed synchronization.

ecStore marks each version of a data record with a timestamp, indicating when it is installed. When an OLAP query q is submitted, the system attaches with it a query timestamp ts. During the processing of an OLAP query, only data records whose version timestamp is just before ts, are used to process the query q. Consequently, with the use of snapshot isolation in ecStore, the put and delete operations in the OLTP interface are actually isolated from the get operation in the OLTP interface and the next operation in the OLAP interface. In other words, ecStore is able to support both OLTP and OLAP queries running simultaneously within the same storage while providing the needed data

freshness for OLAP queries. In Chapter 6, we shall present details of the transaction management techniques used in ecStore.

4.5 Summary

In this chapter, we have proposed a new system architecture for supporting database operations in the cloud and leveraging elasticity of cloud environments. In particular, we have described the design and implementation of ecStore, an elastic cloud data storage system which has been designed to support both OLTP and OLAP workloads within the same storage. The system provides flexible data partitioning scheme, load-adaptive replication, efficient distributed indexes and transactional accesses across multiple records, which are important features but limited in other cloud data serving systems. In the next chapter, we shall present details of the generalized distributed indexing framework developed in ecStore.

Chapter 5

Generalized Distributed Indexing

In the previous chapter, we have described the overall architecture of `ecStore`, our proposed elastic cloud storage for supporting a combined OLTP and OLAP workload. In this chapter, we further present its advanced feature to provide DBMS-like indexing mechanism in the cloud. Specifically, we propose a simple but extensible and efficient distributed indexing framework that enables users to define their own indexes without knowing the structure of the underlying network or having to tune the index performance by themselves.

Likewise centralized databases, `ecStore` makes use of both primary and secondary indexes. In particular, primary clustered indexes store base data of the tables while secondary indexes are non-clustered and designed for supporting queries on non-key attributes. Our distributed indexes adopt peer-to-peer (P2P) network overlays to provide highly available service with no single point of failure. Moreover, using distributed overlays as infrastructures also allows for building an elastic indexing service where index nodes can be added or reduced based on load characteristics. It is important to note that while the proposals in [141, 145] illustrate the feasibility of supporting an index using the underlying overlay network, it is not feasible to support multiple indexes using these approaches since it is costly to maintain multiple overlays – each overlay for a specific index – over the cluster nodes.

In order to provide an indexing functionality that is able to support a variety of basic indexes efficiently, we propose an extensible indexing framework that supports multiple indexes over a single generic overlay. Our framework is motivated by the observation that many P2P overlays are instances of the Cayley graph [106, 117, 121]. Consequently, we abstract the overlay construction by a set of Cayley graph interfaces. By defining the customized Cayley graph instances, a user can create different types of overlays and support various types of indexes such as distributed hash, B^+-tree-like and R-tree-like indexes. This approach avoids the overhead of maintaining multiple overlays while providing flexibility, and it achieves the much needed scalability and efficiency for supporting multiple indexes of different types in cloud database systems.

The main challenge in developing this framework is how to map different types of indexes to the Cayley graph instances. To address this problem, we define two mapping functions, a data mapping function and an overlay mapping function. The data mapping function maps various types of values into a single Cayley graph key space. We propose two data mapping functions: a uniform mapping function, which assumes the data are uniformly distributed and maps data to different keys with the same probability, and a sampling-based mapping function, which maps data based on the distribution of samples.

The overlay mapping function is composed of a set of operators that represent the routing algorithms of different Cayley graph instances. The user can define new types of overlays by implementing new operators, which simplifies the inclusion and deployment of new types of indexes. Additionally, the use of data mapping and overlay mapping reduces the cost of index creation and maintenance. Performance tuning in cloud environments is not trivial, and therefore, our indexing framework is designed to be self-tunable. Independent of the index type, our self-tuning strategies optimize the performance by adaptively creating network connections, effectively buffering local indexes and aggressively reducing the random I/Os.

The remainder of this chapter is organized as follows. In the following section, we discuss the application of distributed indexes in the cloud. In Section 5.2, we give

an overview of the proposed indexing framework. We present details of our proposed Cayley graph-based indexing scheme in Section 5.3. The issues of index self-tuning and failure handling are presented in Section 5.4 and Section 5.5, respectively. We conclude the chapter in Section 5.6.

5.1 Application of Distributed Indexes

Suppose we are designing a large-scale web-based auction system and the schema of *item* table for this application data is defined as follows:

Table 5.1: Sample item data table

item_id	name	price	time	status	owner
1001	iphone	599	2010-10-02	available	2001
1002	htc desire	560	2010-10-03	closed	2002
1003	milestone	389	2010-10-01	available	2003
1004	iphone	520	2010-10-02	closed	2004
1005	ipad	750	2010-10-04	available	2005

To provide a scalable service, we horizontally partition the table among the cluster nodes by the item ID. Therefore, given an *item_id*, the system can efficiently locate the tuple and return the result. However, in fact, most popular queries in the system involve in retrieving items by non-key attributes, such as the *name* of an item, for example:

```
SELECT * FROM item WHERE name='iphone'
```

To answer the above queries, the system can build a hash index on *name*. The index data are distributed among cluster nodes to avoid the bottleneck of a single index node and facilitate parallel query processing. In particular, the cluster nodes are organized as a Chord [130] overlay. In order to retrieve item records with *name* "iphone", we use the value of $hash(iphone)$ to locate the corresponding index node, which is responsible for maintaining the index data of "iphone". Therefore, by traversing the index, all item records related to "iphone" can be retrieved.

As a popular product, hundreds of "iphone" records are returned to the user, who might want to set a filter to refine the results. One commonly used filter is the *price* of an item. Suppose the user sets the price range to $400-$550, the above query evolves as follows.

```
SELECT * FROM item WHERE name='iphone' AND price>400 AND price<550
```

The query can be processed by searching via the index on *name* and then pruning the results by their prices. However, a better solution is to build a 2-dimensional (2-D) index on *name* and *price* together. In distributed environment, we can build a 2-D CAN [122] and use it to realize the 2-D distributed index as proposed in [141].

Besides data retrieval requests from end users, the *item* table is also accessed by the application provider to analyze the users' behavior. For example, the following query is issued by the application provider for calculating the average number of published auctions in last 10 days.

```
SELECT count(*)/10 FROM item
WHERE time≤'2010-10-10' AND time≥'2010-10-01'
```

As this is a range query, we need to build a distributed B^+-tree index on *time* to facilitate the processing. One solution is to use the BATON [86] overlay in order to realize the B^+-tree-like index in distributed environment as proposed in [145].

In summary, to answer different types of queries in a distributed auction system, we need different indexes as in centralized database systems. A number of overlays that support different types of searches have been proposed in the context of peer-to-peer (P2P) systems, and such overlays could be adapted for distributed applications such as the example auction system. As there are three common indexes, namely hash, the B^+-tree and R-tree, which are supported in most commercial database systems, we need to provide equivalent indexes in the distributed environment, and in this research, we identify Chord, BATON, and CAN as indexes that can provide, exact-match, single-dimensional range search and multi-dimensional search respectively. However, each

overlay typically needs to maintain $O(log_2 N)$ neighbors in its routing table and a specific key space for a given search key. Suppose we have hundreds of tables and each table at least has one secondary index, direct application of P2P overlays will not work since the management of the overlays will already contribute to heavy network traffic and computational overhead. Therefore, we propose an indexing scheme with the following features:

Low Maintenance Cost: Instead of maintaining multiple overlays, the indexing framework creates only one generic overlay. All indexes are built on top of this generic overlay.

High Flexibility: The indexing scheme provides simple interfaces, through which users can easily define a new type of index should the need arise.

Scalability: The indexing scheme can efficiently support multiple index structures in a large-scale distributed system.

Performance Self Tuning: The indexing scheme is self-tunable to improve the index performance.

5.2 Overview of the Framework

The proposed indexing framework in `ecStore` consists of an indexing service that provides interface for upper layer components, e.g., the data access manager to perform basic operations on the index data such as insert, update, delete and search. The indexing service runs as a set of distributed processes on the machines in the cluster.

We refer to the index process that runs on a machine as an index node. The index nodes are organized into a generic Cayley graph based overlay [34]. Figure 5-1 plots the overall architecture of our proposed indexing service. The multiple types of indexes supported by the framework are mapped to Cayley graph instance managed by the Cayley Graph Manager. One Cayley graph instance is required for each type of index.

For example, a hash index on a string or numeric attribute is mapped to a Chord [130] instance while a kd-tree or R-tree index on multiple attributes can be supported by a CAN [122] instance. Cayley graph is described further in Section 5.3.1. We define a generalized key space $\mathcal{S}$ for the Cayley graph. A client application simply needs to define a data mapping function $\mathcal{F}$ that maps an index column c to a value in $\mathcal{S}$. Based on the type of indexes, different $\mathcal{F}$s can be defined. If the index is built to support range index, $\mathcal{F}$ must preserve the locality of data. Otherwise, $\mathcal{F}$ can be defined as a universal hash function in order to balance the system load.

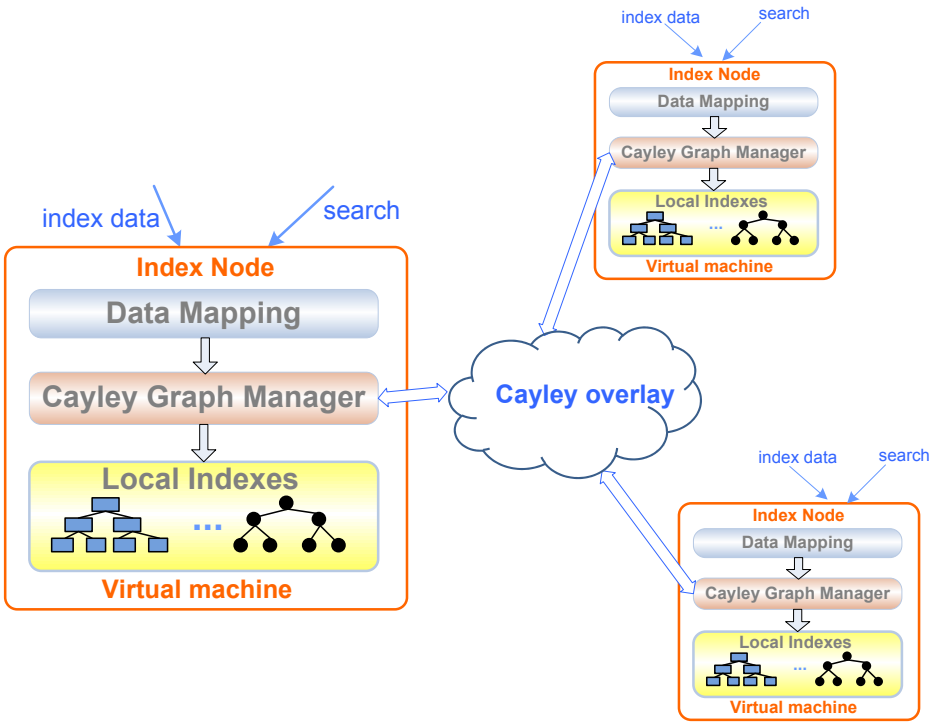


Figure 5-1: Architecture of generalized distributed indexes.

After being mapped with function $\mathcal{F}$, the value of the index keys are normalized to the Cayley graph key space. The detailed description of the data mapping technique is presented in Section 5.3.2. The indexing process in our framework is analogous to the publication process in P2P overlays. Specifically, an index entry is composed of a key-value pair (k, v), where k (referred to as index key) denotes the value of the index column and v (referred to as index value) is the data tuple in cases of primary indexes, otherwise (in cases of secondary indexes) the pointers to the location of base tuples

or possibly portions of the tuples if the user opts to use *covering indexes*. Under the covering index scheme, the indexes include additional, non-key columns in the index entries so that they can service the queries without having to refer to the base records, i.e., the indexes "cover" the queries. To index this (k, v) pair, the indexing framework applies a P2P overlay routing protocol to publish (k, v) based on k', the mapped value of k with the mapping function $\mathcal{F}$. Upon receiving a (k, v) pair and its mapped key k' from the data mapper, the Cayley graph manager retrieves the corresponding Cayley graph instance of the index column and applies the routing protocols of the instance to index the data. Based on the routing protocol of the Cayley graph instance, each index node in the cluster is responsible for a key set and needs to keep a portion of index data (i.e., the published data).

A typical database application such as Human Resource Management or Customer Relationships Management has many tables, and each table has a few indexes. Given the size of the current data sets, the index data will be too large to be maintained in memory. To address this problem, each index node creates a local disk-resident index, e.g., hash table or B^+-tree, for maintaining the index data of a distributed index. Therefore, the index lookup in `ecStore` is performed in two steps. First, it follows the routing protocol defined by the Cayley graph operators to locate the node responsible for the index data. Then, it searches the node's local index to get the index data.

In our framework, the message is routed via the TCP connections between index nodes. Since it is more efficient to send messages via established connections rather than on-the-fly created connections, a connection manager is developed to manage the connection pool based on the query patterns so that only beneficial connections, in addition to the core set of connections required by the routing protocol, are maintained. To further improve the search performance, we employ a buffer manager locally on each index node to reduce the I/O cost for traversing local indexes.

It is important to note that we aim to design a simple yet efficient and scalable solution for indexing data in the cloud. Our indexing system is different from current

proposals [141, 145] in several aspects. First, our indexing system provides the much needed scalability with the ability to support a large number of indexes of different types. Second, our indexing framework is designed as a service which is loosely coupled with the underlying storage system while the indexes in [141, 145] ride directly on the data nodes of the storage system. Decoupling system functions of a cloud system into loosely coupled components enables each component to scale up and scale down independently. Third, our index system is more efficient since in their approaches the index pages of local indexes are selectively published. The selection of index pages is affected the query pattern, and hence can be expensive or they become ineffective after a while if not updated in a timely manner.

The flexibility of the indexing framework lies in the design of its architecture. The user can define a new Cayley graph instance in the Cayley graph manager to support a new type of index, without having to know how the data are partitioned or how the overlays are maintained. The indexing framework automatically handles the underlying implementation and self-tunes the performance.

5.3 Cayley Graph-based Indexing

In this section, we present a Cayley graph-based indexing scheme and use it to support multiple types of distributed indexes such as hash, B^+-tree-like and multi-dimensional index, on the cloud platform. We first present two techniques for mapping multiple P2P overlays of different types to a generic Cayley graph. Then we give details of index operations including index building, index search and index maintenance.

5.3.1 Overlay Mapping

Cayley graph, which is an extension of Cayley theory [79], is initially used as a generic group theoretic model for analysis of symmetric interconnection networks [34]. The formal definition of Cayley graph is as follows.

Definition 1. *A Cayley graph $\mathcal{G} = (\mathcal{S}, G, \bigoplus)$, where $\mathcal{S}$ is an element set, G is a generator set and $\bigoplus$ is a binary operator, is a graph such that:*

1. *$\forall e \in \mathcal{S}$, there is a vertex in $\mathcal{G}$ corresponding to e.*

2. *$\bigoplus : (S \times G) \rightarrow \mathcal{S}$.*

3. *$\forall e \in \mathcal{S}$ and $\forall g \in G$, $e \bigoplus g$ is an element in $\mathcal{S}$ and there is an edge from e to $e \bigoplus g$ in $\mathcal{G}$.*

4. *There is no loop in $\mathcal{G}$, namely $\forall e \in S, \forall g \in G \rightarrow e \bigoplus g \neq e$.*

In a Cayley graph, we create a vertex for each element in $\mathcal{S}$. If for elements e_i and e_j, there is a generator g satisfying $e_i \bigoplus g = e_j$, then edge $(e_i \rightarrow e_j)$ is created.

Based on the definition, we can see that the element set and the generator set actually define the Cayley graph. Figure 5-2 shows a 3-dimensional Hypercube [125] as an example of Cayley graph. In the example, the element set is the binary strings with length 3, namely {000, 001, 010,...,111}. And the generator set is {001, 010, 100}. The operator $\bigoplus$ is defined as the bit-wise "XOR". In Figure 5-2, as 011 $\bigoplus$ 100 = 111, there is an edge between node 011 and 111. To show the role of generators, we mark every edge by its corresponding generator.

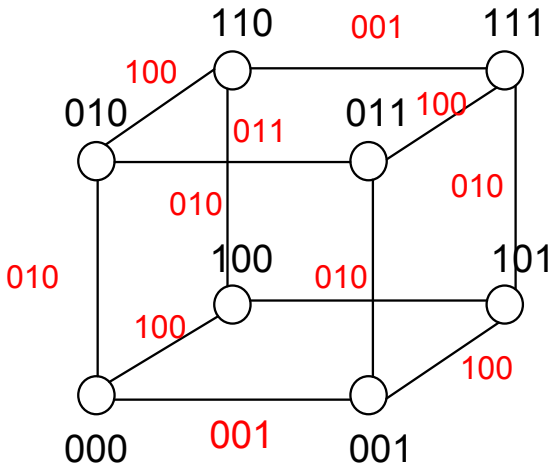


Figure 5-2: An example of Cayley graph.

Each node in the Cayley graph can be considered as a peer node, allowing the Cayley graph to define a P2P overlay. In the following, we formalize the overlay mapping problem so as to provide generic interfaces for mapping popular P2P overlays to the Cayley graph model. This is useful for developing a generalized indexing framework that employs these P2P overlays as underlying distributed data structures.

When a cluster node assumes the role of an index node in the Cayley graph, we use a hash function to generate a unique value in $\mathcal{S}$ as its identifier. Suppose the list of index nodes is $\{n_0, n_1, ..., n_d\}$ and let $I(n_i)$ denote node n_i's identifier. $I(n_i)$ refers to an element in $\mathcal{S}$ and an abstract vertex in the Cayley graph. To support index construction, we partition the element set into subsets with continuous elements based on nodes' identifiers. We first sort the nodes by their identifiers and hence, $I(n_i) < I(n_{i+1})$. The subset $\mathcal{S}_i$ is defined as:

$$\mathcal{S}_i = \begin{cases} \{x | I(n_{i-1}) < x \leq I(n_i)\} & \text{if } x \neq 0 \\ \{x | 0 \leq x \leq I(n_i) \vee I(n_d) < x \leq \mathcal{S}.max\} & \text{otherwise} \end{cases}$$

where $\mathcal{S}.max$ is the maximal element in $\mathcal{S}$. Node n_i is responsible for subset $\mathcal{S}_i$. The overlay mapping problem is formalized as:

Definition 2. ***Overlay Mapping*** *: Given a P2P overlay $\mathcal{O}$ and a Cayley graph $\mathcal{G} = (\mathcal{S}, G, \bigoplus)$, $\mathcal{O}$ can be mapped to $\mathcal{G}$ by using the following rules:*

1. *For a node n in $\mathcal{O}$, $I(n)$ is defined as an element in $\mathcal{S}$.*

2. *Given two nodes, n_i and n_j, n_j is a routing neighbor of n_i, iff there is a generator g in G, satisfying $I(n_i) \bigoplus g \in \mathcal{S}_j$.*

In a Cayley graph, the element set and generator set can be defined arbitrarily, which makes the mapping problem very complex. In our proposal, we fix the element set $\mathcal{S}$ and the generator set G as $\{x | 0 \leq x \leq 2^m - 1\}$ and $\{2^i | 0 \leq i \leq m - 1\}$, respectively. In this way, the overlay mapping problem is transformed into finding a proper operator $\bigoplus$ for each overlay.

Popular P2P overlays such as Chord [130], CAN [122] and BATON [86] can be integrated into the above model since they have been shown to be specific instances of Cayley graph in [121], [117] and [106], respectively. These three types of overlay network are able to support DBMS-like indexes for distributed and cloud environments similar to those commonly available in commercial centralized DBMSes, namely the hash, B^+-tree and R-tree. In what follows, we show the mapping from Chord, BATON and CAN to the Cayley graph abstraction and their respective routing algorithms based on the defined *operators*.

1. Chord

In our framework, Chord [130] is a 2^m-ring since Cayley graph's element set is $\{x|0 \leq x \leq 2^m - 1\}$ (we set $m = 30$ in our experiments to support large datasets). The operator for mapping Chord to Cayley graph can be defined as $x \bigoplus y$ as $(x + y) \bmod 2^m$ [121]. Therefore, given a node n_i and its identifier $I(n_i)$ in the element set S, we create edges between $I(n_i)$ to keys $I(n_i) + 2^k$ $(0 \leq k \leq m - 1)$, based on the generator set $\{2^i|0 \leq i \leq m - 1\}$. Namely, each node maintains m routing entries, which follows the structure of Chord.

Algorithm 1 : `ChordLookup(Node` n_i`, Key` $dest$`)`

```
1. Key start= I(n_i)
2. if start == dest then
3.    return node
4. else
5.    for i=m-1 to 0 do
6.       if (start + 2^i mod 2^m) ≤ dest then
7.          Node nextNode = getNodeByKey(start + 2^i)
8.          return nextNode
```

Algorithm 1 outlines the routing algorithm that simulates the Chord protocol. In Algorithm 1, n_i refers to the index node that receives the routing request and *start* is the node's identifier. Basically, we iterate all generators and try to route the message as far as possible. In line 7, given a key, the function *getNodeByKey* returns the address of the index node that is responsible for the key based on the routing table.

2. BATON

To support BATON overlay [86], the operator $x \bigoplus y$ is defined as $(x+y)\%2^m$ as well since it has been shown that BATON can be transformed to Chord by adding a virtual node [106]. In our case, the key space is fixed to $[0, 2^m - 1]$ for all overlays and thus the maximal level of BATON tree is m. Given a BATON node n_i at level l, suppose its ID at level l is x, n_i can be transformed into a node in Chord by using the following function:

$$\theta(l, x) = 2^{m-l}(2x - 1)$$

The routing neighbors of BATON are similar to those of Chord except for the parent-child and adjacent links. If n_i is a left child of its parent, then the links to its parent node, right child and right adjacent node can be emulated by $I(n_i) + 2^k$. Since BATON has a tree topology, n_i's left adjacent link and left child link cannot be emulated by Chord's routing fingers, and therefore we define new generators ($2^m - 2^x$, x is an integer) to handle the links. However, to keep the framework generic, we choose to use the old generator set, namely $\{2^k | 0 \leq k \leq m - 1\}$. Even without half of adjacent/child links, we note that the queries can still be routed to the destination node using a similar routing scheme as in Algorithm 1.

3. CAN

CAN [122] partitions multi-dimensional space into zones such that each zone is managed by a node. The kd-tree space partitioning provides a good basis for multi-dimensional data indexing. Compared to other overlays, supporting CAN is more challenging, as we need to establish a mapping between CAN's identifiers (multi-dimensional vector) and Cayley graph's key space (1-dimensional value). There are many works on dimensionality reduction, such as space-filling curve [96, 70]. In our current implementation, we adopt the approach proposed in [142]. The basic idea is to partition the search space by each dimension iteratively, and assign a binary ID to each sub-space, which is mapped to the Cayley graph based on its ID. Similarly, a

multi-dimensional query is transformed into a set of sub-spaces, which are denoted by their IDs as well.

The operator $x \bigoplus y$ to map CAN into Cayley graph can be defined as *x XOR y* [117]. With this operator, the basic routing algorithm is outlined in Algorithm 2. We first get the key space of current node (lines 1 and 2). If the search key *dest* is covered by current node, the lookup process can stop by returning current node (lines 3 and 4). Otherwise, we handle the lookup as in the following two cases. First, if this is the first node receiving the search request, the initial key *start* is empty. We iterate all keys in the node's key space to find the one, which has the longest common prefix with the search key *dest*. That key will be used as the initial key *start* (lines 5-10). Second, if the node is not the first node in the search route, the initial key *start* has already been decided by the last node in the route (lines 12 and 13). In either case, we attempt to reduce the difference between *dest* and *start* by routing to a property neighbor (lines 14 and 15).

Algorithm 2 : `CANLookup(Node` n_i`, Key` *start*`, Key` *dest*`)`

1. Key $k_0 = I(n_i.predecessor)$
2. Key $k_1 = I(n_i)$
3. **if** $dest > k_0$ and $dest \leq k_1$ **then**
4. return n_i
5. **if** $start == NULL$ **then**
6. **for** $i = k_0 + 1$ to k_1 **do**
7. $p = getCommonPrefix(i, dest)$
8. **if** $p.length > maxlength$ **then**
9. $maxlength = p.length$
10. $start = i$
11. **else**
12. $p = getCommonPrefix(start, dest)$
13. $maxlength = p.length$
14. Node $nextNode = getNodeByKey(start\ XOR\ 2^{maxlength+1})$
15. return *nextNode*

5.3.2 Data Mapping

Database applications commonly build indexes on various columns for supporting different types of query. These columns typically have different value domains. Since

we use the values of indexed columns as the keys to build the index, before publishing an index entry with key k we need to normalize k into a value k' in the element space $\mathcal{S}$ of the generic Cayley graph. For this purpose, it is also necessary to define key mapping functions, also called as data mapping functions interchangeably. Given an element domain $\mathcal{S}$ and an attribute domain $\mathcal{D}$, the mapping function $f : \mathcal{D} \rightarrow \mathcal{S}$ maps every value in $\mathcal{D}$ to a unique value in $\mathcal{S}$. Note that even different instances of the same overlay type, be it Chord [130], CAN [122] or BATON [86], also need data mapping functions so that they can be realized as a generic overlay with the same key domain in order to avoid the cost of maintaining separate instances of the overlay with different domains.

Suppose we have d index nodes, $\{n_0, n_1, ..., n_d\}$, and use $\mathcal{S}_i$ to denote the element subset of n_i. Given a table T, if the index is built for T's column c_0, the number of index entries published to n_i is estimated as:

$$g(n_i) = \sum_{t_j \in T} \Phi(f(t_j.c_0)) \tag{5.1}$$

where function $\Phi(x)$ returns 1 if $x \in \mathcal{S}_i$, or 0, otherwise.

A good mapping function should provide the properties of ***locality*** and ***load balance*** defined in the following.

Definition 3. ***Locality*** *: The mapping function $f : \mathcal{D} \rightarrow \mathcal{S}$ satisfies the locality property, if $\forall x_i \forall x_j \in \mathcal{D} \wedge x_i < x_j \rightarrow f(x_i) \leq f(x_j)$.*

Definition 4. ***ϵ-Balance*** *: The mapping function $f : \mathcal{D} \rightarrow \mathcal{S}$ is an ϵ-balance function for column c_0, if for any two cluster nodes, n_i and n_j, $\frac{g(n_i)}{g(n_j)} < \epsilon$.*

Locality requirement is used to support range queries, but is not necessary for the hash index. Load balance guarantees that the workload is approximately uniformly distributed over the index nodes. Definitions 3 and 4 can be extended to support multi-dimensional (multi-column) indexes. For the d-dimensional case, the mapping function is defined as $f : \mathcal{D}_0 \times ... \times \mathcal{D}_d \rightarrow \mathcal{S}$, while the locality is measured by the

average value of $L(x_i, x_j) = \frac{d(x_i,x_j)}{|f(x_i)-f(x_j)|}$, where $d(x_i, x_j)$ returns the Euclidean distance between two multi-dimensional points, x_i and x_j.

Definition 5. ***d-Locality*** *: The mapping function $f : \mathcal{D}_0 \times ... \times \mathcal{D}_d \rightarrow \mathcal{S}$ satisfies the locality property, if the average $L(x_i, x_j)$ is bounded by a function of d.*

In our current implementations, we provide two data mapping functions: a uniform mapping function and a sampling-based mapping function.

Uniform Data Mapping

The uniform mapping function is defined based on an assumption that the data are uniformly distributed in the key space. For single dimensional data, as depicted in Figure 5-3, given a key k in the original key space $[l, u]$, we linearly transform it to a new value in the Cayley key space as follows:

$$f(k) = min(2^m - 1, \lfloor \frac{(k-l)2^m}{u-l} \rfloor) \tag{5.2}$$

For example, suppose the original domain is $[0, 10]$ and the key space of Cayley graph is $[0, 2^2]$, keys 6 and 8 will be mapped to values 2 and 3, respectively.

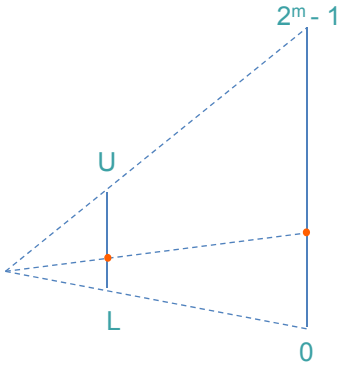


Figure 5-3: Uniform data mapping for one dimensional data.

Theorem 1. *When the data distribution is uniform, Equation 5.2 provides a mapping function that has the properties of locality and 1-balance.*

Proof. Equation 5.2 is a monotonic increasing function and scales the original domain to the new key space using the same factor, $\frac{2^m}{u-l}$. Therefore, it satisfies the two properties. □

For a string value, the uniform mapping function is defined as a hash function h, which maps the string to a value in $\mathcal{S}$. If the index needs to support range queries for strings, h is implemented as a locality sensitive hashing [59].

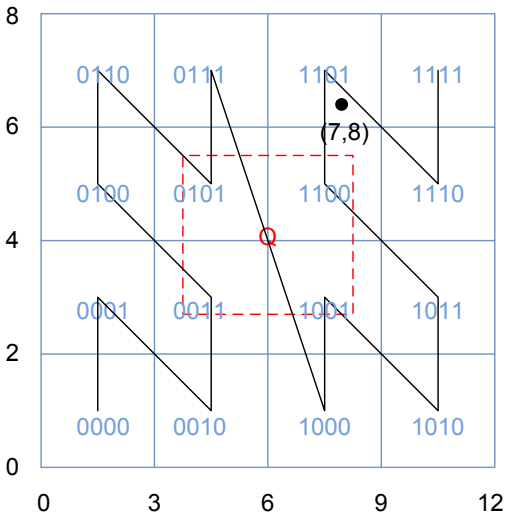


Figure 5-4: Mapping multi-dimensional data.

In the multi-dimensional case, we partition the space in a way similar to kd-tree style [39]. In particular, the key space is partitioned into sub-spaces by different dimensions iteratively. In the xth partition, we partition each sub-space evenly by the jth dimension, where $j{=}x \bmod d$. Suppose we have 2^a sub-spaces before partitioning. The next iteration will generate 2^{a+1} sub-spaces. The partitioning process terminates when 2^m partitions are created. Then, we assign each partition an m-length binary string as its ID, recording its partitioning history. The ID can be transformed back to a value in $[0, 2^m - 1]$, namely the key space of the Cayley graph. It is noteworthy that in uniform mapping, sub-spaces have equal size. Suppose the Cayley graph key space is $[0, 2^4 - 1]$ and the data domains are $x = [0, 12]$ and $y = [0, 8]$, respectively. Figure 5-4 shows how a 2-D space is partitioned. The point $[7, 8]$ is transformed into 1101. Linking the partitions with adjacent IDs generates a multi-dimensional Z-Curve.

Theorem 2. *Z-Curve mapping provides a mapping function that has the properties of d-locality and 1-balance for uniform distributions.*

Proof. Hilbert-curve has been proven to satisfy d-Locality using the metric properties of discrete space-filling curve [74]. The same proof technique can be applied for Z-Curve. Although Z-Curve performs slightly worse than Hilbert-Curve, it still preserves the locality property. As we split the space into equal-size partitions, we also achieve the 1-balance property for uniform distribution. □

Sampling-based Data Mapping

If data distribution is skewed, the uniform mapping function cannot provide a balanced key assignment. Hence, some nodes may need to maintain more index data than the others, which is undesirable. Consequently, we may need to use a load balancing scheme to shuffle the data dynamically during query processing, which is costly. In our framework, a sampling-based approach is used to address this problem.

Before we map and partition the space, we first collect random samples from the base tables to get a rough estimate of the data distribution. We employ the stratified random sampling method since it has been shown to provide both good load balancing and high accuracy of the estimate [126]. This process is performed when the table is initially bulkloaded from external data sources into the cloud databases. Specifically, the system divides the table being sampled into disjoint subsets in the staging phase and takes a specified number of samples from each subset. The formula to calculate the sample size to be taken for estimating multinomial proportions is proposed by Thompson [136].

Based on the retrieved samples, the data mapping function could be defined as illustrated in Figure 5-5. We map the data to $\mathcal{S}$ in such a way that each partition in $\mathcal{S}$ has approximately the same number of samples. In the one dimensional case, the partitioning strategy is equivalent to building an equal-depth histogram [127]. In the k-dimensional case, we apply the kd-tree style partitioning [39], i.e., when partitioning a space we guarantee that the generated subspaces have the same number of samples.

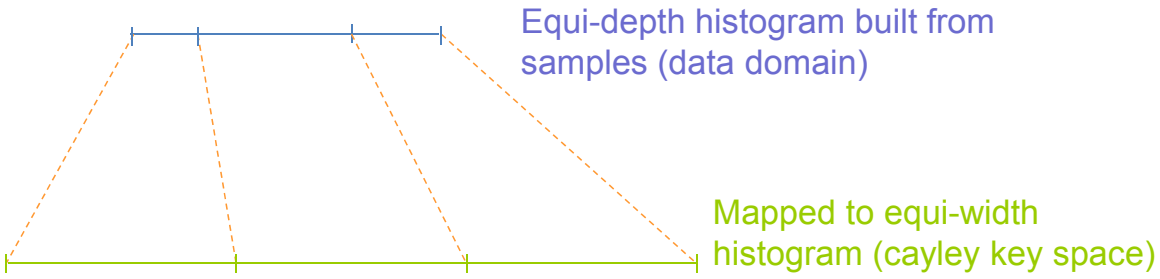


Figure 5-5: Sampling data mapping.

Theorem 3. *The sampling-based mapping keeps locality and provides* $\log_2 N$*-balance (*N *is the total number of cluster nodes), if the samples provide an accurate estimate of the overall data distribution.*

Proof. The proof of locality is similar to the uniform case. Here, we only focus on the load balance property. It has been shown that, for any two cluster nodes, n_i and n_j, $\frac{|S_i|}{|S_j|} < \log_2 N$ [130]. In our sampling-based mapping approach, each sub-space has the same number of samples. When we distribute the sub-spaces in the cluster, node n_i will get k sub-spaces, where k is proportional to $|S_i|$. Therefore, if the samples provide an accurate estimation for the data distribution, sampling-based mapping approach has the property of $\log_2 N$-balance. □

It is noteworthy that bulk insertion from external data sources into cloud databases is a common operation [128]. For instance, in a webshop application, partner vendors publish a large number of new items every day, and the system needs to bulk insert this daily feed of new products into its operational table. In these systems, sampling operations are typically done during this bulk insertion process. Hence, the statistics such as data domains and data distribution could be estimated quite accurately.

It is possible that the above collected statistics might become obsolete due to many skewed online updates after the bulk insertion, and the distribution of the index data among index nodes will become unbalanced as a result. However, when the level of load imbalance among index nodes reaches a predefined threshold, the system can activate a minor data migration process to redistribute the index data, e.g., migrating data to

adjacent nodes to make adaptive changes to the partitions and re-balance the storage load, and update the data mapping function correspondingly.

Moreover, besides the default uniform and sampling-based mapping functions, the user can define his own mapping functions by extending the interface of the framework. More specifically, in this interface, the abstract mapping function is defined. Users can overload this abstract function with customized implementation for specific application and data characteristics. After a mapping function has been linked to an index, our indexing framework will automatically invoke it at runtime to transform the data for indexing and querying.

Query Mapping. The objective of distributed indexes is to facilitate fast retrieval of a subset of data without having to scan every data node. The query optimizer of the client applications will decide if index scan or full table scan should be employed. For queries that involve a small portion of data that fall within a small range, a simple but efficient mapping solution is sufficient to handle such a query pattern. A range query Q, in the one dimensional case, is mapped into a single key range, while in the multi-dimensional case, is transformed into multiple key ranges. For example, in Figure 5-4, the query $Q = \{3.5 \leq x \leq 8.5, 3 \leq y \leq 5.5\}$ is transformed into four key ranges, $[0011, 0011]$, $[0101, 0101]$, $[1001, 1001]$ and $[1100, 1100]$. To retrieve the index for Q, we need to search the four key ranges in the Cayley graph.

5.3.3 Handling High Dimensional Data

For high dimensional data (tens of dimensions), we can break these dimensions into smaller groups and create an index for each of such low dimension group. Take an example table that has a total of 10 attributes: $a_1, a_2, ..., a_{10}$, four of which (from a_1 to a_4) are frequently used in query predicates (e.g., 80% of all queries). The system builds a separate index for each of these frequently queried attributes, while dividing the remaining attributes into smaller groups, say, three attributes in each group for indexing together. Therefore, given a query that has predicate across multiple attributes, we can

choose to traverse the best suited index on some attributes to get preliminary results and then filter out records that are not satisfied with the predicates on other attributes.

When the high dimensional data is skewed and the domain of each dimension only covers some specific ranges, it may be beneficial to build a bitmap index for each dimension because a query predicate on these dimensions can be broken into multiple sub-predicates on each dimension which will be easily processed by the corresponding bitmap index. In fact, the support of bitmap indexes for cloud environments is a big research issue that has been studied in `epiC` [104].

For "very" high dimensional (feature-rich) data such as images and videos, other indexing techniques such as Locality Sensitive Hashing (LSH) [107] tend to be more suitable for similarity search over such high dimensional data. Note that our proposed indexing framework is mainly designed for relational structured data rather than unstructured feature-rich data.

5.3.4 Index Building

Algorithm 3 : `Insert(Tuple` t`, CayleyManager` M`)`

```
1. for every column c_i of t do
2.   if M.isIndexed(c_i) then
3.     Instance I = M.getInstance(c_i)
4.     MappingFunction F = M.getMappingFunction(c_i)
5.     Key k = t.c_i
6.     Node n=I.lookup(F(k))
7.     IndexData v = getIndexData(t)
8.     publish (k, v) to n
```

The generalized indexing framework in `ecStore` has integrated the operators for Chord [130], BATON [86] and CAN [122], as those overlays are used to build the common distributed hash, B^+-tree-like and R-tree-like indexes. The Cayley graph manager considers each operator as a class of overlays. In the initialization process of an index for column c, suppose its operator is op, the Cayley graph manager registers the index as an instance of op. In other words, each operator can have multiple

instances, referring to different indexes on the same type of overlays. The Cayley graph manager also keeps the information of table name, column name, value domain and data mapping function, which it broadcasts to all index nodes to initialize them.

Following the initialization, the index is created for each incoming tuple. Algorithm 3 outlines the indexing process used to insert a new tuple. First the overlay instance of the index is obtained from the Cayley graph manager, which is then used to map and publish the data. In line 6, the *lookup* function is an abstraction of the underlying routing algorithms, which will be transformed into different implementations of the overlay. In line 7, the *getIndexData*(t) function returns the whole tuple t in cases of primary indexes, otherwise (in cases of secondary indexes) the pointer to the location of the tuple t in the primary index or possibly portions of the tuple t if the user opts to use covering indexes. Algorithm 3 demonstrates the extensibility of our indexing framework. It hides all the implementation details, such as how data are mapped and which routing algorithms are used, by providing a highly abstract interface for users.

5.3.5 Index Search

Algorithm 4 : `Search(Key` k`, CayleyManager` M `Column` c_i`)`

1. Instance $I = M$.getInstance(c_i)
2. MappingFunction $F = M$.getMappingFunction(c_i)
3. Node n=I.lookup($F(k)$)
4. Array< $IndexValue$ > $values = n.localSearch()$
5. **for** i = 0 to $values$.size-1 **do**
6. getIndexTuple($values[i]$)

Regardless of the underlying overlays, the $Search$ method can be abstracted as outlined in Algorithm 4. The inputs of $Search$ are a search key and the index column name. The column name is used to identify the specific index and the corresponding overlay. The query engine first asks the Cayley graph manager to get the overlay instance for the column. Then, it invokes the *lookup* method of the overlay, which will return the index node that is responsible for the search key.

In line 4, after receiving the request, the responsible index node performs a search on its local disk-resident indexes to retrieve the indexed data of the key, denoted as a set of index values. If the index being accessed is a primary index or a secondary index which is defined as covering index, then the returned index values themselves are the data of interest for the query. The index search process in cases of primary indexes and covering indexes is illustrated in Figure 5-6.

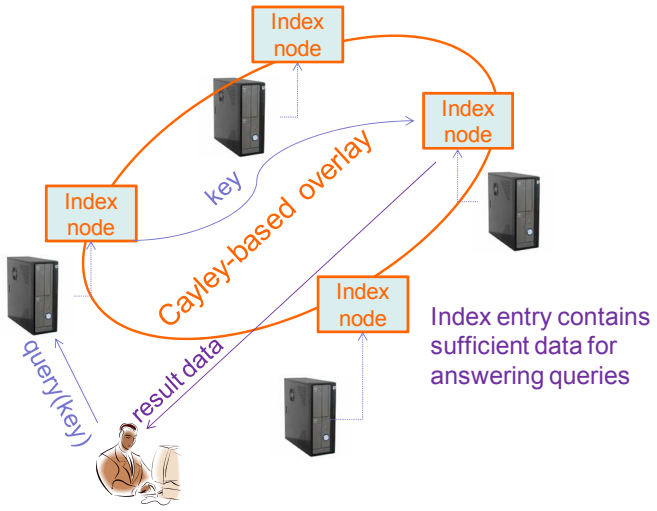


Figure 5-6: Index search with primary indexes and covering indexes.

On the contrary, in cases of secondary indexes without index covering option, i.e., the *index+base* approach, the index values are only pointers to the base records, i.e., the location of the records in the base table that is typically stored as a primary index. In these cases, the system needs to retrieve the records from the base table by querying its corresponding primary index, as shown in Figure 5-7. For clarity, in this figure, the nodes maintaining secondary indexes are referred to as index nodes while the nodes maintaining primary indexes are referred to as data nodes.

Range search can be processed in a similar way, except that in line 3 of Algorithm 4, multiple index nodes could be returned. In this case, the query should be processed by these nodes in parallel. We note that this parallelism mechanism is especially useful for processing equi-join and range join queries. The joined columns of different tables typically share the same data semantics and the index data of these columns are normally

partitioned and distributed over the index nodes in the same way. Therefore, these index nodes can process the join queries in parallel.

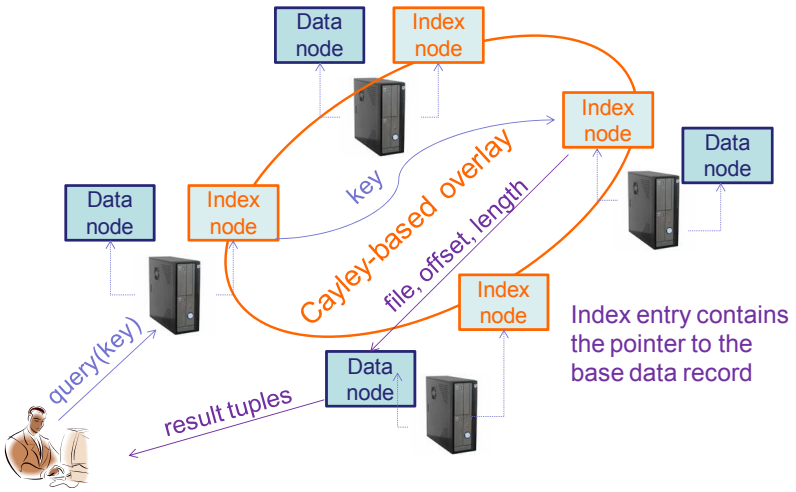


Figure 5-7: Index search with secondary indexes.

In addition, the support of parallel scans of different indexes also facilitates correlated access across multiple indexes, which is necessary when a query accesses multiple indexed columns. It is important to note that the indexing service only provides basic interfaces for upper layer components to access the index data, while the join order between tables is determined by the upper layer query processing engine such as the OLAP controller of `epiC` system (cf. Section 4.1).

5.3.6 Index Update

We now present in detail how the index data are maintained in `ecStore`. For the sake of clarity, we describe the index maintenance process with concrete examples. Consider an employee management application where the information of employees are managed in a table $Emp(EmpID, Name, Salary)$ as a primary index on the $EmpID$ attribute. This primary index is distributed across a set of machines, referred to as data nodes. To support efficient range queries on the $Salary$ attribute, a secondary index is built on that attribute. The machines maintaining the index data of this secondary index are referred to as index nodes.

The maintenance of primary and secondary indexes in `ecStore` in response to update operations from client applications is depicted in Figure 5-8. In particular, for primary indexes, `ecStore` updates the index entries directly in the data nodes. Since `ecStore` employs multiversion concurrency control (cf. Chapter 6), multiple versions of a tuple are maintained with the transaction's timestamp attached to the versions. As shown in Figure 5-8, at timestamp 20 the client inserts a new record for employee 'Tom' whose id number and salary are 8 and 3K respectively, the data node stores this information as $[8, Tom, 3K, 20]$. Then, at timestamp 40, Tom's salary is updated to 4K, and the data node adds another record version $[8, Tom, 4K, 40]$ with the change in salary and timestamp as compared to the initial version.

In `ecStore`, secondary indexes are updated correspondingly when there are changes in the base table. Receiving the update request from clients, `ecStore` updates the primary index and instructs the index maintainer to realize this update to appropriate secondary indexes in two steps. First, the corresponding old index entry (if exists) of the update is logically deleted by attaching the timestamp of the update operation to this index entry in order to invalidate this version. Then, the update is inserted into the index as a new index entry with the timestamp of the update operation as its valid timestamp. So, each index entry in a secondary index is attached with two timestamps, namely valid timestamp and invalidated timestamp, to record the duration when the index entry is still valid. Note that the old and the new index entry may reside on different index nodes because the index key is updated.

As depicted in Figure 5-8, after Tom's record is inserted to the primary index at timestamp 20, a new index entry $[3K, 8, 20, inf]$ is also inserted to the secondary index on $Salary$ attribute. Its invalidated timestamp is marked as 'inf', i.e., infinitive, to show that this index entry is still valid. When Tom's salary is updated to 4K at timestamp 40, the invalidated timestamp of that index entry is changed to 40, viz., $[3K, 8, 20, 40]$, representing that the index entry is only valid during $[20, 40]$, and a new index entry $[4K, 8, 40, inf]$ is inserted to the index in order to reflect the update.

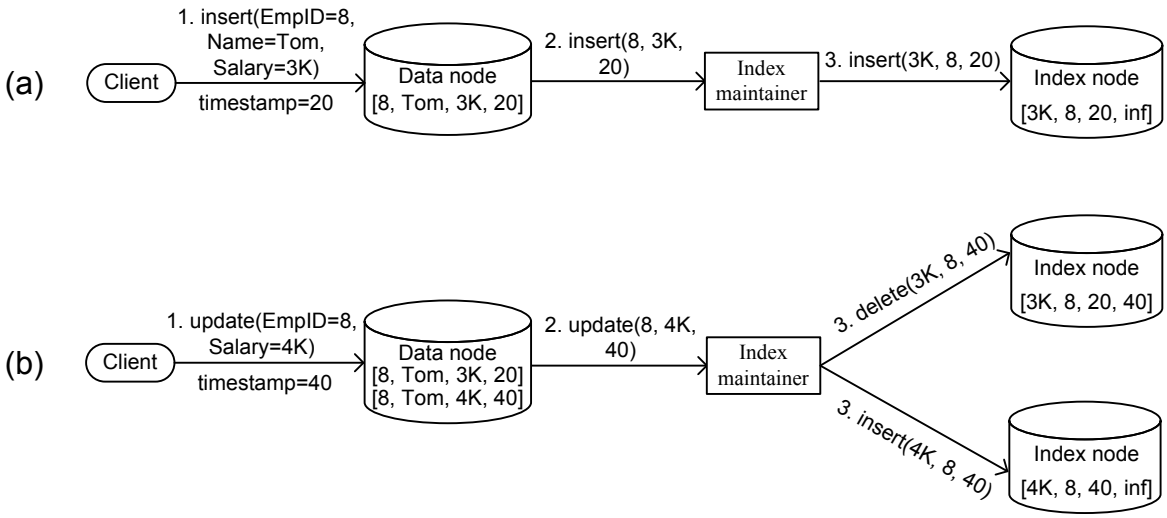


Figure 5-8: Index maintenance: (a) insert a new base record, (b) update index key.

Since old versions of index entries accumulate over time and take up disk space, `ecStore` periodically runs a background pruning process to trim out obsolete versions and reclaim the storage (cf. Section 6.2.7). Also note that update operations might need to modify the local disk-resident index pages. `ecStore` employs a similar approach as B^{link}-tree [98], to guarantee the correctness of concurrent updates to local disk-resident indexes on each index node.

Consistency of Index

It is notable that the enforcement of consistency and ACID properties [76] for index update is based on the requirements of applications. To deal with the common trade-off between consistency and performance, `ecStore` provides two options for setting index consistency: (1) the indexes are updated under strict enforcement of ACID properties and (2) the indexes are updated in a less demanding bulk update approach. The client applications, based on its consistency requirements, will determine the appropriate policy to perform index updates.

In the former approach, `ecStore` needs to reflect all modifications on the base records to the associated indexes before returning acknowledgement messages to users. This approach requires a refresh transaction in order to bring all associated index data

up-to-date with the base data. Since these data, i.e., base data and index data, are possibly located on different machines, a distributed consensus protocol is needed, and therefore affects the update performance.

In the latter approach, `ecStore` backlogs the modification on the base records and performs bulk update to the associated indexes. The frequency of bulk index updates is determined at runtime based on the system workload. For instance, when the system faces a peak load, i.e., a sudden increase in the update rate submitted from clients, it defers the index bulk update to reserve system resources for handling clients' requests first, and then resumes the index bulk update process after the peak load has passed.

5.4 Performance Self-tuning

In this section, we describe how the indexing service self-tunes its performance when maintaining multiples indexes of different types. To improve the performance of the indexing service, we can deploy more index processes. However, in a pay-as-you-go cloud, a more economical way is to optimize the performance of current processes in the service. The main challenge of optimization is the existence of multiple types of indexes. It is impractical for a user or upper layer application to do performance tuning in such a complex setting. Therefore, we identify the factors that may affect the performances for all types of indexes and apply general strategies to improve the global performance of the entire indexing system.

First, the index process routes queries based on the operators defined in the Cayley graph instances; however, it is expensive for each process to maintain active network connections to all other processes. Second, the memory capacity of the index process relative to the application size is limited and all the index entries cannot be cached. To solve these issues, regardless of the index types, our self-tuning strategies optimize performance by adaptively creating network connections at runtime and effectively buffering local indexes.

5.4.1 Adaptive Network Connection

In our framework, the index process routes queries based on the operators defined in the Cayley graph instances. The approach to maintaining a complete connection graph is not scalable since each index process can only maintain a limited number of open connections. Therefore, in our system a connection manager is developed to manage the connections adaptively. It attempts to minimize the routing latency by selectively maintaining the connections. The connection manager classifies the connections as *essential connections* and *enhanced connections*. An essential connection is an active network connection established between two index processes (I_p, I'_p) where I'_p is a routing neighbor of I_p by the definition of any Cayley graph instance in the Cayley graph manager. An enhanced connection is established at runtime between two frequently communicating processes.

By maintaining essential connections, we keep the overlay structures defined by Cayley graph instances. Suppose K types of indexes are defined in the framework for a cluster of N nodes, each node will maintain at most $Klog_2N$ essential connections, with log_2N connections for each type. That is, even if we have thousands of indexes defined for tables using these K types of indexes, we need only $Klog_2N$ essential connections. Queries are mainly routed based on the overlay routing protocols via these essential connections. Enhanced connections can be considered as shortcuts for essential connections. When routing a message, the index process first performs a local routing simulation using the Cayley graph information. Then, it checks the enhanced connections for shortcuts. If no shortcut exists, it follows the normal routing protocols and forwards the message via an essential connection. Otherwise, it sends the message via the available enhanced connections, which is adaptively created during query processing as follows. We use $\mathcal{N}_{es}$ and $\mathcal{N}_{en}$ to denote the essential connections and enhanced connections, respectively. Suppose each node is allowed to support at most S active network connections. The connection manager adaptively creates additional $S - |\mathcal{N}_{es}|$ enhanced connections during query processing as follows.

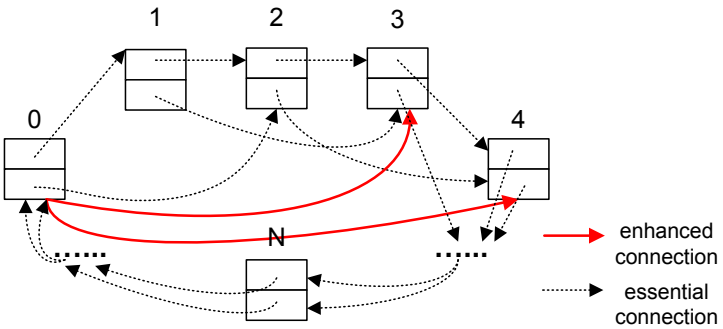


Figure 5-9: Candidate enhanced connections.

To route a query, the connection manager performs a local routing simulation based on the Cayley graph information, and get a path P which is a set of essential connections. For example, in Figure 5-9, every index process maintains 2 essential connections. Suppose we try to route a message from 0 to 3 and the path is 0→1→ 3. We generate enhanced connections by replacing the chain of essential connections in P by a shortcut connection. Specifically, connection $c = (I_p, I'_p)$ is a candidate enhanced connection for P if there exists a shortcut path P' of P, satisfying that P''s starting node is I_p and its ending node is I'_p. We define the length of c as the number of essential connections in P that P' has made the shortcut. In the above example, the candidate enhanced connection is $c = (0, 3)$, whose length is 2.

The connection manager keeps a counter for each candidate connection, recording the number of appearances of the connection during query processing. After every T seconds, the connection manager discards current connections in $\mathcal{N}_{en}$ and adds in the top $S - |\mathcal{N}_{es}|$ frequently used connections to $\mathcal{N}_{en}$. The counters are then reset to 0 and a new tuning iteration starts.

5.4.2 Index Buffering Strategy

Each index process maintains the index data for different Cayley graph instances. To support efficient retrieval of index data, given a Cayley graph instance g, its index data, denoted as $\mathcal{I}(g)$, are stored in a local disk-resident index structure of the index nodes. Based on the type of g, different index structures are built for $\mathcal{I}(g)$. For example, if g is a

Chord [130] overlay, then a hash index is created. Otherwise, if g is an instance of CAN [122], we create an R-tree index for $\mathcal{I}(g)$. Similarly, a B$^+$-tree index is created if g is an instance of BATON [86].

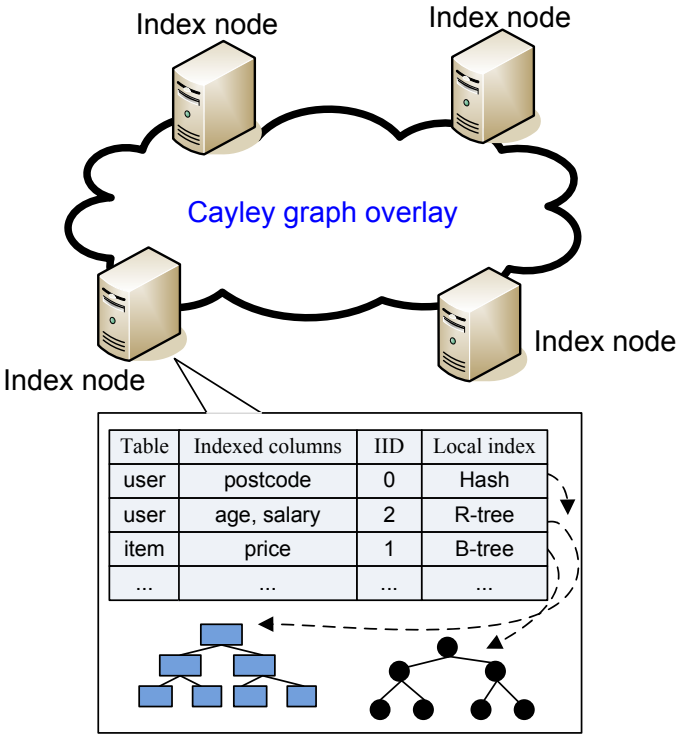


Figure 5-10: Local indexes.

Figure 5-10 shows how the local indexes are maintained. In this example, each index node is responsible for maintaining the index data for three Cayley graph instances. The indexing framework uses the instance ID (IID) to identify a specific instance in the Cayley graph manager. To improve the performance of the local disk-resident indexes of each index node, we buffer some index entries, i.e., the nodes of B$^+$-tree and R-tree, or buckets of hash index, in memory. However, the available memory of the virtual machines hosting the indexing service relative to the application size is limited. Therefore, each index process in `ecStore` establishes a buffer manager to manage the buffer dynamically.

Let $Idx = \{e_1, e_2, ..., e_m\}$ be all index entries on the disk, where e_i represents a node of B$^+$-tree and R-tree, or a bucket of hash index. We define two functions to measure the importance of an index entry. $f(e_i)$ returns the number of queries involving e_i in last T

seconds and $g(e_i)$ is the size of e_i. Suppose M bytes are used to buffer the local indexes, we define a buffering vector $v = (v_1, v_2, ..., v_m)$. When e_i is buffered in memory, v_i is set to 1. Otherwise, v_i equals to 0. The buffer manager tries to find a buffer vector that maximizes

$$\sum_{i=1}^{m} v_i f(e_i)$$

and satisfies

$$\sum_{i=1}^{m} v_i g(e_i) \leq M$$

This is a typical knapsack problem. We solve it using a greedy algorithm. After every T seconds, we periodically run the above algorithm to select the index entries for buffering. The old entries are replaced by new ones to catch the query patterns.

5.5 Failures and Replication

The proposed indexing service is designed to meet service level agreements (SLA) such as 24×7 system availability – an important desideratum of cloud services, which requires the system's ability to handle failures on the index nodes. While there could be fewer failures in a cloud environment relative to the churn experienced in a P2P system, machine failures in large clusters are more common. Consequently, `ecStore` makes use of replication of index data to ensure the correct retrieval of data in the presence of node failures. Details of our proposed load-adaptive replication scheme for Cayley graph-based data structures shall be presented in the next chapter (cf. Section 6.1.3). Here, for the sake of clarity, we briefly describe the method for replicating the index data in `ecStore`.

Specifically, we employ a two-tier partial replication strategy to provide both data availability and load balancing for the indexes. The first tier of replication, which consists of totally K copies of the index data, is designed to guarantee the data reliability requirement (K is commonly set to 3 in distributed systems with commodity

servers [61, 119]). At the second tier, additional replicas of frequently accessed data are adaptively created at runtime based on the query workload as a way to distribute the query load on the "hot" queried data across their replicas.

Given a Cayley graph instance g, its index data, denoted as $\mathcal{I}(g)$, are partitioned and distributed across multiple index nodes. The index data on an index node (primary replica) are replicated to the successors (secondary replicas) of that index node. Note that the successors of an index node regarding to a specific index are determined by the type of g (e.g., Chord [130], BATON [86], CAN [122]), or more specifically, the operator of the Cayley graph (cf. Section 5.3.1). When the primary replica fails, we apply the Cayley graph operator corresponding to the type of index in order to locate its successors and retrieve the index data from one of these replicas. Consequently, the query processing of our indexing service is resilient to the failures of index nodes.

Paxos-based replication algorithm [119] has been shown to be feasible to maintain strict consistency of replicas. However, the trade-off is the complexity of the system and the performance of write operations. When the relaxed consistency is acceptable for client applications, the replicas can be maintained asynchronously instead. Specifically, the primary replica is always updated immediately, while the update propagation to secondary replicas can be deferred until the index node has spare network bandwidth or the peak load has passed.

To avoid the "lost updates" issue, i.e., the updates have not been propagated to secondary replicas due to a sudden crash of the primary replica, the system performs write-ahead logging before updating the primary replica. Therefore, in our indexing service, the updates to the primary replica of the index data are durable and eventually propagated to the secondary replicas. It is noteworthy that if the primary replica fails during the processing of an update, its immediate successor will be promoted to take over the mastership. Therefore, when an index node recovers from a failure, it can retrieve back all latest updates from the secondary replica that previously has been promoted to the mastership.

5.6 Summary

In this chapter, we have proposed a comprehensive and efficient indexing framework for `ecStore` to provide DBMS-like index functionality in the cloud. With a high level abstraction for the definition of new indexes, the indexing framework reduces the maintenance cost and provides the much needed scalability for supporting multiple indexes of different types. To achieve this goal, we define two mapping functions to transform different indexes into the Cayley graph instances. We further exploit the characteristics of Cayley graph to reduce the index creation and maintenance cost, and embed some self-tuning capabilities. In the next chapter, we shall present details of the load-adaptive replication scheme and transactional semantics in `ecStore`.

Chapter 6

Load-adaptive Replication and Transaction Management

In the previous chapters, we have described the overall system architecture of `ecStore` – an **e**lastic **c**loud **stor**ag**e** system that supports a combined OLTP and OLAP workload and provides DBMS-like index functionality for database applications in the cloud. In this chapter, we present details of its load-adaptive replication scheme and transactional support for bundled read and write operations spanning across multiple data records which are possibly stored on different storage nodes in the cluster.

As presented in Chapter 5, `ecStore` organizes its storage nodes as a partitioned data store by the use of a generalized distributed indexing framework that can support various types of distributed data structures such as distributed hash, range and multi-dimensional indexes, which could be either primary or secondary indexes. It is important to note that these distributed indexes originally do not support replication and transactional semantics, which are essential for data management on the cloud platform to provide the required reliability and correctness, while improving efficiency. Therefore, in this chapter, we extend `ecStore` to effectively support load-adaptive replication for the large-scale data maintained in its generalized partitioned data store. Furthermore, we also develop a multiversion optimistic concurrency control scheme on

top of the replication layer. While multiversioning enhances the performance of read-dominant applications, the use of optimistic concurrency control takes advantage of emerging applications where users typically access mutually exclusive data. In addition, a complete recovery control technique developed in `ecStore` guarantees the essential data durability requirement when building a transactional cloud storage on virtual infrastructures. A summary of the proposed techniques in `ecStore` and their advantages are presented in Table 6.1.

Table 6.1: Summary of techniques used in `ecStore`

Problem	**Technique**	**Advantages of the technique**
Partitioning	**Generalized partitioned data store** (hash, range, multi-dimensional)	- Efficient support **multiple types of queries** - **Elastically scalable**
Routing	**P2P** with routing cache	- **No central router** needed - Zero-hop routing cost
Load balancing	Data migration and **load-adaptive replication**	- Data migration **balances the storage load** - Replicating popular data ranges **balances query execution load**
Replication	- **Two-tier partial replication** - The replication process **adapts with database workload** - **Self-tuning range histogram** for access frequency statistics	- Provide **both data reliability and load balancing** function - **Low** replica **storage cost** and replica **consistency maintenance cost** - **Low** cost of **access statistics maintenance**
Replica consistency management	Asynchronous write + quorum read following CAP/**BASE principle**	- **Low write latency** - **Adaptive read consistency**
Transaction management	**Distributed** multiversion optimistic concurrency control with **deadlock prevention** by key ordering	- Favor read-only transactions - **No deadlock overhead**
Recovery control	WAL with treatments for **recovery from short-term and long-term** node failures	**Updates** to primary copies are **durable** and eventually propagated to secondary copies

The remainder of the chapter is organized as follows. In Section 6.1, we propose a load-adaptive replication scheme for the large-scale data maintained in `ecStore`. In Section 6.2, we present details of the transaction management and correctness guarantee in `ecStore`. We conclude the chapter in Section 6.3.

6.1 Load-adaptive Replication

In this section, we propose a two-tier partial replication strategy in `ecStore` to provide both data reliability and load balancing function. The replication process is designed to be adaptive with the database workload. Updates to replicas are asynchronously propagated to ensure low write latency, which is important for cloud storages. The system provides adaptive read consistency by the use of quorum model and allows for trade-off between data consistency and availability.

6.1.1 Replication for Cayley Graph-based Data Structures

`ecStore` stores its data (both primary tables and secondary indexes) in the generalized Cayley graph-based distributed data structures which are deployed on a cluster of commodity machines (cf. Chapter 5). Since machine failures are common in commodity cluster environments, maintaining multiple (K) copies of data in the system is essential to ensure data reliability requirement. In particular, each storage node in a Cayley graph-based distributed data structure replicates its data to $K - 1$ successor nodes on the overlay network. The successors of a node on a Cayley graph-based overlay are identified by the routing information analogous to its specific instance such as Chord [130], CAN [122], and BATON [86], which has been generalized by the operator of the generic Cayley graph as discussed in Chapter 5.

Note that the initial key of data records is also stored with its replicas for verification during query processing. When the primary replica fails, `ecStore` is able to locate its successors based on the routing information of the specific Cayley graph instance. With these replica's locations, `ecStore` can retrieve the data of interest from one of these replicas. Therefore, the system is resilient to the machine failures. The failure assumption is that there cannot be K simultaneous crashes in the system. A replication level of 3 replicas is sufficient to guarantee high data reliability, and most distributed storage systems [7, 68, 119, 61] commonly use 3-way replication as default setting.

6.1.2 Two-tier Partial Replication

We have described *where* to replicate a certain data object in Cayley graph-based data structures. The next key question is *which* data should be replicated. A straightforward approach is to replicate all data objects in the system with the same replication level, K. However, this may not be necessarily good. If K is large, the system storage and the overhead to keep them consistent can be considerably high.

Moreover, Gray et al. [77] show that traditional replication schemes do not scale well and the reconciliation rate for maintaining replica consistency grows as the squares of the number of replicas while the deadlock rate increases as the cube. Additionally, in distributed and web applications databases, the access pattern is often skewed and changes over time. Data migration is often used in range-partitioned systems to deal with skewed data distributions [69, 97]. However, under the skewed query execution load, migrating hot data from an overloaded node to another one only shuffles the hot spot through the system from one place to another.

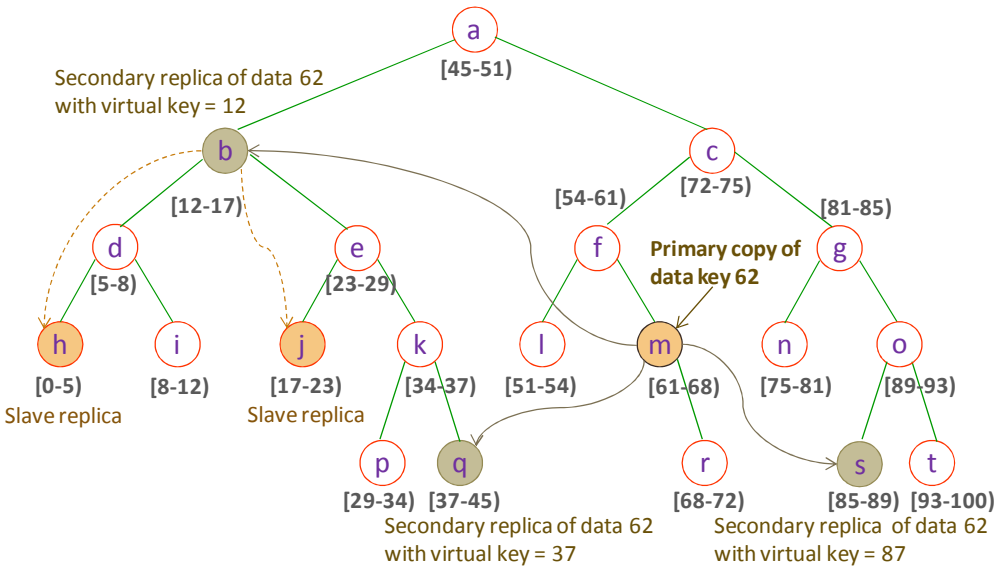


Figure 6-1: Two-tier partial replication.

Therefore, we propose a two-tier replication mechanism to provide both data availability and load balancing function for `ecStore`. In this scheme, each data object (e.g., the data object with key 62 as depicted in Figure 6-1) is associated with *two types*

of replicas - secondary and slave replicas - in addition to its primary copy. The first tier of replication is essentially a level K replication for all data objects, where K is typically a small number. The objective is to maintain a minimal number of replicas, named secondary replica, together with the primary copy for data reliability requirement.

At the second tier, popular data objects are associated with additional replicas, called slave replicas. The purpose is to facilitate load balancing for frequently accessed objects. When a primary copy or secondary replica faces a flash crowd, i.e., sudden increase in query requests, it will create slave replicas (which become associated with it) to help resolve the sudden change in the workload. The initial replica in the first tier maintains pointers to its slave replicas so that it can forward queries to them. The slave replicas will be evicted from the system when they observe more update load than query load. In this way, the costs of replication, including replica storage cost and replica consistency maintenance cost, are always kept minimal. We discuss further on these techniques in the following.

6.1.3 Load-adaptive Strategy

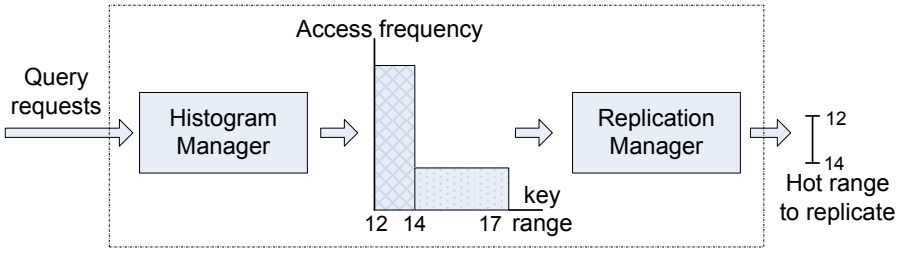


Figure 6-2: Load-adaptive replication workflow.

Figure 6-2 depicts the workflow of the load-adaptive replication algorithm run at each storage node in `ecStore` to selectively replicate data ranges that are beneficial for relieving the hot spot. While the idea of tuning replication process based on data popularity is common, previous works on adaptive replication for classical distributed systems [83, 144] and P2P systems [73, 138] propose to maintain the query access

statistics on a per data object basis. This approach is impractical when the amount of data in the system is large, especially for cloud scale databases. Furthermore, in these works the replication decision for load balancing is based on local workload of each storage node, i.e., an overloaded node will replicate hot data to the nodes on the query routing path. This approach is not necessarily effective since there could be other under-loaded nodes in the system that are more suitable to share the workload.

In this research, we propose a new approach to effectively support load-adaptive replication for large-scale data with low cost of access statistics maintenance. In particular, we use histogram to approximately estimate the access frequency of a data range. The boundary of a bucket forms the two ends of a data range. When the storage node serves a range query (an exact query can be considered as a range query whose start and end value are equal), it will increase the access frequency of all the buckets whose boundaries overlap with the query range.

Consider a storage node S which manages a whole data range R. Suppose there are n buckets in the histogram and r_i is the range of bucket i, we have $\bigcup_{i=1}^{n} r_i = R$. Let Q_i be the *access frequency* of the data range corresponding to bucket range r_i. Then we define the workload $Load(S)$ of node S as the total access frequency of all bucket range r_i.

$$Load(S) = \sum_{i=1}^{n} Q_i \tag{6.1}$$

Since the replication process for load balancing incurs additional overhead to the system, it should not be activated in an ad-hoc manner. Therefore, we consider an approach in which a storage node will trigger the load balancing process whenever its $Load(S)$ increases by a *threshold factor* λ. In particular, we can model the values of $Load(S)$ of a storage node during its operation time as a geometric series of $L_i = c\lambda^i$ where c is a constant representing a unit amount of workload. Thus, when the $Load(S)$ of a storage node increases from L_i to L_{i+1}, we initiate the replication process to balance the query workload across the system.

When the load balancing process is triggered, the storage node will choose m most popular data ranges to replicate to other lighter-loaded nodes in order to relieve the amount of its overloaded load. The replication speed, which determines how many replicas should be populated for the chosen data range, is load-dependent. In other words, the more query load that the data range observes, the more replicas will be created for this range.

Note that when an initially lightly-loaded node becomes heavily-loaded due to an increase in load for its own data or slave replicas residing on that node, the issue is resolved as in the above common case, i.e., each node in the system periodically checks its workload status and performs necessary replication to distribute its sudden overload. In addition, to ensure that a storage node can gather the load information of other nodes in `ecStore`, we piggy-back the load information on the query processing messages and heart-beat messages sent between the storage nodes in the system. The convergence rate of the load statistics information will be studied in the experimental part. Based on this load statistics of other nodes, the overloaded node will choose the lightest-loaded node for replication to shed its work load.

Now, we describe how to determine the *suitable range for each bucket* in the histogram. A straight-forward approach is to use equi-width histogram for maintaining access statistics. Specifically, each bucket is assigned a range approximately to the ratio of the key range R managed by the storage node divided by the number of bucket n. However, this method is not flexible. If we assign a large key range for buckets, the benefit is low cost in histogram maintenance; but the access frequency estimation provided by this histogram is not accurate enough, which results in high cost in replication due to replicating a large data range containing non-popular data objects.

On the contrary, a small key range for buckets guarantees the accuracy of access frequency statistics, thus the replication process is more effective since we only need to replicate the beneficial data ranges. However, the cost of histogram maintenance is high in this case. To solve this problem, in our approach, we employ a self-tuning

histogram, which was first used in [23] for maintaining an estimate of data distribution in a relational table, to get a more accurate estimate of the data access frequency while keeping the histogram maintenance cost minimal. The key idea of self-tuning histogram is dynamically restructuring the histogram, i.e., splitting/merging the buckets, so that the total number of buckets in the histogram is kept constant.

In particular, all the buckets in the histogram are initially assigned equal bucket ranges. At runtime, the buckets will diverge in the value of access frequencies maintained by them: some buckets will have much higher access frequencies than the others due to skewed access patterns. In this case, we merge the consecutive buckets with similar frequency into a bucket with a larger data range and split the bucket with high access frequency into buckets with smaller data range.

With the estimates of access frequency provided by the self-tuning histograms, during the replication process for load balancing we only choose to replicate the data ranges maintained by small buckets because they provide more accurate access frequency estimate and the cost of replicating small data ranges is also cheaper than replicating large data ranges.

It is common that data access patterns change over time. The slave copies of the used-to-be popular data object may no longer serve its purpose and become redundant after a period of time. Hence, we need to reduce the cost of maintaining unnecessary replicas. When a slave replica of a data range does not provide benefit to load balancing anymore, we discard it from the system. For each data range r_i managed by the bucket i in the histogram, we also maintain the data *update frequency* U_i on this data range. When the update frequency U_i of a range is larger than the access frequency Q_i, maintaining the replicas of such a data range only incurs high cost of update propagations. In this case, we remove the information of this data range from the replica list and notify the nodes storing these replicas to discard them from the storage.

6.1.4 Replica Consistency Management

In cloud data storages, it is essential to provide 24x7 service availability. Therefore, propagating synchronous updates to all copies is not a good design choice since the system takes longer time for response to users, and the situation becomes even worse when there are machine failures and/or when these storage nodes are located in distributed clouds [26].

Unlike the proposal in [35] which uses pessimistic replication technique (an update needs to be reflected on all replicas before coming to effect), we employ optimistic replication method in `ecStore`. In particular, the primary copy is always updated immediately, while updates to secondary (and slave) replicas can be deferred. In this optimistic replication method, the single primary copy is the data object indexed with the original key. Secondary copies of a data object form a set of replicas stored on the successor nodes of the primary copy on the index overlay.

Note that `ecStore` provides *adaptive read consistency* by using the quorum model for read operations. A read request is successful only when it collects sufficient votes for a read quorum. If users require strict consistency (desire to access the latest version of a data item), then they might want to configure the value of read quorum to be equal to K, the total number of copies of that data object. In the other extreme, users can set read quorum value to 1 to speed up the read process at the cost of weak consistency, i.e., reads might observe older versions of data. We shall study the trade-off between data consistency and data availability in the below Section 6.1.5.

Algorithm 5 : Read operation

Input: *Query* (key or range)
Input: Read quorum (R)
Output: *data* with latest version observed by the read quorum

1. Send *query* to R replicas
2. *quorum* = CollectReadQuorum[R]
3. $result\_set$ = ExtractData[*quorum*]
4. *result* = SelectDataWithLatestVersion[$result\_set$]
5. **return** *result*

In our scheme, K is normally the number of replicas, including the primary copy and secondary replicas, for data reliability requirements. A read request will collect read votes from these copies. A read request is successful only when it collects sufficient read quorum. Algorithm 5 illustrates the main steps performed by a read operation.

Although there could be an increasing number of replicas created by the self-tuning replication process for load balancing, to process a read request the system still collects votes from above K copies. However, if any copy (among the primary copy and secondary replicas) is overloaded, that copy will redirect the read request to one of the slave replicas attached to it. Thus, in total, we still get K votes in the read quorum. In other words, `ecStore` does not need to track the exact number of replicas corresponding to each data record.

For write operations, we employ optimistic replication, i.e., a write request will update the primary copy first and propagate the effect to secondary replicas asynchronously. The procedure to execute a write operation is given in Algorithm 6. Note that the write operation needs to perform one more step: a secondary replica is responsible to update its slave replicas asynchronously. Nevertheless, this step could be executed periodically and less frequently than the initial propagation from primary to secondary replicas. For example, secondary replicas can send update messages to slave replicas when there is spare network bandwidth.

Algorithm 6 : Write operation

Input: key of data record (*key*)
Input: value of data record (*data*)
Input: number of replicas (K)

1. Construct $data\_record = \{key, data\}$
2. Send *data_record* to the *owner* of *key*
   (the node responsible for *key* based on Cayley graph routing scheme)
3. Asynchronously replicate *data_record* to $(K-1)$ $successors$ of the *owner*
   (the nodes in the *routing table* of the *owner*)

The use of optimistic replication allows `ecStore` to provide high responsiveness and data availability for users. Nevertheless, by CAP theorem [72], any distributed system faces the trade-off between availability and consistency. In this case, there is a

possibility that the modification to primary copy gets lost when this operation has not been propagated to other secondary copies before the primary copy crashes suddenly. In ecStore, we adopt the write-ahead logging scheme and devise a recovery technique (cf. Section 6.2) to deal with the problem of "lost updates" due to different types of node failures. Thus, ecStore ensures that updates to the primary copy are durable and eventually propagated to secondary copies, i.e., it provides eventual replica consistency similar to other cloud data serving systems like Dynamo [61] and Pnuts [54].

It is also noteworthy that ecStore guarantees the *order of modification* done by different users to be the same on each replica in spite of the asynchronous update propagation process. As we shall discuss in Section 6.2, ecStore is designed as a version-based storage system: each data object is attached with a transaction commit number, which is monotonic increasing in the system. Based on this version number, the replica of a data object can order the updates propagated to it correctly.

In summary, ecStore adopts the notion of BASE (BAsically available, Soft state, Eventually consistency) [116] to deal with the issue of maintaining replica consistency. In this way, it avoids the need to implement a costly two-phase commit protocol for the refresh transactions in order to keep replicas up-to-date with the primary copy as in the case of synchronous replication.

6.1.5 Trade-off between Data Consistency and Availability

Now we study the trade-off between the user's observed data consistency and the data availability of the system when the read quorum varies.

Data Consistency

The *adaptive read consistency* property in ecStore can be formalized as the probability of getting the most recent replica, i.e., the replica that contains the latest update to the record, when a read request receives R votes from the read quorum. We call this the *consistency probability*, which is computed as follows.

There are totally C_R^K cases of selecting R read votes out of K replicas. Assume that we have received the vote containing the most recent replica, then there are C_{R-1}^{K-1} cases of choosing $R-1$ read votes out of $K-1$ replicas to get enough votes for the read quorum. Hence, the probability of reading the most recent replica is:

$$Consistency\_Probability(K,R) = \frac{C_{R-1}^{K-1}}{C_R^K} = \frac{R}{K} \tag{6.2}$$

Equation 6.2 agrees with the intuition that with a certain total number of replica (K) , a larger read quorum (R) provides a higher probability of reading the most recent replica.

Data Availability

Another characteristic of a distributed system with replicated data is *data availability*, which can be formalized as the probability of getting enough votes for the read quorum. A formula for data availability in replicated environments has been studied in [68]:

$$Availability\_Probability(K,R) = \frac{\sum_{i=0}^{K-R} C_i^K \rho^i}{(1+\rho)^K} \tag{6.3}$$

where K is the number of replicas, R is the size of read quorum, ρ is the probability a storage node in the cluster fails.

The Trade-off between Data Consistency and Data Availability

Figure 6-3 depicts the trade-off between data consistency and data availability with different values of the total number of replicas K and the read quorum R. Particularly, with a certain value of K, the *data consistency* level increases together with the value of read quorum R. In contrast, the *data availability* level decreases when the value of read quorum grows since the larger read quorum will reduce the probability of getting sufficient number of votes for the read quorum.

It can also be observed from Figure 6-3 that there is not a clear equilibrium point between each pair of the curves representing data consistency and data availability. In

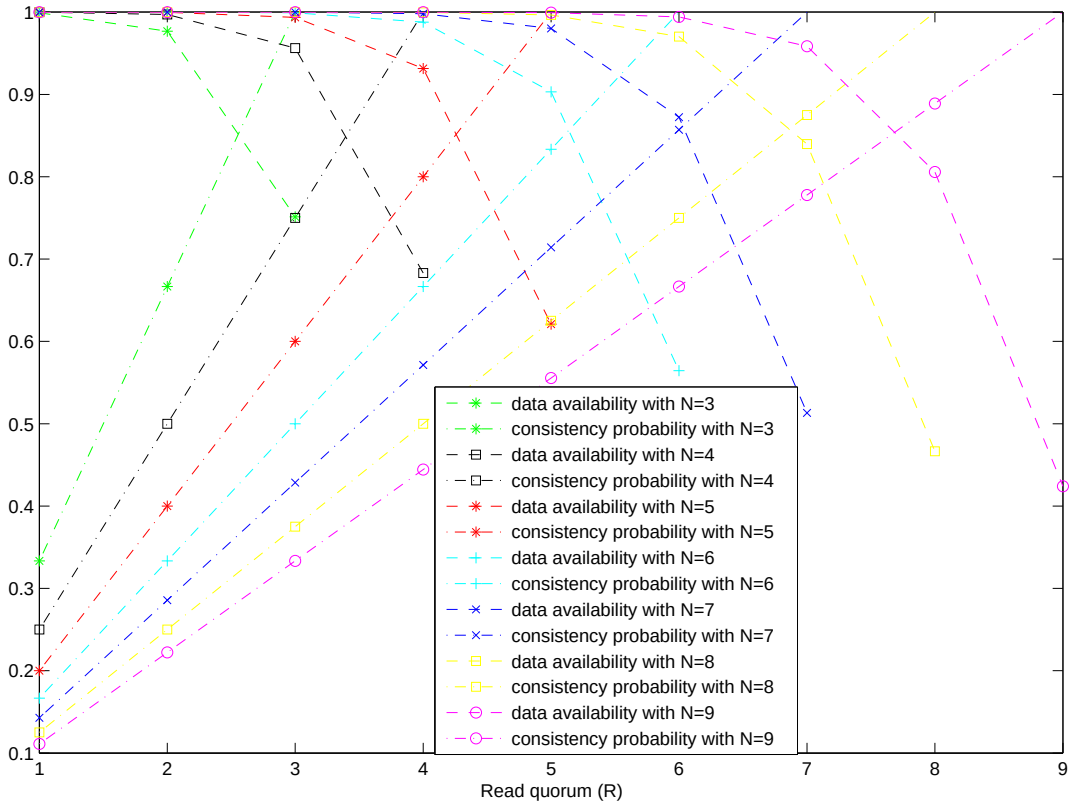


Figure 6-3: The trade-off between data consistency and data availability.

other words, there is no optimal setting of read quorum that satisfies both high data consistency and high data availability. Therefore, it is important to choose the value of the read quorum that balances the requirement of data consistency and data availability, i.e., ensuring acceptable data consistency with sufficient level of data availability. For example, in a system with 3-way replication, a read quorum that requires two votes provides high probability of accessing the replica containing the latest update without paying much reduction in data availability.

6.2 Transaction Management

Different parts of the system can choose different points in the spectrum between BASE [116] and ACID [76]. As described above, `ecStore` adopts the notion of BASE for managing replica consistency. Nevertheless, since it is desirable to provide transactional

semantics for bundled read-modify-write operations in cloud storages, in this section we present how ecStore ensures ACID properties for its transaction management.

6.2.1 Concurrency Control

Design Considerations

In general, data in cloud storages possess two typical characteristics. First, it is usually sufficient to perform operations on a recent snapshot of data rather than on up-to-second most recent data [22]. Second, the locality of data accessed by transactions: the data tend to be independent between concurrent transactions of different users since users are more likely to operate on their own data, which forms an entity group as characterized in [37, 82]. In addition, the careful design of data partitioning strategy in ecStore, as presented in Section 4.4.2, splits the data into sub partitions while reducing number of cross transactions between partitions. Therefore, with high probability, concurrent update transactions issued from different users typically modify data in separate key groups, and thus incur little data contention.

The above characteristics of cloud data drive the design of the concurrency control technique in ecStore. A hybrid scheme of multiversion optimistic concurrency control (MVOCC) becomes a good candidate to implement isolation and consistency for cloud-scale databases. The essence of this approach is that multiple versions of data can benefit the read-only transactions, while the optimistic method protects the system from the locking overhead of update transactions. In what follows we present the rationale of combining the multiversion and optimistic scheme.

It has been a consensus that locking approach may suffer from problems such as the lock maintenance overhead and the lack of deadlock-free locking protocols for general-purpose applications. In addition, in environments with little resource contention, locking may be unnecessary in most cases, and transactions could be allowed to optimistically execute while possible conflicts among concurrent

transactions will be validated later when these transactions enter their commit phase. With this optimistic scheme, there is no blocking caused by the locks, and thus the system performance is improved in query-dominant environments.

The main shortcoming of the optimistic concurrency control scheme is that transactions may be restarted unnecessarily and even a read-only transaction may have to abort due to data conflicts with other transactions committed during its execution time. Generally, there are two potential ways to reduce the data contention among the concurrent transactions in the system. One possible way is to compromise the data consistency by running queries at non-repeatable read or dirty-read isolation level [40]. This approach, nevertheless, suffers from a certain level of serializability violation. Another promising way is to compromise the timeliness of the data by the use of versioning to avoid conflicts between the read-only and update transactions. In this method, multiple versions of data are maintained to allow queries to run against consistent snapshots of the database. Hence, read-only transactions are serializable before other concurrent update transactions and more importantly, there is no concurrency control overhead for read-only transactions.

Multiversion Optimistic Concurrency Control (MVOCC)

In this hybrid scheme, each transaction has a startup timestamp, which is assigned when the transaction starts, and commit timestamp, which is set up during the commit process. In addition, each data object also maintains the commit timestamp of its most recent update transaction. When a transaction accesses a data object, the most recent version of the data with a timestamp less than transaction's startup timestamp is returned. Thus, no locking overhead is incurred by the read requests.

A major advantage of MVOCC is the separation of read-only transactions and update transactions so that they will not block each other at runtime. That is, read-only transactions access a recent consistent snapshot of the database while update transactions operate on the latest version of the data. Therefore, read-only transactions

always commit successfully without the need to check conflicts with other transactions. In contrast, an update transaction after finishing its read phase has to validate its possible conflicts with other concurrently executing update transactions before being allowed to enter the write phase.

While traditional OCC needs to store old write-sets of committed transactions just for the purpose of verifying data conflicts [92, 28], the MVOCC in `ecStore` provides another advantage that in the validation phase of update transactions, the transaction coordinator can use the version numbers of data records to check for conflicts with other update transactions. In particular, to commit an update transaction T, the transaction coordinator checks whether T's write set are updated by other concurrent transactions that have just committed by comparing the versions of the records in T's write-set that T has read before (i.e., there is no blind write) with the current version of the records. If there is any change in the record versions, then the validation fails and T is restarted. Otherwise, the validation return success and T is allowed to enter the write phase and commit the transaction.

Algorithm 7 : Validation with write lock

Input: write-set (WS) of the validating transaction T
Input: server S where T is validated
Output: *valid_state* of the validation process

```
1.  valid_state := true
2.  for all data record Rec in WS(T,S) do
3.     if lock_conflict(Rec) then
4.        valid_state := false
5.     else
6.        Acquire lock(Rec)
7.     if read_version(T,Rec) < latest_version(Rec) then
8.        valid_state := false
9.  if valid_state = true then
10.    Send COMMIT message to the transaction coordinator
11. else
12.    Send RESTART message to the transaction coordinator
13. return valid_state
```

It is noteworthy that concurrent writes, i.e., multiple transactions execute the write phase at nearly the same time, might incur inconsistency issues if cares are not taken.

More specifically, if two conflicting transactions whose write-sets overlap perform validation simultaneously, they might both succeed since the corresponding data versions have not been changed by any transaction; however, when these transactions actually enter the write phase, inconsistencies might occur. In order to avoid this problem, `ecStore` embeds write locks into the validation phase of MVOCC, as outlined in Algorithm 7.

In particular, an update transaction first executes its read phase as per normal; however, at the beginning of validation phase, the transaction coordinator will request write locks over the data records for its intention writes (lines 3 - 6). If all the locks can be obtained and the validation (lines 7 - 8) succeeds, the transaction is allowed to start the commit process (line 10) which executes its write phase and finally releases the locks. On the contrary, if the transaction coordinator fails to acquire all necessary write locks, it will still hold the existing locks while re-executing the read phase (line 12) and trying to request again the locks that it could not get in the first time. In other words, the transaction keeps pre-claiming the locks until it obtains all the necessary locks, so that it can enter the validation phase and write phase safely.

It is complicated to detect and resolve deadlocks at runtime [118], and the problem is even more challenging in large-scale distributed environments such as cloud. Therefore, `ecStore` chooses to avoid deadlocks by enforcing each transaction to always request the locks in the same sequence, e.g., based on the order of records' key. For instance, if both $T1$ and $T2$ desire to lock a write set $\{x, y\}$ where $x < y$, each of them has to request $lock(x)$ and $lock(y)$ in sequence so that no transaction requests and waits for locks on new items while still holding locks on other transactions' desired items. This approach is inspired from a classical method that an operation system uses to prevent deadlock when handling resource allocation [134]. More specifically, it classifies the resources of a computer into categories assigned with priority levels, and applications running concurrently on the computer are to request their needed resources in the order of the priority level of the resources.

Note that although ecStore employs write locks in its algorithm, it is different from traditional two-phase locking (2PL) in that the transaction only holds write locks for a short period during validation and write phase rather than the whole transaction execution time as in 2PL. Furthermore, since it is challenging to maintain distributed lock tables in a dynamic environment, ecStore delegates the task of managing distributed locks to a separate service, Zookeeper [9], which is widely used in distributed storage systems such as Cassandra [93] and HBase [6] for providing efficient distributed synchronization.

6.2.2 Correctness Guarantee

As described above, in ecStore, the method to obtain write locks during the validation and write phase of transactions helps prevent conflicts of concurrent writes, thereby ensuring the "first-committer-wins" rule [40]. With this property, the implemented MVOCC, which is a type of multiversion concurrency control, provides similar consistency and isolation level to the standard snapshot isolation level which was first formalized in [40].

Note that snapshot isolation is a widely accepted correctness criterion and adopted by many centralized open-source as well as commercial database systems such as MySQL, PostgreSQL, InterBase, Oracle, and SQL Server. Therefore, we believe that it is also useful for large-scale distributed environments such as cloud, which is the targeted environment of ecStore.

If strict serializability is required, read locks also need to be acquired by transactions [28, 135], but that will adversely affect transaction performance as read locks block the writes and void the advantage of snapshot isolation. Another approach called serializable snapshot isolation [44] detects potential cyclic "read-write" dependency at runtime and restarts one of the involving transactions. However, this approach might abort transactions unnecessarily and is complicated to implement in distributed environments.

6.2.3 Interaction between Transaction and Replication

Now, we describe how ecStore handles the interplay of transaction consistency and and replica consistency. In ecStore, read-only transactions will access the replicas for load balancing purpose, so that the primary copy will not be the bottleneck under skewed workloads. In addition, the consistency of the replicated data observed by the read-only transactions is tunable with the quorum model (cf. Section 6.1.5). That is, users can set the quorum parameter to appropriate values based on the consistency requirements of their applications.

However, the update transactions are always required to access the primary copy of data, both in the read phase and write phase, to ensure that the updates in ecStore are well-behaved. Additionally, ecStore uses *mastership failover* to handle unsuccessful updates on the primary copy; if the primary copy fails during the processing of an update transaction, one of the secondary copies will be promoted to take over the mastership. It will store the updated value of the data, and then wait for the primary copy to recover and finally send back the updated value to the primary copy.

In summary, ecStore provides snapshot isolation for primary copy of data, while replicas are kept asynchronously updated with the primary copy to ensure low write latency, which is important in cloud storages. This design choice also allows for tunable read consistency across replicas as a trade-off for read performance.

Although there are solutions that provide standard snapshot isolation and serializable snapshot isolation for replicated databases such as [100, 88], these works mainly focus on full replication of a centralized database to a small number (tens) of nodes with ROWA (read-one write-all) approach. Concurrent transactions at different replicas are to be finally certified (checked for possible conflicts with others) by a centralized certifier or a group communication protocol, which restricts the scalability of the system.

In contrast, ecStore is inherently designed as a large-scale distributed storage system that partitions data across many (possibly hundreds of) nodes and employs

partial replication. Therefore, we relax the consistency of replicas compared to its primary copy (via asynchronous update propagation) to meet the latency requirement and deal with the scale of cloud storages.

6.2.4 Timestamp Management

The benefit of multiversion optimistic concurrency control scheme does not come for free. The challenging task when implementing this hybrid scheme in `ecStore` is how to ensure a global order of all committed update transactions in a dynamic distributed environment. In `ecStore`, a certain storage node is chosen as the commit-number generator, or also referred to as timestamp authority (TA).

Typically, the first storage node in the cluster will assume this role. The TA also chooses other two storage nodes in the cluster as its standby successor. In our implementation, we randomly select two nodes in the cluster that have just sent some messages (e.g., the query processing messages) to the TA. In case the TA fails, one of its two successors can take over the role. Moreover, the contact information of the TA and its standby successors can be easily maintained at each storage node in the cluster. Piggy-backing this information on the periodical heartbeat messages sent between storage nodes in the cluster is sufficient.

When an update transaction successfully validates against other update transactions which have committed during its execution time, it will get a commit-number, i.e., the timestamp for commit, from the TA. The TA guarantees to generate monotonic values over the sequence of requests from the update transactions by increasing the value of latest committed timestamp before returning the new committed timestamp.

Note that only update transactions need to contact with the TA after successful validation phase; hence, the TA is not the critical point of failure. In addition, the latest committed timestamp is replicated on storage nodes in the cluster and also piggy-backed on the query processing messages and heartbeat messages sent among the storage nodes. In this way, each storage node can cache the recent committed

timestamp and use this as the start timestamp for the coming transactions. Further, we can delegate the task of generating timestamps to a separate service, e.g., ZooKeeper [9, 84], to reduce the overhead on the storage node selected as the TA.

6.2.5 Commit Protocol

In the above section, we have addressed the concurrency and isolation issue in `ecStore`. Now, we consider two other desired properties, atomicity and durability, which require that all or none of the updates of a transaction come into effect and the modifications which have been confirmed with users should be persistent in the storage.

The properties of atomicity and durability in `ecStore` are guaranteed by the commit protocol and recovery control. By adopting multiversion optimistic concurrency control, all read-only transactions would always succeed because they only access data in a consistent snapshot of the database. The timestamp to identify a snapshot is the commit numbers of committed update transactions in the system. Since optimistic concurrency control defers update effect until commit time, we can piggy-back its concurrency control information for validation phase on the messages of the commit protocol as illustrated in Algorithm 8.

Algorithm 8 : Commit protocol at transaction coordinator

1. *Send validation requests* to cohorts
2. *Collect vote messages* from cohorts
3. **if** all validation successful **then**
4. *Get commit number* from the timestamp authority
5. Store and replicate the *log records* and *commit record*
6. Send *COMMIT message* to cohorts
7. **else**
8. Send *RESTART message* to cohorts

It is important to note that in the commit algorithm, when all the transaction participants have positively voted, the transaction coordinator will store the log records and commit records to its local disk and also replicate these records over the storage nodes in the cluster for durability. This information is useful for the recovery process

from different types of node failures. When all the updates of a committed transaction have been successfully propagated to other replicas, the storage node can safely delete the log records and commit record for this transaction. Therefore, the size of the log store is not large.

Also note that a careful design of data partitioning scheme, as presented in Chapter 4 (cf. Section 4.4.2), is useful to reduce the cost of managing distributed transactions in the system. There are also available solutions in the literature for improving the performance of the two-phase commit such as the non-blocking Paxos algorithm [94, 119].

When deploying `ecStore` on virtual infrastructures such as Amazon EC2 [2], the storage nodes (virtual machines) do not have dedicated disks to store the transaction log records. However, when `ecStore` is set up to run directly on physical hardware (e.g., an in-house cluster), installing dedicated disks for storage nodes can help to improve I/O performance since `ecStore` can write data and log entries to separate disks.

6.2.6 Recovery Control

In `ecStore`, a storage node can leave the system in two manners. In the case of safe departures, a storage node will notify appropriate nodes in the cluster, transfer any of it roles and data and safely leave the system. No recovery process is needed for this case. However, we need to take the case of unsafe departures into account. We divide the unsafe departure into two types of failures with different recovery treatment for each: short-term failure (due to software bugs or communication failure) and permanent failure (mainly due to hardware crashes or the virtual machine is terminated).

When a storage node rejoins the system after a short-term failure, it will check its local log store to see whether there is any log record of committed transactions coordinated by itself that has not been sent to other transaction participants. These log records will be forwarded to the involving storage nodes to finish the commit process. In this way, transactions in `ecStore` are durable. The effect of committed transactions are persistent even when the transaction coordinator fails before sending the commit

commands to other transaction participants. Another important point is that since ecStore uses mastership failover, we also need to get the primary copy of data on the failure storage node up-to-date. Particularly, the secondary copy that previously is promoted to mastership will periodically ping to check whether the primary copy has recovered and send back the updated value to the primary copy when possible.

Now, we describe the recovery control in the case of long-term failures. When a storage node suffers from a long-term crash, another healthy node will be chosen to take care of the range index that previously is managed by the failure node. Then the recovery process proceeds in two main steps. First, the new responsible node will recover the data in that range by copying the corresponding replicated data from other nodes in the cluster. Note that we copy back the latest version of data among the secondary copies.

Second, the new responsible node will check the transaction logs replicated in the cluster to see whether there is any log record of committed transactions coordinated by failure node that has not been executed at the transaction participants. The new responsible node will perform redo operations by forwarding the log records to the involving storage nodes to materialize all the effects of the committed transactions. Hence, the update transactions in ecStore are durable even in the case of long-term crashes of storage nodes. Note that redo operations are sufficient for the long-term failure recovery process since ecStore follows optimistic concurrency control scheme, which defers all updates until commit time.

6.2.7 Version Pruning

We have proposed to integrate both replication and multiversion technique into ecStore. In fact, each technique has its own purpose. Replication helps to increase data availability of the system while multiversion scheme supports higher transaction concurrency. As depicted in Figure 6-4, there exist nine instances for a data item in the system which maintains three replicas for each data item with a history of three versions. Therefore, a large amount of the total system storage might be merely used

for replication and multiversion purpose, which reduces the storage utilization. Consequently, mechanisms to prune old versions of data are needed.

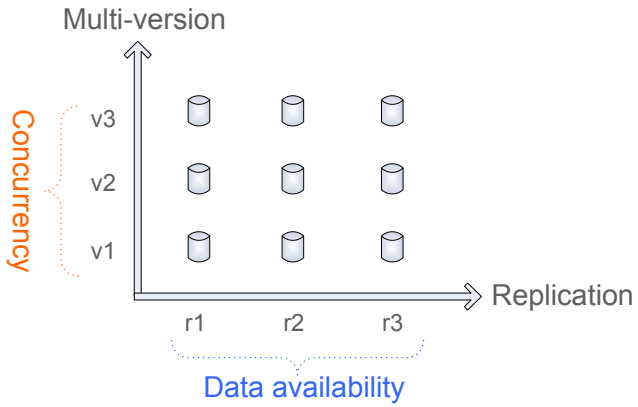


Figure 6-4: Instances of a data object with multiversion and replication technique.

A practical method to trim obsolete versions of data is the use of a version threshold. We only prune the versions whose version timestamps are more obsolete than the threshold. The value of the threshold affects the system in two ways. If we set the version threshold to be too large, then the storage is not effectively utilized. In contrast, small version thresholds might make more transactions to be aborted because they can not access the data versions which are in the snapshot before their start timestamps (all these obsolete versions have been thrown away by the pruning process). Knobs can be provided for users to tune the version threshold value. Moreover, the system can monitor the number of transactions aborted due to accessing pruned versions and automatically balance the version threshold and storage utilization.

6.3 Summary

In this chapter, we have described the load-adaptive replication scheme and transaction management in `ecStore`. The self-tuning replication technique, which is specially designed for large-scale data, is effective for balancing query execution load in the system. Furthermore, the multiversion optimistic concurrency control scheme in `ecStore` matches well with the characteristics of cloud data. With a complete method

for system recovery from different types of node failures, `ecStore` guarantees data durability, which is an essential service level agreement (SLA) when providing storage services on the cloud virtual infrastructures. In the next chapter, we will evaluate the performance of `ecStore` extensively on various cloud platforms.

Chapter 7

System Evaluation

In previous chapters, we have presented the design and implementation of `ecStore` – an elastic cloud storage that provides advanced features for cloud database applications. The features includes smart replication for both high data availability and automatic load balancing, transactional support for bundled read-modify-write operations, distributed indexing for efficient processing of queries on non-key attributes, and hybrid storage structure for supporting both real-time and analytic workloads.

To validate our design and implementation, in this chapter we perform an extensive series of experiments to study various performance aspects of `ecStore` such as system scalability, efficiency, and robustness. Specifically, in Section 7.1, we describe various cloud environments in which we conducted the experiments. The evaluation of distributed indexes is presented in Section 7.2, while the evaluation of replication and transaction management is provided in Section 7.3. We evaluate the overall system and compare it with other systems in Section 7.4.

7.1 Experimental Environments

We experimented `ecStore` on various cloud platforms including an in-house cluster serving as private cloud, the commercial public cloud Amazon EC2 [2], and PlanetLab [17] – a testbed that represents distributed cloud.

7.1.1 In-house Cluster

In-house clusters are commonly used to implement **private cloud** for internal usage in most enterprises and Internet companies. To study the performance of `ecStore` in private cloud environment, we deployed the system in an in-house commodity cluster, named `awan`, which is constructed for the `epiC` project [1].

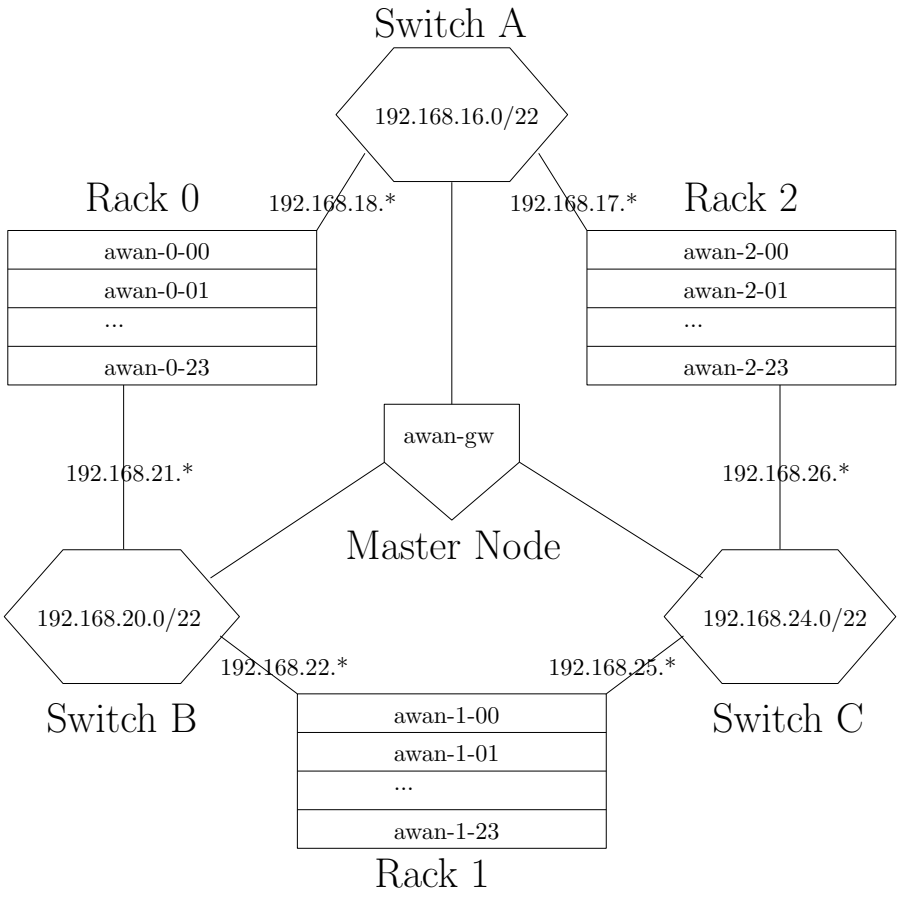


Figure 7-1: Architecture of the in-house cluster for experiments.

The architecture of the cluster is illustrated in Figure 7-1, which basically belongs to the category of flat neighborhood networks. The cluster contains a single master node and 72 slave nodes, which are connected via three switches. The master node is mainly responsible for administration services such as gateway, network file system (NFS) server and dynamic host configuration protocol (DHCP) server. The slave nodes

[1] http://www.comp.nus.edu.sg/~epiC/

are evenly divided into three racks and are used to accommodate our data storage system. Table 7.1 summarizes the hardware and software configuration of the cluster.

Table 7.1: The hardware and software configuration of the cluster

	Master Node (awan-gw)	Slave Node (awan-x-xx)
#-of-CPU	2	1
CPU	E5620 4(8) @ 2.4GHZ	X3430 4(4) @ 2.4GHZ
Memory	48 GB	8 GB
Hard Disk	2x 146 GB SAS 15k rpm 2x 500 GB SAS 7.2k rpm	2x 500 GB SATA 7.2k rpm
Network Interface	Gigabyte Ethernet	Gigabyte Ethernet
Operation System	CentOS 5.5	CentOS 5.5

7.1.2 Commercial and Distributed Clouds

To demonstrate the use of `ecStore` in broader environments, we also deploy the system on the commercial **public cloud** Amazon's EC2 [2]. Each storage node in our system runs on a small instance of EC2. This instance is a virtual machine equipped with a 1.7 GHz Xeon processor, 1.7 GB memory and 160 GB disk capacity.

Part of our research deals with the consistency issue of replicated data and load balancing problem in partitioned systems, which is also applicable to storage nodes that are located across the wide-area network (WAN) as in **distributed cloud** [26]. Therefore, we also deploy `ecStore` and conduct experiments on PlanetLab [17], a widely accepted testbed for distributed systems on WAN.

7.2 Evaluation of Generalized Distributed Indexing

In this section, we evaluate the robustness, efficiency and scalability of the generalized indexing framework developed in `ecStore`. First, we study the query performance with two query plans: *index covering* approach where the index entries contain a portion of the data records to service the query request directly and *index+base* approach where the index entries only contain pointers to the records in the base table.

Second, we compare the performance of distributed indexes against parallel full table scans. Third, we study the scalability of the system in terms of both the system size and the number of indexes. Additionally, we also present experimental results on the effect of varying data size, the effect of varying query rate, the index update performance, the ability of handling skewed data and query distribution, and the performance range join query.

7.2.1 Experimental Setup

We tested the indexing framework in two environments. We ran experiments on a set of 64 nodes in an in-house cluster to stress test the system with varying query and update rates. We also conducted experiments on a cluster of commodity machines on Amazon EC2 [2] to test the robustness and scalability of the indexing framework when varying the system size from 16 to 256 nodes. Details of the experimental environments have been presented in Section 7.1.

We run most of experiments with TPC-W benchmark dataset [20]. TPC-W benchmark models the workload of a database application where OLTP queries are common. These queries are relatively simple and selective in nature, and thus building indexes to facilitate query processing is essential, especially in a large scale environment such as the cloud. In particular, we generate 10 million to 160 million records of *item* table. Each record has an average size of 1KB. Hence, the total data size ranges from 10 GB to 160 GB. The data records are stored and sorted by their primary key, i.e., the *item_id* attribute.

Distributed indexes are employed to improve the performance of processing queries whose predicates do not contain the primary key. More specifically, we build distributed indexes on the *item_title* attribute and the *item_cost* attribute by instantiating a distributed hash index and distributed B^+-tree-like index respectively, based on the generalized distributed indexing framework developed in `ecStore`. We evaluate the performance of these distributed indexes when they are used for processing two types

of queries, namely **exact match query** with a predicate on the *item_title* attribute:

```
Q1:   SELECT item_id, item_cost
      FROM item
      WHERE item_title ='ξ'
```

and **range query** with a predicate on the *item_cost* attribute:

```
Q2:   SELECT item_id, item_title
      FROM item
      WHERE item_cost > α AND item_cost < β
```

Note that α and β are configurable and in the experiments we vary the values of α and β to define the selectivity of the test queries.

In addition, to test the system with a bigger number of indexes, we also synthetically generated data and indexes as follows. There are multiple tables T_i with schema $T_i(a_1, a_2, a_3, a_4, p)$ where each attribute a_i takes integer values that are randomly generated from the domain of 10^9 values, and attribute p is a payload of 1 KB data. Each table is generated with 10 million records. For each table T_i, the attribute a_1 is indexed with a distributed hash index, a_2 is indexed with a distributed B^+-tree-like index, and (a_3, a_4) is indexed with a distributed multi-dimensional index. Thus, each table T_i has 3 indexes, and we can test the effect of varying number of indexes in the system by increasing the number of testing tables.

Table 7.2: Experiment settings for evaluating indexes

Parameter	Default	Min	Max
System size	64	16	256
Data size	10 GB	10 GB	160 GB
Buffer size for local indexes	64 MB	-	-
# client threads per node	10	5	50
# operations per thread	1000	-	-
Query type	Exact match	-	-
Query plan	Index covering	-	-

Table 7.2 summarizes the default experiment configuration. The default system size for the experiments is 64 index nodes. The memory buffer for the local indexes at each

node is set to 64 MB. The system uses the adaptive connection management strategy as the default setting. We test the system with the default 10 client threads at each node. Each client thread continually submits a workload of 1000 operations to the system. A completed operation will be immediately followed up by another operation. We also vary the query rate and update rate submitted into the system by changing the number of client threads at each node.

7.2.2 Index covering vs. Index+base Approach

In this experiment, we study the query processing performance using distributed indexes with two alternative query plans, namely *index covering* and *index+base*. In the former approach, the index entries include the data of non-key attributes to service the queries directly. For example, if the index entries of the distributed index on *item_cost* contain the information of *item_id* and *item_title*, then the above query Q2 can be processed efficiently with only index traversal. On the contrary, in the latter case, the query processing needs to follow the pointers in the index entries and then perform random reads on the base table, which add more network round trips to the data retrieval process and increase the query response time.

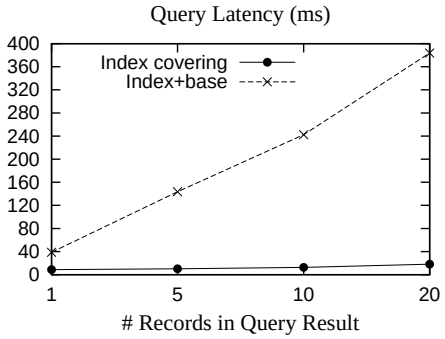


Figure 7-2: Performance: index covering vs. index+base.

Figure 7-2 shows that the *index covering* approach outperforms the *index+base* approach, especially when the size of the query result set is large. Note that even though the *index+base* has worse performance in this case, it still performs better than

scanning the entire table to retrieve only a few qualified data records. In our experiment, even a parallel scan using Hadoop MapReduce [14] on the 10 GB *item* table takes about 23 seconds (see Figure 7-4), which is significantly slower than the *index+base* approach. Moreover, it is also notable that the response time of both *index covering* and *index+base* approach depends only on the size of the query result set, while the response time of full (parallel) table scan increases with the table size, as will be shown in Figure 7-4 in the below section.

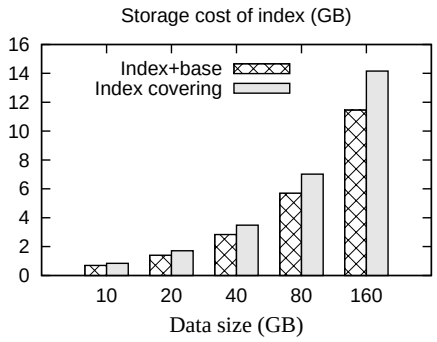


Figure 7-3: Storage cost: index covering vs. index+base.

The benefits of the *index covering* approach do not come for free since it has to spend more storage for replicating appropriate portion of the base records. Keeping the index data consistent with the base data also introduces other overhead (see Section 7.4.2 for more results on this overhead). Here, we make a comparison on the storage cost of the *index covering* and *index+base* approaches, as depicted in Figure 7-3. In the *index+base* approach, index entries only contain pointers referring to the base records. This pointer consumes low storage overhead (about 8 bytes in total including the file number and record size of short type, i.e., 2 bytes each, and the offset of the record in the file of integer type, i.e., 4 bytes). On the contrary, in the *index covering* approach, each index entry replicates the content of the *item_id* and *item_title* attributes, which sum up to about 20 to 30 bytes. Therefore, the *index covering* approach consumes more storage overhead compared to the *index+base* approach, but it is acceptable in this case since the difference in storage cost is not large while the *index covering* approach is superior in query response time.

In the extreme case of the *index covering* approach, users can opt to include all the data of the original records in the index entries to speed up query processing. Although this approach incurs additional storage cost, the query performance using the secondary indexes is improved considerably. Moreover, the additional storage cost is acceptable in the case the sizes of data records are relatively small or we only include a portion of a data record that is needed for common queries. Hence, we mainly test the performance of the system with the *index covering* query plan in other experiments.

7.2.3 Index Plan vs. Full Table Parallel Scan

In this test, we vary the data set size from 10 GB to 160 GB (from 10 million to 160 million records). We compare the performance of the system to process the query Q2 using the distributed B^+-tree-like index with the approach that performs a full parallel scan on the *item* table using Hadoop MapReduce [14] in a system of 64 cluster nodes.

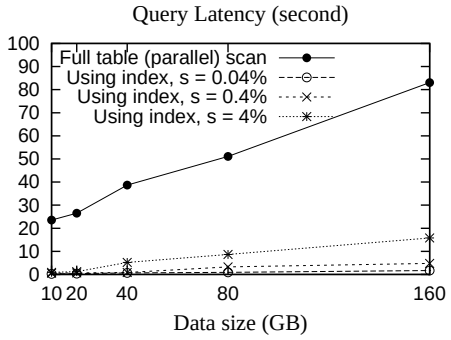


Figure 7-4: Index plan vs. full table scan.

As shown in Figure 7-4, when the data set size increases, the query latency of the distributed index approach also increases due to the increasing size of the result set. However, it still performs much better than the full table scan approach, whose query response time increases almost linearly along with the data set size.

The distributed index achieves better performance because it can directly identify the qualified records and retrieve them from the local indexes of the index nodes, while in the other approach the whole table is scanned. The parallel scans with MapReduce

still consume a significant execution time when the table size is large. Note that the distributed index also employs parallelism to speed up query processing. When the query selectivity is set to 4% or higher, the data that qualify the query could be stored on multiple index nodes. These index nodes would perform the index scan in parallel and the query latency would not increase with the size of result set any longer.

7.2.4 Multiple Indexes of Different Types

In this test, we study the query performance of the proposed indexing framework when the number of indexes in the system varies. Specifically, we experiment with 8 tables, each of which has 3 distributed indexes of the 3 different index types as described in the experimental setup. The workload, i.e., the number of client threads, submitted to the system is increased with the number of indexes in the system (1 client thread for each index). We expect that the more indexes exists in the system the bigger workload from the users can be handled. Figure 7-5 and Figure 7-6 plot the effect of varying the number of indexes on the query latency and system throughput respectively.

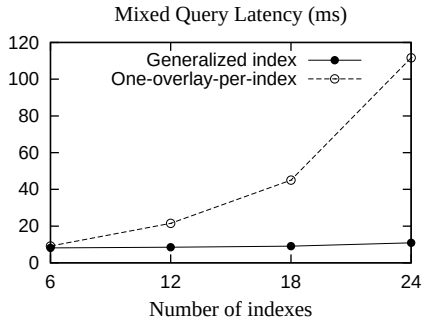


Figure 7-5: Query latency with multiple indexes.

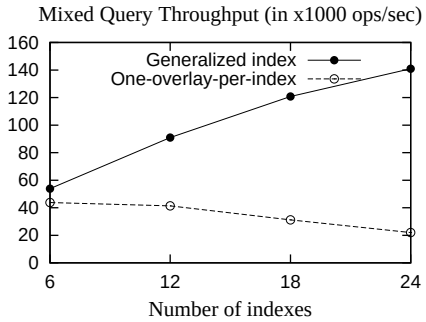


Figure 7-6: Query throughput with multiple indexes.

The results confirm the superiority of our *generalized index* over the *one-overlay-per-index* approach that runs one overlay for a specific index, i.e., 24 overlays totally for 24 indexes in this test. The *generalized index* can guarantee a constant query latency and its query throughput increase steadily with the number of indexes that it instantiates. This is due to the fact that with more indexes, there will be more index traversal paths

that can be used to answer the queries. In addition, the query execution load is better shared among the index nodes in the cluster.

While the *generalized index* approach only runs one index process to maintain multiple indexes and self-tunes the performance among these indexes via sharing resources such as memory and network connections, the *one-overlay-per-index* approach runs multiple processes, each for a specific overlay. When there are more indexes, the more processes need to be launched, which considerably adds overhead to the virtual machine and affects the query latency and throughput. Therefore, our approach provides the much needed scalability for supporting multiple indexes of different types in the cloud.

Note that increasing the number of indexes and the query rate does not help to increase the system throughput forever. When the system reaches its threshold, the query throughput will be stable even if we build more indexes. In this case, we can only improve the query throughput by adding more resources (i.e., adding more index nodes) into the system.

7.2.5 Scalability

In this experiment, we evaluate the scalability of the proposed indexing service in terms of the system size. In particular, we vary the system size from 16 to 256 virtual machines on the commercial public cloud Amazon's EC2 [2] and measure the latency and throughput of queries with different selectivities.

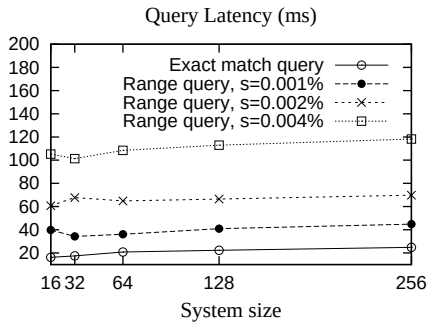


Figure 7-7: Scalability test on query latency.

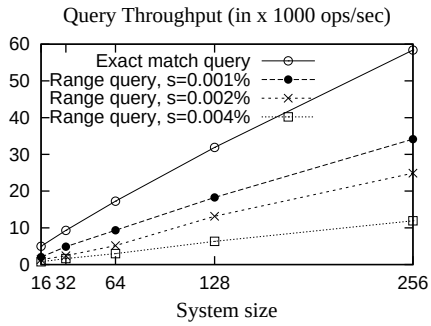


Figure 7-8: Scalability test on query throughput.

As can be seen from Figure 7-7, the system scales well with nearly flat query latency when the system size increases. In our experimental setting, the workload submitted to the system is proportional to the system size. However, more workload can be handled by adding more index nodes to the system. Therefore, the system response time for a query request with respect to a specific query selectivity is maintained nearly unchanged with different number of index nodes. Figure 7-7 also shows that a query with lower selectivity incurs higher latency, due to the larger result set, local processing costs, and higher communication cost.

As the number of index nodes increases, the aggregate query throughput also increases as shown Figure 7-8. In addition, the system query throughput scales almost linearly when the query has high selectivity, especially for the exact-match query. With a high query selectivity, the result set is small and therefore, the local processing at each index node and data transfer have less effect on the query throughput. More importantly, the indexing service achieves better throughput when there are more high selectivity queries for its ability of being able to identify the qualified data quickly rather than scanning multiple storage nodes.

7.2.6 Effect of Varying Data Size

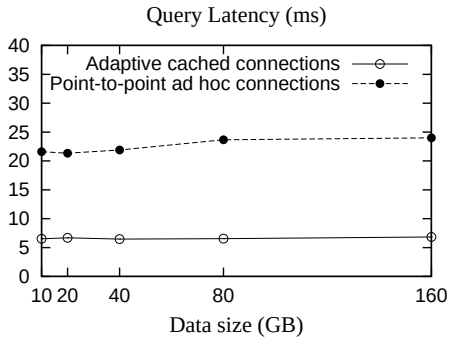


Figure 7-9: Effect of varying data size.

In this experiment, we perform the scalability test on data size and study the query performance in the system. Specifically, we measure the latency of exact-match queries

on the *item_title* attribute, a non-key attribute of the *item* table. By instantiating a distributed hash index on this attribute based on the proposed indexing framework, the system can support queries on this attribute efficiently without the need of full table scan. As shown in Figure 7-9, the query response time using distributed indexes is not affected by the database size.

In addition, the result also confirms the advantage of the proposed adaptive connection management for maintaining distributed overlays. The system can guarantee low query latency for users with the use of adaptive cached connections. In this approach, each index node in the cluster keeps a limited number of established connections to other frequently accessed nodes in the distributed index overlay. In this way, we do not need to pay the cost of creating new connections which is the norm in the case of ad-hoc point-to-point connection approach.

7.2.7 Effect of Varying Query Rate

In this test, we study the performance of exact match queries and range queries using the distributed indexes when we vary the query selectivity and the query input rate.

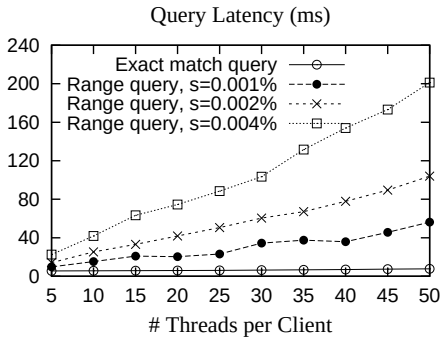


Figure 7-10: Effect of varying query rate.

Figure 7-10 demonstrates that the system has better query latency with higher query selectivity. This is due to the fact that with high query selectivity, the result set is small and the system does not need to spend much time to scan the local disk-resident index at the index nodes in the cluster and retrieve the qualified records. The results also show

that the latency of exact-match queries is less affected by input query rate than that of range queries. As discussed above, range queries incur more local disk scans than an exact-match query. When the input query rate is high, more queries will compete with each other for the disk I/Os. Thus, the latency of range queries increases with the query input rate.

More importantly, the advantage of our proposal is well demonstrated in Figure 7-11. The system achieves better load when there are more concurrent exact match queries. With the use of indexes, the system can facilitate better load distribution since it does not have to scan all nodes just to get the exact match tuples. Furthermore, due to the ability of being able to identify the storage node that contains the qualified tuple quickly, we only search that node, and search it efficiently with the support of local indexes on that node. Therefore, the system can admit more queries and the throughput increases linearly along with the input load.

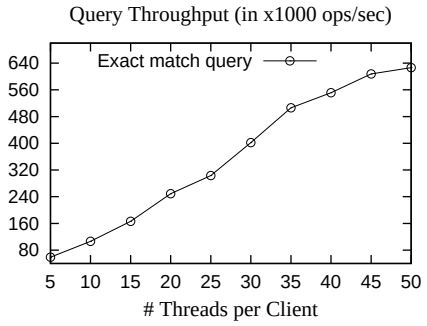


Figure 7-11: Exact-match query throughput.

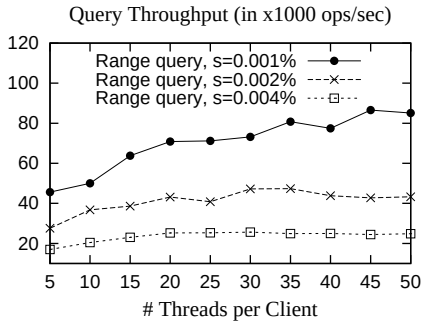


Figure 7-12: Range query throughput.

The system is also able to serve better load when there are more high selective range queries, e.g., 0.001%, as shown in Figure 7-12. However, range queries with lower selectivities incur more local disk scans, and thus the throughput of range queries is more constrained by the input load. Specifically, for range queries with selectivities 0.002% and 0.004%, when the input load reaches the threshold of 15 client threads per node, more queries will compete for disk resources and the system throughput does not increase with the input load any longer.

7.2.8 Index Update

As discussed in Section 5.3.6 (cf. Chapter 5), to process an update request the system might need to perform two rounds of index traversal since the old and the new index entry might reside on different machines, which increases the network cost. In addition, update operations might also need to modify the local disk-resident index pages. Thus, an update operation in the indexes is much costlier than a search operation during query processing. Another source of latency cost of update operations is the concurrency control on the local indexes. In summary, the latency cost of an update to a distributed index in our framework consists of three factors: the network cost, the local index update cost, and the concurrency cost.

Regarding to the three types of distributed indexes implemented in our framework, namely distributed hash, B^+-tree, and R-tree indexes, the network cost and concurrency cost of update operations are similar. The only difference is the local index update cost, which is dependent on the type of the local index, e.g., the local hash table, B^+-tree and R-tree implemented for their corresponding distributed indexes. Therefore, we shall present here the update performance of the distributed hash index. We get similar observations on the update performance of the distributed B^+-tree and R-tree index.

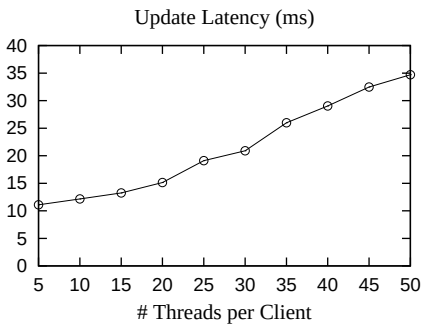


Figure 7-13: Index update response time.

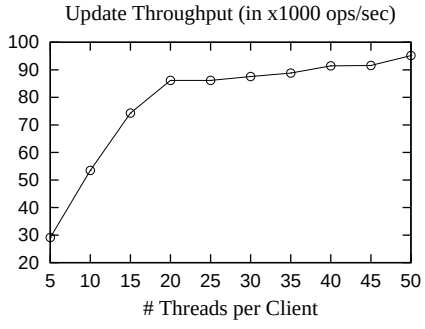


Figure 7-14: Index update throughput.

In this experiment, we test the update performance of the distributed hash index built on the *item_title* attribute of the *item* table in the 10 GB TPC-W benchmark data set. We perform scalability test with different system sizes on an in-house cluster and

measure the performance of update operations in term of the update latency and update throughput. For each system size, we also vary the input update load. Specifically, we launch from 5 to 50 client threads at each node and each client thread continuously submits update requests to the distributed index. Each completed operation will be followed up by another request.

Figure 7-13 and Figure 7-14 plot the update performance of the distributed hash index in a system of 64 nodes. As we have discussed, the update operation suffers from three considerable factors of latency cost. That is the reason why, in this test, the system is saturated when the input update rate is high, i.e., there is a large number of client threads. In particular, the update throughput becomes stable when we increase the input rate to the level of 20 client threads per node. When the input update load gets larger, there will be more number of concurrent update operations in the system. These operations compete with each other for the resources such as disk I/Os and concurrency lock holding. Therefore, given a fixed amount of resources, i.e., 64 nodes in the cluster, the update performance is constrained by a threshold of input load (about 20 client threads per node as can be seen in the result).

7.2.9 Handling Skewed Multi-Dimensional Data

In this test, we study the efficiency of our proposed indexing system in the presence of skews both in data and query distribution. We apply the Brinkhoff data generator [2] to generate a dataset of 10 million skewed 2d (two dimensional) moving objects based on the city map, which represents the real-time traffic.

System storage load distribution. The storage load of an index node is measured by the amount of index data maintained by that index node. Since the data has skewed distribution, the use of uniform data mapping function will assign some index nodes with much more data than other nodes, leading to an unbalanced system storage load distribution. This is the case where our proposed sampling-based data mapping takes

[2] http://www.fh-oow.de/institute/iapg/personen/brinkhoff/generator

its effect. As can be seen from Figure 7-15, when the sampling-based data mapping function is used, the system storage load is well distributed, i.e., a certain percentage of the number of index nodes (in the system of 64 nodes) services the corresponding percentage of the index data.

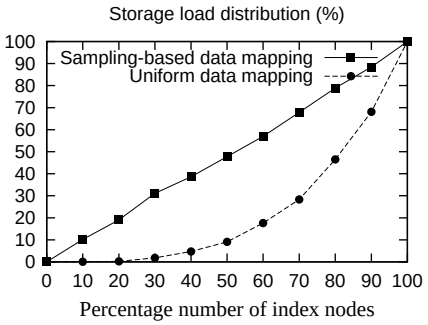


Figure 7-15: Distribution of load under skewed data distribution.

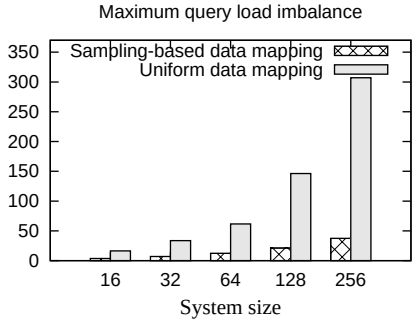


Figure 7-16: Load imbalance under skewed query distribution.

System execution load imbalance. The maximum query load imbalance is defined as the ratio between the query execution load of the heaviest-loaded node divided by the query execution load of the lightest loaded node in the system. Figure 7-16 shows the maximum query load imbalance of different system sizes under the skewed query distribution (Zipf factor = 1).

Recall that in our experimental setup, a larger number of nodes in the system results in a higher query workload input. The situation becomes even worse when the query distribution is skewed because an increasing number of queries will be directed to some hot data. If the uniform data mapping function is used while the data stored in the system have skewed distribution, the system will end up with high imbalance in the query execution load.

On the contrary, the sampling-based data mapping function proposed in our indexing framework can roughly estimate the data distribution and distribute the data over the index nodes. Thus, the incoming queries on the skewed data are also distributed over the index nodes, leading to less query load imbalance in the system.

7.2.10 Range Join Query

In this test, we synthetically generate data to evaluate the performance of indexes for processing range join query. In particular, the dataset includes two tables T_1 and T_2 with the same schema (rid, val, p) where *rid* is the record id, *val* takes its values from the domain of 10^9 values and p is a payload of 1KB. These two tables are stored with *rid* as the primary key and we instantiate secondary indexes for both tables on the *val* attribute using the distributed B$^+$-tree of the proposed indexing framework.

We measure the latency of the following range join query on the *val* attribute:

```
SELECT  T1.rid, T2.rid
FROM    T1, T2
WHERE   T1.val between (T2.val + α) and (T2.val + β)
```

We define the join selectivity of the test queries, i.e., the average number of the joining T_2 records per T_1 record, by setting the values of α and β. Note that the joined columns (the *val* column of table T_i in this experiment) typically share the same data semantics and hence, the index data of these columns are normally partitioned and distributed over the index nodes in the same manner.

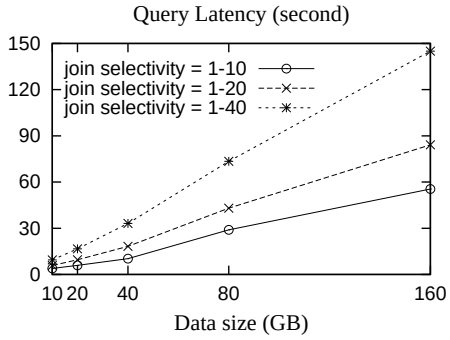


Figure 7-17: Range join performance.

Figure 7-17 plots the response time of range join queries with different selectivities and data sizes in our in-house cluster of 64 nodes. The result confirms the benefit of using distributed indexes for processing join queries. In particular, the system only takes 3.9 seconds to perform a range join query over two tables of an aggregate data size of

10 GB. Join queries can be performed efficiently with the support of indexes because the index data have already been partitioned based on the join attribute, enabling the index nodes to scan their local indexes and join the records in parallel.

7.3 Evaluation of Replication and Transaction Management

In this section, we evaluate the performance of replication and transaction management in `ecStore`. The experiments are conducted on the commercial cloud Amazon's EC2 [2]. Specifically, we study the scalability of the system in term of system throughput and response time, the advantages of load-adaptive replication, range scan query performance, the effect of self-tuning histogram, and experimental results with TPC-W benchmark [20]. In addition, we also study the performance of `ecStore` on PlanetLab environment [17].

7.3.1 Experimental Setup

Experimental data is synthetically generated based on a social application. A data record has a key, which is the user identity, and contains a string representing this user's friend list (a list of other user ids). Data are stored in a clustered table that has a primary index on the key of records. The index is instantiated as a distributed B^+-tree-like index from the generalized indexing framework developed in `ecStore`, to facilitate partitioning the user table based on users' location which is part of the user identity. A write operation will update the friend list in the record while a read operation returns this information to the users. When two users accept as friend of each other, we execute a transaction bundling four operations: two read operations to retrieve information of these two users and two write operations to update their buddy lists. The identity of a user is randomly chosen from a space of 10^9 users. The system is initially bulk loaded with $10,000 * N$

records where N is the number of storage nodes in the system. The default system size for the experiments is 18 storage nodes. Each data object is stored with replication level of 3. The threshold factor to trigger the replication process for load balancing (cf. Section 6.1.3) is set to 2. A workload of 1000 operations is continually submitted to each storage node in the system. A completed operation will be immediately followed up by another operation.

7.3.2 Scalability

In this experiment, we study the *elastic* scaling property of `ecStore` in terms of system throughput and response time when testing the system with different system sizes.

System throughput

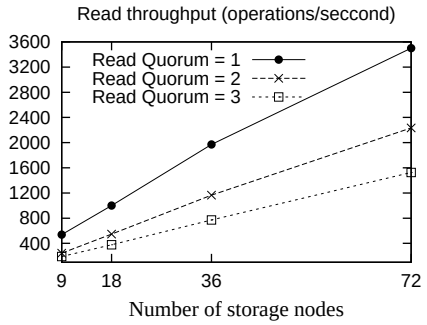


Figure 7-18: Read throughput with different consistency levels.

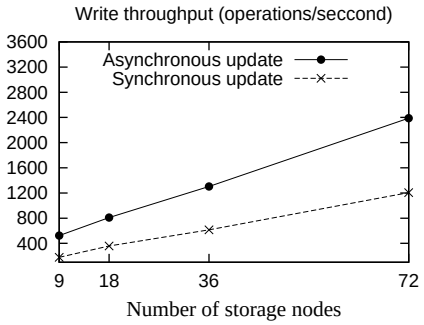


Figure 7-19: Write throughput with replication level 3.

Figure 7-18 shows the read throughput of `ecStore` with different levels of read consistency. When users require strict consistency for a read operation, the system needs to collect all replicas of a data record and return the most recent copy to the user. The trade-off of high level of read consistency is the decrease in throughput. This figure also shows that `ecStore` can scale well: as the number of storage nodes increases, the aggregate read throughput also increases. The system read throughput scales almost linearly when read consistency is relaxed (by requesting a small read quorum).

In addition, the write throughput of ecStore with replication level 3 is shown in Figure 7-19. As expected, the pessimistic replication method is outperformed by the optimistic replication technique adopted in ecStore. Consequently, with the use of optimistic replication and write-ahead logging, ecStore can provide high write throughput while still being able to guarantee data durability.

System response time

Figure 7-20 demonstrates a good elastic scaling property of ecStore, where more load can be handled by adding more storage nodes to the system. In our experiment setting, the workload submitted into the system is proportional to the system size. However, with a larger number of nodes, the system has more capacity as well. Therefore, the system response time for a read request with respect to a specific read consistency is maintained nearly unchanged with different number of storage nodes. In addition, it can also be observed from Figure 7-20 that a query which requires better read consistency, i.e., requesting a larger read quorum, suffers from higher latency.

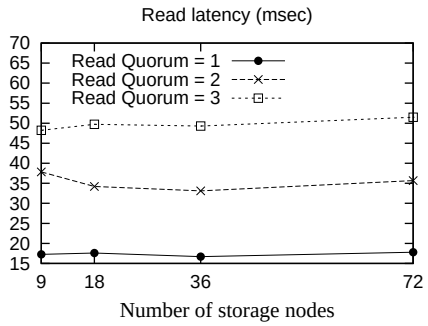


Figure 7-20: Read latency with different read consistency levels.

Transaction throughput

Figure 7-21 plots the transaction throughput of ecStore when the percentage of read-only transactions (Txn mix) varies from 10% to 90%. In this experiment, each update transaction bundles two read operations and two write operations while a read-only transaction performs only two read operations.

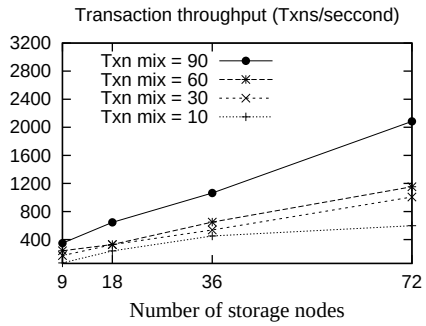


Figure 7-21: Transaction throughput with different read/write ratio.

The multiversion concurrency control scheme guarantees that read-only transactions will always commit successfully without spending time to check data conflicts with other concurrent update transactions. Hence, the transaction throughput regarding to each system size increases together with the percentage of read-only transactions in the workload. In addition, Figure 7-21 also illustrates that the transaction throughput scales well under heavier read workload (Txn mix = 60% and 90%).

7.3.3 Handling Skewed Query Distribution

In this experiment, we first study the convergence rate of the load statistics information that each storage node observes. We then examine the effect of replication on the system load distribution and maximum load imbalance when the query distribution is skewed. Zipfian factor is set to 1 in this test. We also study the effect of varying replication threshold factor and transaction restart probability under the skewed access pattern.

Load statistics convergence rate

The load-adaptive replication technique requires each storage node in the system to know the load of other nodes to facilitate its decision where to replicate the hot query ranges. As presented in Chapter 6, `ecStore` piggy-backs the load information of storage nodes on the periodical heart-beat messages sent between the storage nodes in the system.

Thus, after every few iterations of heart-beat messages, each storage node can obtain the load statistics of other nodes in the system. Figure 7-22 illustrates the quick

convergence rate of the load statistics information when we experiment `ecStore` with different system sizes.

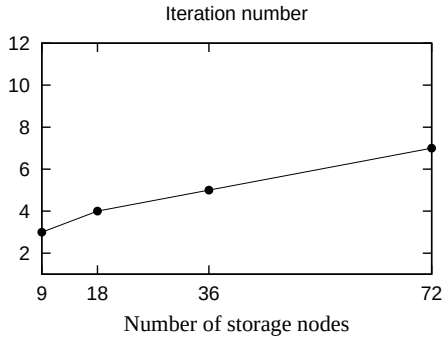


Figure 7-22: Load statistics convergence rate.

System load distribution

The load of a storage node is measured by the number of queries that have been served by this node. Ideally, a certain percentage of the number of nodes in the system is expected to serve the corresponding percentage of the total system workload. However, as can be seen from Figure 7-23, this is not the case when the system employs no replication. Under skewed query distribution, the only one copy of data will experience high workload and become the bottleneck after a short while, which leads to high imbalance in the system load distribution.

A higher replication level will balance the system load distribution since the additional replicas could help to shed the workload on the overloaded primary copy. However, the system cannot afford to replicate all data records at a high replication level due to storage cost and replica consistency maintenance cost.

This is the case where the two-tier partial replication takes its effect. As can be seen from the curve labeled '3-adapt' in Figure 7-23, when a replication level of 3 is augmented with load-adaptive replication, the system load is well distributed, even better than using replication level 4. It is because the proposed load-adaptive replication method selectively replicates more copies for the hot data ranges to shed the workload

of the overloaded node to other under-loaded nodes. In this way, we can achieve a balanced system load distribution while keeping the cost of replication minimal.

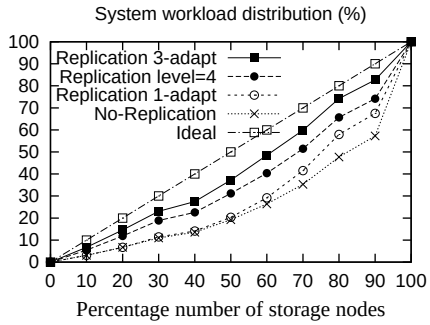


Figure 7-23: Distribution of load under skewed query distribution.

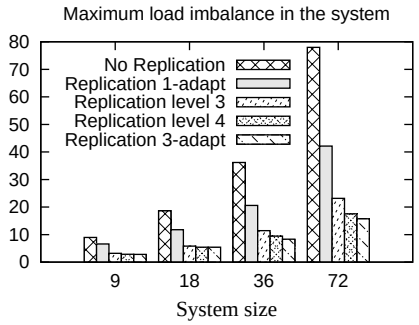


Figure 7-24: Load imbalance under skewed query distribution.

Maximum load imbalance

The maximum load imbalance is defined as the ratio between the loads of the heaviest-loaded node divided by the loads of the lightest-loaded node in the system. Figure 7-24 plots the maximum load imbalance of different system sizes under the skewed access pattern. With our experiment set up, a larger number of nodes in the system will also result in a higher query workload input. The situation becomes worse when the query distribution is skewed: an increasing number of queries will be directed to one hot spot. If no replication scheme is used, the system will end up with high load imbalance. On the contrary, the load-adaptive replication implemented in `ecStore` quickly helps to reduce more than half of the maximum workload imbalance without the need of replicating all data in the system at high replication level.

Effect of varying replication threshold

Recall that the threshold factor determines the rate at which the system can react to changes in access patterns. Figure 7-25 shows the effect of varying the threshold factor on the maximum load imbalance in a system of 18 nodes with replication level 3-adaptive.

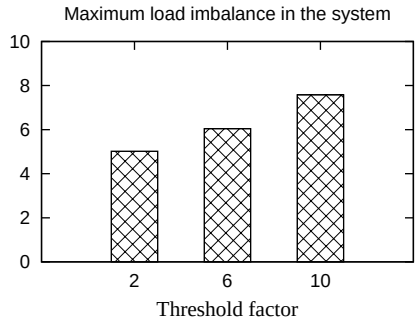


Figure 7-25: Effect of threshold factor to activate replication process.

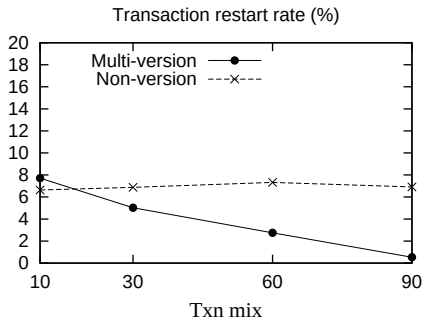


Figure 7-26: Transaction restart probability under skewed workload.

We can observe from the figure that the system has less load imbalance when this threshold is set to small values. It is because with a small threshold, a storage node can recognize its overloaded state and hot query ranges faster. Consequently, it can activate the replication process for load balancing at the right time when the system faces a flash crowd query. However, setting a small value for the threshold factor benefits the system only when the query access pattern is often skewed and changes overtime. Otherwise, constantly checking the system overloaded state and determining which data ranges to replicate could consume CPU time and affect the overall performance of the system.

Transaction restart rate

In this experiment, each transaction submits a query with range size 100 (the start value of query range is selected with Zipfian distribution), updates ten values among them and writes back to the system. This setting of large read-set and write-set together with the skewed query distribution increase the probability of transaction restart as shown in Figure 7-26.

However, we can observe the advantage of multiversion concurrency control. Since read-only transactions do not need to check data conflicts with other concurrent update transactions in the system, the transaction restart probability reduces when the transaction mix, i.e., the ratio of read transactions over total number of transactions in the system, increases. On the contrary, under non-versioning scheme, each transaction

needs to validate against other concurrent transactions at the commit time. Therefore, in the case of non-versioning scheme the transactions almost have the same restart probability with different transaction mixes.

We note that the advantage of multiversion concurrency control however introduces some overhead. In addition to the extra disk space for storing the history of data, we need to maintain a global counter in the system for version time-stamping. Furthermore, each update transaction which has successfully passed the validation phase needs to contact this global counter to get its commit-number. Thus the commit process takes longer time to complete and there is a higher probability for other transactions to conflict with it. Hence, with small transaction mix, i.e., the number of update transactions is much larger than the number of read-only transactions, the multiversion scheme suffers from a little higher transaction restart probability than the non-version scheme.

7.3.4 Varying Size of Range Scans

In this experiment, we study the impact of varying the size of range scans on the request latency. Consider a Web 2.0 photo sharing application, e.g., Flickr[3], where users upload and share photos. Examples of range scan queries in this application are: finding the photos having the top ranking by users within the last 7 days, the last month, the last 4 months, etc.

We generate a data set representing the metadata records for 9 million photos ordered by the date when photos are uploaded. The average record size is 200 bytes. We distribute the data set on 18 storage nodes where each node maintains the metadata of photos uploaded in 1000 days. Thus, the query "finding the top ranking photos within the last 4 months" requires scanning about 0.7% of the sample data set.

As shown in Figure 7-27, while sequentially scanning the range could be inefficient, we can improve the request latency by utilizing the existing replicas of the data and perform *parallel range scan*. In particular, we divide the range request into smaller

[3]http://www.flickr.com/

chunks and scan these chunks at the same time but on different replicas. Since these replicas are distributed on different nodes, the completion time for scanning the whole range is improved considerably.

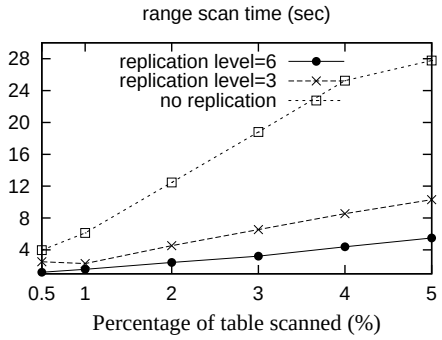


Figure 7-27: Parallel range scan performance.

7.3.5 Effect of Self-tuning Range Histogram

We now study the effect of self-tuning range histogram in handling access patterns with flash crowd queries. In the above photo sharing application, for instance, there are more queries like "finding the highly ranked photos uploaded today" where today has some special event like the eclipse happening. In this experiment, we test the system by continuously submitting 200 queries, 60% of which are the flash crowd requests, to each storage node in a system of 18 nodes. Under this access pattern, the system load distribution is highly skewed as shown in Figure 7-28 when no replication-based load balancing technique is employed. Note that the data migration technique would not help in this case because it only migrates the hot data from one node to another.

We also examine the effect of the load balancing technique with different histogram configurations: STR-10 stands for self-tuning range histogram with 10 buckets while FIR-200 and FIR-400 stands for fixed range histogram with 200 and 400 buckets respectively. Although the memory cost for maintaining FIR-200 and FIR-400 histogram is much higher than STR-10 histogram, they still could not capture the access frequency of popular queries precisely.

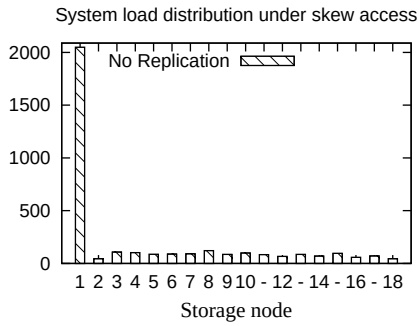


Figure 7-28: Load distribution without load balancing.

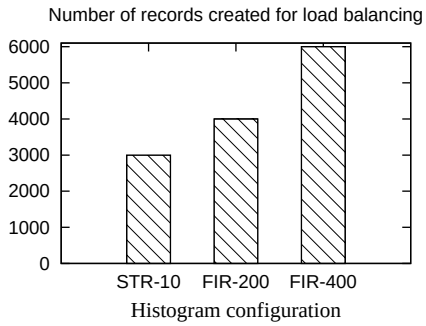


Figure 7-29: Number of created replicas.

As a result, FIR-200 and FIR-400 populated many more records during the replication process for load balancing than STR-10 as depicted in Figure 7-29. Unfortunately, many of these records are “false positive”, populated but do not really help much for load balancing purpose. Note that FIR-200 could not estimate the access frequency as accurate as FIR-400, thus it cannot afford to replicate data at high speed as FIR-400; otherwise leading to high storage cost and replica consistency maintenance cost. Therefore, FIR-200 populated less number of records than FIR-400 in the end.

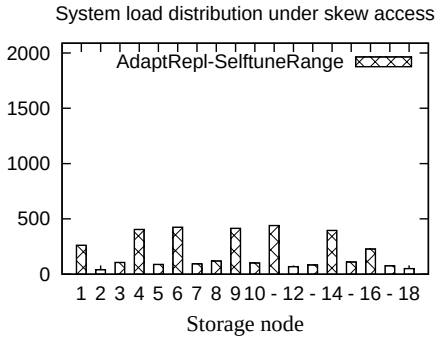


Figure 7-30: Load distribution with self-tune range replication.

On the contrary, STR-10 can capture the hot query even when its memory cost is much less than the other two histograms. Hence, STR-10 can comfortably replicate the right small number of hot queried data at high replication speed, i.e., creating more replicas each time, to quickly balance the system load. Consequently, the query execution load when using STR-10 is well distributed across the system as shown in Figure 7-30.

7.3.6 TPC-W Benchmark

We now describe the results when testing `ecStore` on EC2 with TPC-W benchmark [20], which models the on-line book store application workload. Specifically, the browsing mix, shopping mix and ordering mix have 5%, 20% and 50% update transactions, respectively. Shopping mix is the most representative workload. Since we only focus on storage system performance, we do not implement the application server or measure the web-interaction throughput and web-interaction response time.

Instead, we stress test the system by using a client thread at each storage node to continuously submit transactions to the system and then benchmark transaction throughput and response time. Read-only transactions perform one read operation to query the details of a product. We implements two kinds of update transactions: adding an item to a user's shopping cart (this transaction includes one write operation) and performing the order request (this transaction bundles one read operation to retrieve the user's shopping cart and one write operation to the orders table). Each storage node is bulk loaded with 10000 items and customers before the experiment.

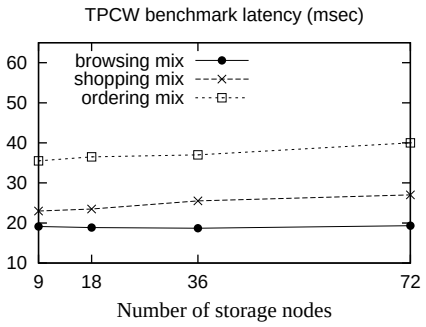


Figure 7-31: TPC-W transaction latency.

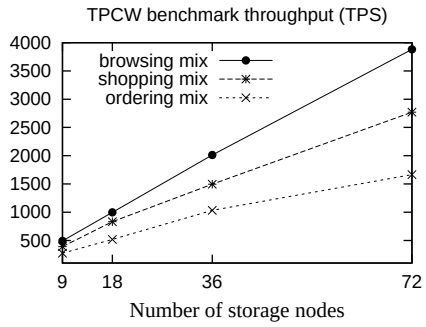


Figure 7-32: TPC-W system throughput.

Figure 7-31 illustrates that under browsing mix and shopping mix, `ecStore` scales well with nearly flat transaction latency when the system size increases. It is because of the fact that the multiversion optimistic concurrency control scheme used in `ecStore` favors read-dominant workload. As a result, the transaction throughput shown in Figure 7-32 scales linearly under these two workloads. In contrast, there is a decline in the

transaction throughput when the ratio of update transactions increases as in the case of ordering mix.

Note that the transaction throughput when we test with TPC-W benchmark is higher than that of the social application in Section 7.3.2 since the transactions in this benchmark setting bundle less number of operations than in the other experiment. Moreover, the transactions in TPC-W benchmark have more data locality when users update their own shopping carts and orders information, which are usually located on one storage node. Hence, the transactions do not spread over different storage nodes in the system.

7.3.7 Experiments on PlanetLab

Part of our research deals with the consistency issue of replicated data and load balancing problem in partitioned systems, which is also applicable to storage nodes that are located across the wide-area network (WAN) as in distributed clouds [26]. Therefore, we also deploy `ecStore` and conduct experiments on PlanetLab [17], a widely accepted testbed for distributed systems on WAN. The system size includes 18 nodes in the US region. In this experiment, the query workload is generated according to Zipfian distribution with the skew factor set to 1.

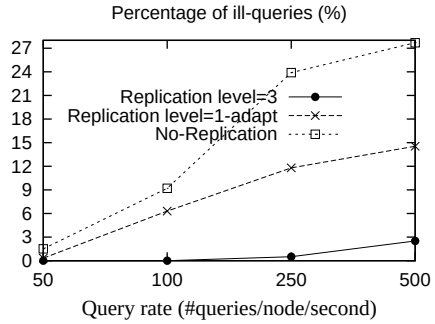


Figure 7-33: Percentage of failed-queries under skewed workload.

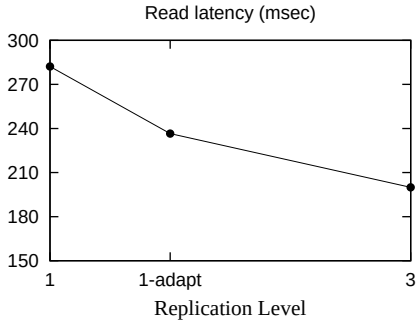


Figure 7-34: Latency of read operation under skewed workload.

Percentage of failed-queries. In this test, we measure the number of failed queries with different queries rate ranging from 50 to 500 queries submitted to each node per

second. We set the capacity of each storage node to 100 messages in the message-processing queue. This means that incoming messages will be dropped if the message processing queue is currently full with 100 messages already. A query request will fail if its messages are dropped during the query processing.

Figure 7-33 plots the percentage of failed-queries with different query rates under the skewed query distribution. It can be observed from the figure that there is a high percentage of failure queries when no replication is exploited. Especially, the system suffers from the highest percentage (up to 27%) of failure queries when there are 500 queries submitted to each node per second.

On the contrary, the system can perform well when the replication-based load balancing technique takes its effect. By maintaining a replication level of three, we can reduce the percentage of ill-queries significantly. Furthermore, the curve with key '1-adapt' in Figure 7-33 also shows that the load-adaptive replication reacts effectively to the skewed access pattern. The system starts with no replication, then gradually creates more replicas of popular data ranges to shed the skewed query execution to others node, thus reducing the percentage of failed-queries.

Improved query response time. In this test, we measure the latency of read operations in three settings: replication level 1, '1-adapt' and 3. As depicted in Figure 7-34, the workload of the skewed access pattern is dispersed to other replicas, which prevents the primary copy of a data object from becoming the bottleneck. Hence, the latency of read operations is decreased when the replication technique is employed in the system. In particular, the average query latency is significantly improved when we increase the replication level from 1 (no-replication) to 3 (there are totally three copies of data in the system). Especially, with the replication level 1 augmented with the load-adaptive technique, `ecStore` gradually populates more replicas of the hot query ranges and improves the query response time when compared to the case of no-replication.

7.4 Evaluation of Overall System

In previous sections, we have evaluated various features of `ecStore` including distributed indexes, smart replication and transactional management. Now, we further evaluate the performance of `ecStore` as a whole system with TPC-H benchmark [19]. Firstly, we examine the update throughput of `ecStore` in the presence of indexes and replication. Secondly, we analyze the performance characteristics when testing `ecStore` with various types of query including single-dimensional exact match, multi-dimensional range selection, and join queries. Thirdly, we study the data freshness that `ecStore` can provide for OLAP jobs when the data are being updated by OLTP operations simultaneously. Finally, we also compare `ecStore` with other cloud data serving systems in terms of both system features and performance of data operations with Yahoo! cloud serving benchmark (YCSB) [55].

7.4.1 Experimental Setup

Table 7.3: Default settings for evaluating overall system

Parameter	**Default**
System size	64
Buffer size for local indexes	64 MB
# client threads per node	1
# operations per thread	1000
Query plan	Index covering
Replication	3-way, asynchronous

Table 7.3 summarizes the default experiment configuration. The default system size for the experiments is 64 nodes in the in-house cluster (see Section 7.1 for hardware and software setup). The memory buffer for the local indexes at each node is set to 64 MB. The system uses 3-way replication with asynchronous update propagation (cf. Section 6.1.3), which ensures low write response time while still guaranteeing data durability, as the default setting. We test the system with a default client thread submitting workload at each node. Each client thread continually submits a workload of 1000 requests to the

system, and a completed request will be immediately followed up by another request. We also vary the query rate submitted into the system by changing the number of client threads at each node. The system employs index covering approach where the index entries contain a portion of the data records to serve the query directly without having to access the base table.

7.4.2 Update Performance

In this experiment, we test the update performance of `ecStore` in the presence of indexes as well as replication of both base data and index data. Since `ecStore` employs asynchronous replication with write-ahead logging (cf. Section 6.1.3) to ensures low write response time while still guaranteeing data durability, the main overhead of update operations comes from writing new version of base data and index data.

The system is initially bulk loaded with the TPC-H dataset at scale factor 20, which results in about 20 GB dataset. We test the system with varying system size ranging from 8 to 64 nodes. To facilitate queries with predicates on *totalprice*, a secondary attribute of the *Orders* table, we build a distributed B^+-tree-like index on this attribute by instantiating the corresponding type of index from the generalized indexing framework developed in `ecStore`.

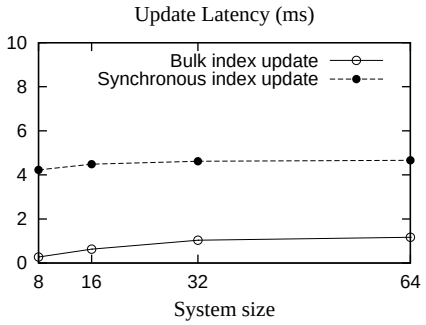


Figure 7-35: Update latency.

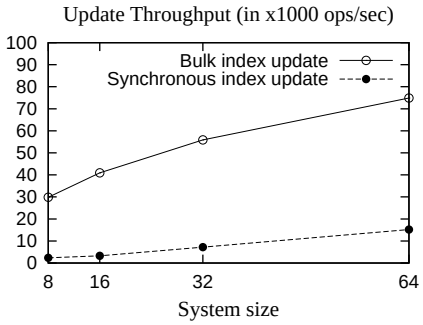


Figure 7-36: Update throughput.

After the data bulk loading phase, we execute an update workload from the TPC-H benchmark that generates and inserts new order records into the *Orders* table, which will also trigger the update of indexes accordingly. As we have presented in Section 5.3.6,

ecStore provides two options of index update, namely synchronous update and bulk (asynchronous) update. Figure 7-35 and Figure 7-36 show the performance trade-off of these two options.

In the former option, the indexes are updated under strict enforcement of ACID properties, i.e., ecStore needs to reflect all modifications on the base records to the associated indexes before returning acknowledgement messages to users. Since the base data and index data, are possibly located on different machines, a distributed consensus protocol is needed, and therefore affects the update performance.

On the contrary, in the latter option, ecStore backlogs the modification of base records and performs bulk update to the associated indexes when the system has spare I/O and network bandwidth. Consequently, the performance of client update workload is significantly less affected by the index maintenance process. It is noteworthy that the enforcement of consistency and ACID properties for index update is configurable and determined by users based on the requirements of applications.

It can also be observed that for each option, the update latency slightly increases as the system grow in size. This is due to the fact that in our experimental setting a bigger number of nodes in the system results in a bigger concurrent update workload, which means more update operations will compete with each other for I/O and network resources and therefore affect the update latency. However, this slightly increase in the latency does not have much impact on the aggregate throughput of the system. The update throughput scales almost linearly when more resources are added to the system.

7.4.3 Query Performance

In the following, we examine the query performance of ecStore with TPC-H benchmark [19], particularly on single dimensional, multi-dimensional and join queries.

Simple Select Query

In the previous experiment, we have initially bulk loaded a TPC-H dataset with scale 20, and created a distributed B^+-tree-like index on the *totalprice* attribute of the *Orders* table. Now, we show the performance of a simple select query (Q3) that has an exact match predicate on this attribute.

```
Q3:   SELECT custkey, orderkey, orderdate
      FROM Orders WHERE totalprice = x
```

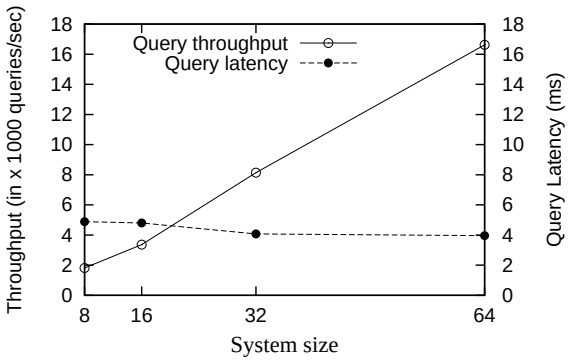


Figure 7-37: Performance of query with single-dimensional predicate.

The response time and throughput of Q3 with varying system sizes are plotted in Figure 7-37. It can be seen that the query response time declines as the system grows in size, which confirms that with more added resources, the system is able to speed up query processing. Further, with the support of the distributed index, the system is capable of locating the storage nodes that maintains the records of interest quickly and retrieving the data from these nodes efficiently. Therefore, the results also show that the system achieves almost linear throughput with this Q3.

Multi-dimensional Query

We now measure the query throughput of Q4 with varying system sizes to show the scalability of the system when executing queries contains range selection predicates on

multiple attributes. Since the selection predicate of the query does not contain the primary key attributes that are used to partition the base table, ecStore employs distributed indexes to improve the performance of query processing. Specifically, we build a distributed R-tree-like index on two attributes *(totalprice, orderdate)* of the *Orders* table, by instantiating the corresponding type of index from the generalized indexing framework developed in ecStore.

```
Q4:  SELECT custkey, count(orderkey), sum(totalprice)
     FROM Orders
     WHERE totalprice ≥ y and totalprice ≤ y + 100 and
        orderdate ≥ z and orderdate ≤ z + 1 month
     GROUP BY (custkey)
```

Similar to the previous experiment, the TPC-H dataset is generated at scale factor 20. We test the system with varying system size ranging from 8 to 64 storage nodes in the in-house cluster (cf. Section 7.1). We populate multiple client threads at each node (up to 10 client threads per node) to continuously submit queries into the system. The values of parameters in Q4 are generated following uniform distribution.

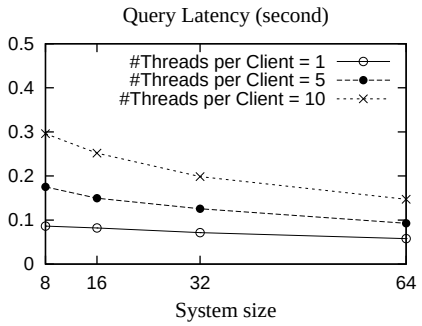


Figure 7-38: Response time of multi-dimensional query.

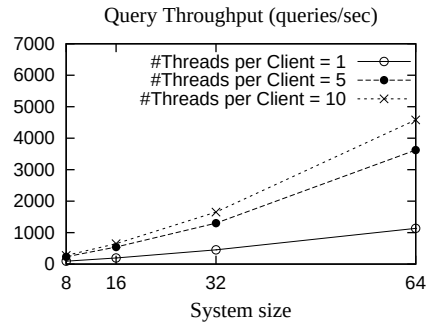


Figure 7-39: Throughput of multi-dimensional query.

The query response time and throughput of Q4 with varying system sizes and number of client threads per node are plotted in Figure 7-38 and Figure 7-39 respectively. The result confirms the benefit of distributed indexes in ecStore in

improving the performance of query processing. In particular, the system achieves almost linear throughput with respect to the increasing number of storage nodes. It is noteworthy that, in our experiment setting, the more storage nodes in the systems, the more query requests will be populated. However, with the support of the distributed index, the system can handle these queries efficiently for its ability to identify the storage nodes containing the data of interest quickly and distribute the workload among storage nodes uniformly. The system is therefore able to provide elastic scaling property where extra workload can be handled by adding more nodes into the system.

Join Query

As presented in Section 5.3.5, distributed indexes in `ecStore` can facilitate parallel joins and speed up the performance of join query processing. The joined columns of different tables typically share the same data domain and hence, the index data of these columns are partitioned and distributed over the index nodes in the same manner. As a consequence, these index nodes are able to scan their local indexes and join the records in parallel.

In this experiment, we compare the join query performance of two approaches, namely index join in `ecStore` and sort merge join in Hadoop MapReduce [14]. The MapReduce-based sort merge join (provided as a contribution code package in the Hadoop open source) is implemented as follows. In the map phase, the mappers scan through the two joining tables and with each tuple t the mappers generate an intermediate records $(k_t; t')$ where k_t is the joining key and t' is tuple t tagged with the name of the table that it belongs to. The mappers then partition these intermediate records based on the value of the joining key (a hash function is normally used to guarantee load balance) and shuffle them to the reducers. To deal with large-scale data, the reducers have to write these intermediate data into disks instead of buffering them in memory due to the scale of data. In the reduce phase, the reducers just need to form groups of the same key value and yield the final join results.

To compare the performance of the two approaches, we measure the response time of the join query Q5 based on the TPC-H schema [19].

```
Q5:   SELECT  O.orderkey, orderdate, L.partkey, quantity, shipdate
      FROM    Orders O, Lineitem L
      WHERE   O.orderkey = L.orderkey
            and O.orderpriority='1-URGENT'
```

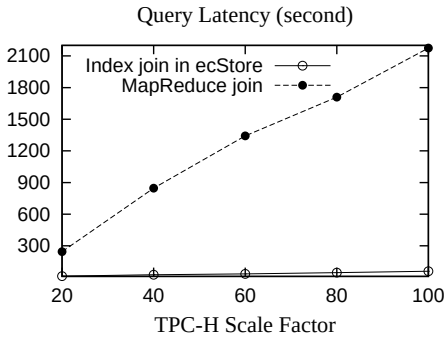


Figure 7-40: Index join vs. MapReduce join.

Figure 7-40 plots the response time of Q5 when we vary the scales of TPC-H dataset in the in-house cluster of 64 nodes. The results confirm the benefit of distributed indexes in ecStore for join query processing. As expected, the index join approach outperforms the MapReduce join approach because the indexes are able to identify the joinable records quickly and join them in parallel on each index node. On the contrary, the main shortcoming of the MapReduce join approach is the need to transfer the entire base tables from the mappers to the reducers, which incurs a great deal of overhead including network bandwidth and disk I/O for writing intermediate data.

7.4.4 Data Freshness

In this experiment, we measure the data freshness that ecStore can provide for OLAP queries which typically scan the whole data in the tables. In particular, when an ad-hoc OLAP job is running on a dataset which is being simultaneously manipulated by OLTP

operations, we measure how old the version of the dataset read by the OLAP job is, compared to the latest version of the dataset.

We deploy the system on a set of 64 nodes and bulk load different sizes of data (32 GB to 512 GB). In the experiment, each record maintains a maximum of 8 versions. We employ another set of 5 cluster nodes to submit updates to the system continuously at the rate of 100 operations/sec. The updates follow either uniform distribution or normal distribution, denoted as U and N respectively in the result graphs.

Two metrics are used in the benchmark. In a scan operator starting at t_0, when reading record r, `ecStore` retrieves the ith version, whose timestamp is t_1. Note that $t_1 \leq t_0$ and r does not have any other version between t_1 and t_0. After the scan operator completes, the latest version of r is j and the timestamp is t_2. The version difference regarding to r is $j - i$ and the time delay is $t_2 - t_1$. For comparison purpose, we examine another scan approach, which always retrieves the latest version of the records. That is, when it reads r, it retrieves r's current latest version. We refer to these two scan approaches as *ecstore* and *recent* respectively.

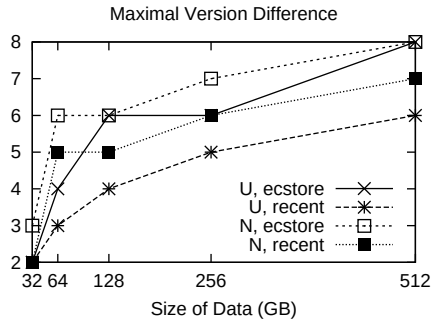


Figure 7-41: Maximal version difference.

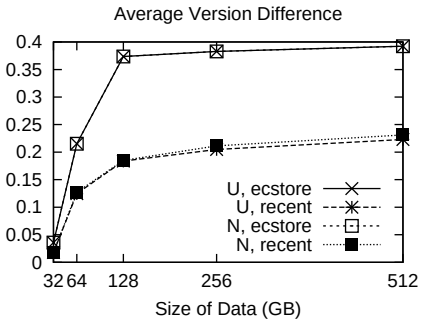


Figure 7-42: Average version difference.

Figure 7-41 and Figure 7-42 show the maximal and average version difference among all records, respectively. When the data size increases, it incurs more overhead to scan the dataset. Hence, both *ecstore* and *recent* approaches suffer from stale versions observed by scanning jobs. However, scanning 512 GB dataset just leads to a maximal of 8 version difference. Further, *recent* only provides a slightly "fresher"

result than *ecstore*, even it always attempts to read the latest version. This is because of the fact that in `ecStore`, multiple storage nodes start the scanning the data in parallel, which is quite efficient and hence reduces the affect of concurrently executing update operations. It can also be seen that for each approach, the update pattern does not have much impact on the data "freshness" observed by scanning jobs. Uniform updates generate a similar result (data freshness) to the normally distributed ones. For such a large dataset, a specific record will not receive too much updates, even in a skewed distribution.

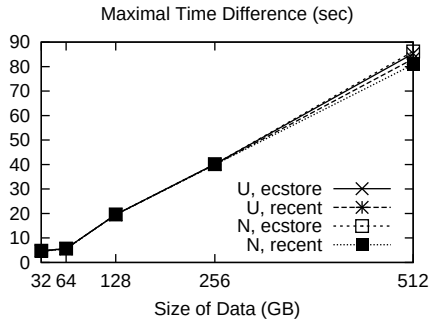


Figure 7-43: Maximal time delay.

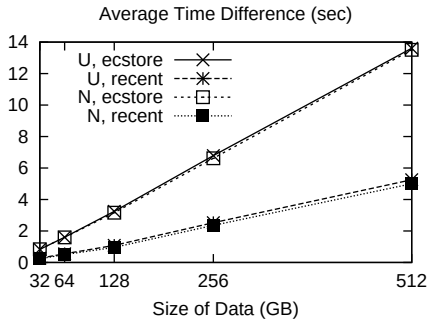


Figure 7-44: Average time delay.

Figure 7-43 and Figure 7-44 plot the maximal and average time delay. We get a similar result as in the version difference metrics. Scanning 512 GB dataset only incurs a maximal delay of 90 seconds. In most cases, such delay is acceptable since users commonly do not mind to get a global statistics which provides a view for the system of 90 seconds ago. The above results show that `ecStore` can provide for most OLAP scanning jobs a fresh and consistent snapshot of the data which are simultaneously manipulated by OLTP operations.

7.4.5 Comparison with Other Systems

Feature comparison

Compared to other closed-source cloud data serving systems (such as BigTable [49], Megastore [37], Dynamo [61] and Pnuts [54]) and open-source systems (such as HDFS [7], HBase [6] and Cassandra [93]), `ecStore` provides three additional important

features for supporting database applications in the cloud, including load-adaptive replication, efficient distributed indexes and transactional semantics across multiple keys. We summarize the feature comparison of `ecStore` with other cloud data serving systems in Table 7.4.

Table 7.4: Feature comparison of `ecStore` with other cloud data serving systems

	Partitioning		**Load balancing**	**Replication**		**Distributed transaction**	**Distributed indexes**
	H/R	Routing		Sync/Async	Consistency		
HDFS [7]	Other	Master-slave	N/A	Sync	N/A	N/A	N/A
Pnuts [54]	H+R	Router	Data migration	Async	Timeline + eventual	N	Materialized view
BigTable [49] (and HBase [6])	R	Master-slave	Add tablet server	Sync	Multi-version	N	N
Megastore [38]	R	Master-slave	Add tablet server	Sync	ACID	Y (cross partition)	Lack multi-dimensional
Dynamo [61]	H	P2P	Multiple virtual nodes on a machine	Async	Eventual	N	N
Cassandra [93]	H+R	P2P	Move node on the ring	Sync+ Async	Multi-version + eventual	N	Hash
ElasTraS [57]	R	Master-slave	Add tablet server	Sync	Multi-version	Y (within partition)	N
`ecStore`	H+R +M	P2P	Data migration + **load-adaptive replication**	Async	Timeline + eventual	**Y (cross partition)**	**Generalized distributed indexing framework**

Notation:
- H: Hash, R: Range, M: Multi-dimensional
- P2P: peer-to-peer
- Sync: Synchronous, Async: Asynchronous
- Y: Yes, N: No, N/A: Not applicable

It is noteworthy that these systems have been designed and implemented to achieve different degrees of transaction consistency and fault tolerance. For example, while `ecStore` provide transactional semantics for read-modify-write operations spanning across multiple records, most of other cloud data serving systems only ensure ACID

properties at single row level. We are also the pioneer in providing a comprehensive and efficient indexing framework for cloud environments. Further, our proposed load-adaptive replication scheme enables effective load balancing in the system.

Performance comparison

As discussed above, various cloud data serving systems, including `ecStore`, are designed to support different degrees of consistency and fault tolerance, and therefore it is not straight forward to compare these systems just on the performance of a single read or write operation. However, we shall attempt to compare `ecStore` with Cassandra [93] based on their common features such as system scalability and range query processing. Cassandra [93] is an open-source cloud storage that combines the idea of Bigtable [49] and Dynamo [61]. It is notable that both `ecStore` and Cassandra (we use version 0.6.2 in the experiments) are on-going projects and the results here are based on the snapshot of the systems.

In this experiment, we tested the two systems on a set of 18 nodes in the in-house cluster (see Section 7.1 for the cluster configuration) with the YCSB cloud serving benchmark [55]. The systems are initially bulk loaded with 144 GB of data (144 million 1KB records). Each storage node thus maintains an average of 8 GB on disk. The memory buffer for the persistent B^+-tree used in `ecStore` and for the memtable used in Cassandra are set to 64 MB, which is the default setting in the distribution package of Cassandra. To support range query, Cassandra is configured to use *OrderedPartitioner*. Cassandra is also configured to employ asynchronous replication like `ecStore` to reduce the effect of replication on update latency. A workload of 1000 operations is continuously submitted to each node in the system. A completed operation will be immediately followed up with another operation. The record selection for each operation follows uniform distribution.

It is important to note that the data structures that Cassandra and `ecStore` physically maintain data on each storage node are different. Particularly, Cassandra uses SSTable

(String Sorted Table [49], which is similar to Log-Structured Merge tree [112]) while ecStore uses persistent B$^+$-tree (Berkeley Database Java Edition [10]). The difference in local persistence between Cassandra and ecStore results in the different performance of range query, point query (read operation), and write operation.

Range query. Figure 7-45 shows the performance of range can query in ecStore and Cassandra when varying the size of query range. It can be seen that the response time of range query in both systems increases together with the query range size. In addition, ecStore has lower latency with range query because the B$^+$-tree in ecStore supports range query efficiently, while in Cassandra the range query processing might need to check multiple SSTable. As a result, ecStore also has better range query throughput when testing the systems with different system sizes. Figure 7-46 shows this result with the range query size set to 800.

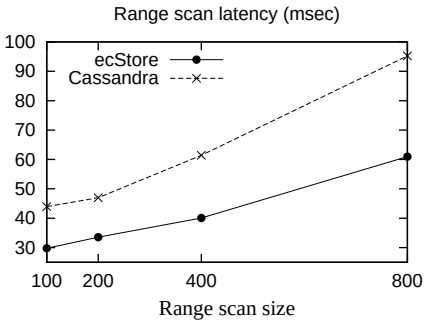


Figure 7-45: Range scan response time.

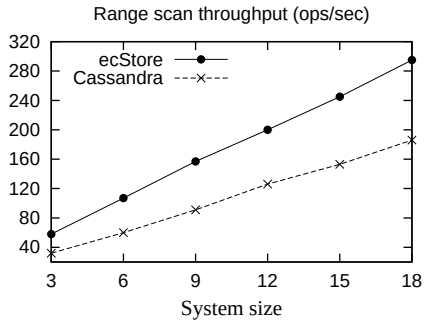


Figure 7-46: Range scan throughput.

Read operation. In contrast to the results of range query performance, Figure 7-47 illustrates that Cassandra has better performance than ecStore in the case of read operations. It is because of the fact that Cassandra uses bloom filters to speed up read operations. The bloom filters help Cassandra efficiently identify which SSTable contains the queried record rather than traversing a long chain of intermediate nodes as in the B$^+$-tree used in ecStore. Thus, there is a *trade-off* on the performance of range query and exact query in Cassandra and ecStore, because of the difference in implementation of the physical store at each node as discussed above.

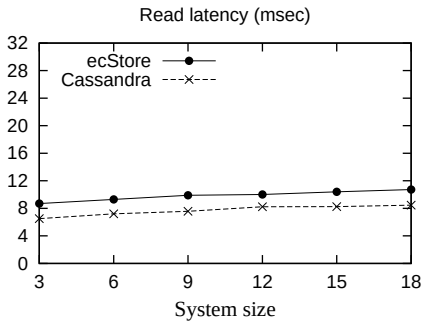


Figure 7-47: Read response time.

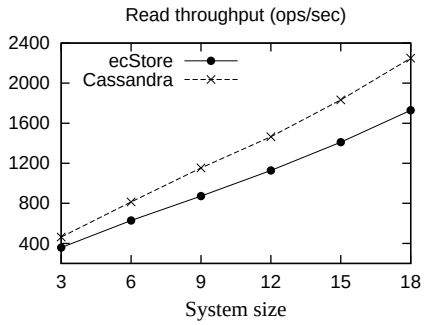


Figure 7-48: Read throughput.

Additionally, the elastic scaling property of Cassandra and `ecStore` are well demonstrated in Figure 7-47. The results show that the read latency in both systems only increases slightly when we increase the system size and the client request load. It confirms that the increased client load can be handled by adding more storage nodes into the system.

Write operation. In Cassandra, writes are batched in a memtable and periodically written to disk, specifically to an SSTable persistent structure, with sequential I/O. For `ecStore` to be competitive, we also employ similar optimization technique. In particular, `ecStore` buffers the write operations in an in-memory B^+-tree and merges out these data to the persistent B^+-tree backing store after a period of time or when the buffer for the in-memory B^+-tree is full. Further, the commit log of both systems is configured to synchronize with disk every 10 seconds by default. Therefore, there is minimal disk I/O at the time of write, and the write operations in Cassandra and `ecStore` have low latency, about 1 msec in the experiment.

7.5 Summary

In this chapter, we have presented an extensive study on the performance characteristics of `ecStore`. The experimental results on various platforms, including the commercial cloud Amazon EC2 [2], an in-house cluster and PlanetLab [17], confirm the scalability, efficiency and robustness of the system. Specifically, `ecStore` can provide elastic

scaling property, i.e., the system aggregate throughput increases when there are more nodes added into the system. Further, the generalized distributed indexing framework developed in `ecStore` has been shown to improve the performance of processing queries on non-key attributes efficiently while keeping the cost of index maintenance minimal. The results also show that the system is able to balance the workload among storage nodes effectively in the presence of skews both in data and query distribution. In the next chapter, we shall conclude the thesis by summarizing our research contributions and indicating future work.

Chapter 8

Conclusions and Future Work

Cloud computing represents a paradigm shift driven by the increasing demand of Web based and enterprise applications for elastic, scalable and efficient system architectures that can efficiently support their ever-growing data volume and large-scale data analysis. With substantial interests in cloud deployment of data-centric applications, cloud storages form an important component in the cloud software stack.

The ultimate goal of this thesis is to address the unique challenges posed by the cloud platforms and propose an efficient and elastic storage service in the cloud with similar capabilities as centralized database systems. In the following, we summarize the main contributions of our research towards this goal (Section 8.1). We then discuss the limitation of our current work and present potential research issues (Section 8.2).

8.1 Summary of the Thesis

The main contributions of our research are linked to the following three aims: (1) building an elastic storage system on top of cloud virtual infrastructures for supporting a combination of OLTP and OLAP workloads, (2) providing generalized distributed indexing functionality for cloud storages, and (3) supporting load-adaptive replication and transactional semantics in the cloud. In the following, we outline the main contributions of our research.

8.1.1 A Hybrid Cloud Storage for Supporting Both OLTP and OLAP

The first part of the thesis investigates how an elastic data storage system can be built on top of cloud virtual infrastructures where machines can be dynamically added into or removed from the system based on load characteristics, while still being able to guarantee data durability and provide highly available data service as well as other important functionalities of a centralized database system. We proposed a new system architecture for supporting database operations in cloud systems spanning clusters of commodity servers. `ecStore` – our proposed **e**lastic **c**loud **stor**ag**e** – provides advanced features for data-centric applications in the cloud, including hybrid storage structure for supporting the combined OLTP and OLAP workload, smart replication for providing both high data availability and automatic load balancing, distributed indexes for improving the performance of query processing and transactional semantics for bundled operations spanning across multiple records, which are important features but missing from most cloud data serving systems.

In order to support both OLTP and OLAP workloads that run simultaneously and interactively within the same storage, `ecStore` devises a hybrid data partitioning scheme that favors both workloads with a careful combined design of vertical and horizontal partitioning based on the trace of query workload. Further, `ecStore` provides snapshot isolation – a widely accepted consistency model – to handle the two workloads simultaneously: OLAP queries run in historical mode by accessing the recent consistent snapshot of the data while OLTP transactions work on the current version of the data. Experimental results on an in-house cluster show that `ecStore` can provide for most OLAP jobs the needed data freshness, i.e., a fresh and consistent snapshot of the data which are simultaneously manipulated by OLTP operations.

`ecStore` provides basic functionalities and data access interfaces for developer users, execution engines (e.g., Hadoop MapReduce [14] and E^3 [52]) and other

applications/tools for submitting data access requests. To facilitate efficient processing of ad-hoc queries, the distributed indexing component of `ecStore` supports declaration of indexes over the distributed data and hence provides efficient data retrieval. Given a specific data access request, the data access optimizer of `ecStore` dynamically chooses a near optimal data access plan, namely parallel sequential scan or index scan or their combination, using a cost-based optimization algorithm that utilizes the statistics information stored in the metadata catalog of the system.

8.1.2 Generalized Distributed Indexing in the Cloud

As we have reviewed in Chapter 3, most cloud data serving systems choose to provide dynamic scalability to take advantage of the elastic characteristic of the new environment as a trade-off for the lack of full DBMS functionalities such as indexing support and transactional semantics. In this second part of the thesis, we addressed the missing feature of supporting DBMS-like indexes in cloud storages and proposed a simple but extensible and efficient indexing framework that enables users to define their own indexes without knowing the structure of the underlying network or having to tune the performance of the system manually. This comprehensive distributed indexing framework supports a set of indexes using P2P overlays and provides a high level abstraction for the definition of new indexes.

The most distinguishing feature of our scheme is the ability to support multiple types of distributed indexes, such as distributed hash, B^+-tree-like and multi-dimensional index, within the same framework, which significantly reduces the maintenance cost and provides the much needed scalability. To achieve this goal, we define two mapping functions, namely overlay mapping and data mapping, to transform different indexes into a generic Cayley graph-based distributed data structure. We exploit the characteristics of Cayley graph to reduce the index creation and maintenance cost, and embed some self-tuning capability such as setting up network connections and buffering indexes adaptively.

The experimental results on the commercial cloud Amazon's EC2 [2] and an in-house cluster confirm the efficiency and scalability of our generalized distributed indexing framework. In particular, the system scales well with flat query latency and linear system throughput when varying the number of index nodes and the number of indexes in the system. In addition, the proposed sampling-based data mapping function guarantees a well balance in storage load and query execution load among index nodes in the presence of skews in both data and query distribution. The distributed indexes also improve the processing of equi-join and range join queries significantly.

8.1.3 Load-adaptive Replication and Transaction Management

We have proposed a comprehensive cloud indexing framework in our second piece of this research. In the last part of the thesis, we deal with the issue of load-adaptive replication and transaction management in cloud storage systems. In particular, `ecStore` supports transactional access which bundles read and write operations spanning across multiple records. `ecStore` also provides high resilience capability with smart replication and complete methods for system recovery from various types of machine failures, which is essential to guarantee data durability requirement – an important service level agreement (SLA) when providing data services on top of cloud virtual infrastructures.

Furthermore, we propose a two-tier partial replication strategy, which is adaptive with the database workload, to enhance the load balancing functionality in `ecStore`. While previous works on replication for load balancing in conventional distributed systems as well as P2P systems maintain the query access statistics on the granularity of data objects, this approach is impractical for cloud-scale databases since the amount of data in the system is typically large, leading to non-trivial overhead for storage and update of access statistics. Therefore, `ecStore` employs self-tuning range histograms to keep the cost of histogram maintenance minimal while being able to deal with skewed access patterns efficiently and creating only a small number of replicas (hence reducing storage cost and replica consistency management cost).

The experimental results on various platforms including the commercial public cloud Amazon's EC2 [2], an in-house cluster serving as private cloud, and PlanetLab [17] representing distributed clouds where machines are geographically located, show that `ecStore` can support a wide range of read consistency and allow for performance trade-off. More importantly, the results confirm the elastic scaling property of `ecStore`, i.e., as the number of storage nodes in the system increases the aggregate system throughput also increases. The experimental results also show that the proposed load-adaptive replication method can effectively balance the system load distribution under skewed workloads. This load-adaptive replication method selectively replicates more copies for the hot data ranges to shed the workload of the overloaded node to other under-loaded nodes. Therefore, `ecStore` can achieve a well balance in system load distribution while keeping the cost of replication – including storage cost and replica consistency maintenance cost – minimal.

8.2 Ongoing and Future Work

In this research, we mainly describe the design and implementation of the storage manager of a bigger cloud data management system named `epiC` [12, 51], and provide the performance evaluation of its main functionalities such as basic data access operations, automatic load balancing, transactional support and distributed indexing. The processing of OLAP and OLTP queries will ride on the functionalities provided by the proposed cloud storage, and the implementing and benchmarking of the whole cloud data management system is our ongoing work. More specifically, the adaptation of conventional query optimization techniques to the cloud environment raises many questions and opportunities for further research.

8.2.1 Freshness-aware Query Processing

A potential research issue is the developing of freshness-aware query processing. In particular, we organize the replicas of a data item into a hierarchy structure of data

freshness guarantee, as depicted in Figure 8-1. At the highest level of the freshness hierarchy are the primary copies of the data, which requires strict consistency (up-to-date data freshness). At the second level of the hierarchy, the replicas can relax the consistency and accept some staleness. The update can be asynchronously propagated to this second level of replicas after a period, which is a configurable parameter. Similarly, at the third level of the hierarchy, the replicas provides even less data freshness compared to the second level; and the frequency of update propagation to this third level of replicas is also lower than the second level.

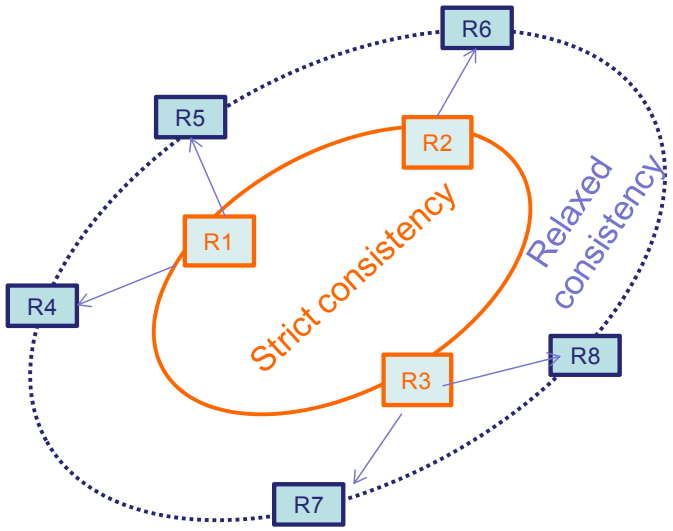


Figure 8-1: Hierarchical freshness of cloud data replication.

With this hierarchy of data freshness, the system can provide a flexible consistency level for the users. Depending on the service level agreement (SLA), the system will attach each user query with a freshness requirement, which will be served by the replicas on the corresponding level of data freshness hierarchy of a specific data item. Thus, the read-only (and stale) replicas on the hierarchical data freshness enable the system to trade the end-user latency and system resource utilization for the timeliness of the queried data. We refer to the query processing strategy that exploits the hierarchical data freshness as freshness-aware query processing.

8.2.2 Replication-aware Query Processing

Another line of research is to improve the performance of OLAP queries in epiC system by exploiting the existing replication provided by ecStore. In ecStore, the data of a certain table are typically partitioned and stored across storage nodes. Furthermore, each partition is replicated on several machines for data availability and durability requirement. The OLAP controller of epiC will transform an input query into a set of sub-queries that are subsequently executed on the processing nodes.

The research issue is how to dynamically choose a specific replica for each partition and assign it to the processing nodes in a suitable manner in order to guarantee the load balance between processing nodes in the system, and therefore improve the query performance. The above approach is referred to as replication-aware query processing. The challenges when developing this query processing strategy lie in two aspects: the load dynamism of the processing nodes at runtime and the separation of the storage component and the processing component of epiC.

Finally, it is also important to note that we can further combine the above two strategies, namely freshness-aware and replication-aware query processing, to develop a comprehensive framework for processing OLAP queries on the replicated data maintained in ecStore.

Bibliography

[1] Amazon cloud services. [Online] http://aws.amazon.com.

[2] Amazon elastic compute cloud (EC2). [Online] http://aws.amazon.com/ec2.

[3] Amazon relational database service (RDS). [Online] http://aws.amazon.com/rds.

[4] Amazon simple storage service (S3). [Online] http://aws.amazon.com/s3.

[5] Apache hadoop. [Online] http://wiki.apache.org/hadoop.

[6] Apache hbase. [Online] http://hbase.apache.org.

[7] Apache hdfs. [Online] http://hadoop.apache.org/hdfs.

[8] Apache pig. [Online] http://hadoop.apache.org/pig.

[9] Apache zookeeper. [Online] http://zookeeper.apache.org.

[10] Berkeley db. [Online]
http://www.oracle.com/technetwork/database/berkeleydb/overview/index.html.

[11] Cloud computing versus grid computing. [Online]
http://www.ibm.com/developerworks/web/library/wa-cloudgrid.

[12] epic project. [Online] http://www.comp.nus.edu.sg/~epiC.

[13] Greenplum mapreduce. [Online] http://www.greenplum.com/technology/mapreduce.

[14] Hadoop mapreduce. [Online] http://hadoop.apache.org/mapreduce.

[15] Nist definition of cloud computing. [Online]
http://csrc.nist.gov/groups/SNS/cloud-computing.

[16] Nosql databases. [Online] http://nosql-database.org.

[17] Planetlab. [Online] http://www.planet-lab.org.

[18] Rackspace cloud services. [Online] http://www.rackspace.com.

[19] Tpch benchmark. [Online] http://www.tpc.org/tpch.

[20] Tpcw benchmark. [Online] http://www.tpc.org/tpcw.

[21] Unifying hadoop, hive and apache cassandra for real-time and analytics. White paper, DataStax, 2011. [Online]
www.datastax.com/wp-content/uploads/2011/03/WP-Brisk.pdf.

[22] Daniel J. Abadi. Data management in the cloud: Limitations and opportunities. *IEEE Data Eng. Bull.*, 32(1):3–12, 2009.

[23] Ashraf Aboulnaga and Surajit Chaudhuri. Self-tuning histograms: building histograms without looking at data. In *Proc. of SIGMOD*, pages 181–192, 1999.

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

[25] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proc. of SIGMOD*, pages 23–34, 1995.

[26] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, and Harbinder Bhogan. Volley: automated data placement for geo-distributed cloud services. In *Proc. of NSDI*, pages 2–2, 2010.

[27] Divyakant Agrawal, Amr El Abbadi, Shyam Antony, and Sudipto Das. Data management challenges in cloud computing infrastructures. In *Proc. of DNIS*, pages 1–10, 2010.

[28] Divyakant Agrawal, Arthur J. Bernstein, Pankaj Gupta, and Soumitra Sengupta. Distributed optimistic concurrency control with reduced rollback. *Distributed Computing*, 2(1):45–59, 1987.

[29] Parag Agrawal, Adam Silberstein, Brian F. Cooper, Utkarsh Srivastava, and Raghu Ramakrishnan. Asynchronous view maintenance for vlsd databases. In *Proc. of SIGMOD*, pages 179–192, 2009.

[30] Rakesh Agrawal, Anastasia Ailamaki, Philip A. Bernstein, Eric A. Brewer, Michael J. Carey, Surajit Chaudhuri, AnHai Doan, Daniela Florescu, Michael J. Franklin, Hector Garcia-Molina, Johannes Gehrke, Le Gruenwald, Laura M. Haas, Alon Y. Halevy, Joseph M. Hellerstein, Yannis E. Ioannidis, Hank F. Korth, Donald Kossmann, Samuel Madden, Roger Magoulas, Beng Chin Ooi, Tim O'Reilly, Raghu Ramakrishnan, Sunita Sarawagi, Michael Stonebraker, Alexander S. Szalay, and Gerhard Weikum. The claremont report on database research. *SIGMOD Rec.*, 37(3):9–19, 2008.

[31] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proc. of SIGMOD*, pages 359–370, 2004.

[32] Marcos K. Aguilera, Wojciech Golab, and Mehul A. Shah. A practical scalable distributed b-tree. *PVLDB*, 1(1):598–609, 2008.

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

[34] S. B. Akers and B. Krishnamurthy. A group-theoretic model for symmetric interconnection networks. *IEEE Trans. Comput.*, 38(4):555–566, 1989.

[35] Shyam Antony, Divyakant Agrawal, and Amr El Abbadi. P2p systems with transactional semantics. In *Proc. of EDBT*, pages 4–15, 2008.

[36] Michael Armbrust, Armando Fox, David A. Patterson, Nick Lanham, Beth Trushkowsky, Jesse Trutna, and Haruki Oh. Scads: Scale-independent storage for social computing applications. In *Proc. of CIDR*, 2009.

[37] Jason Baker, Chris Bond, James Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proc. of CIDR*, pages 223–234, 2011.

[38] Jason Baker, Chris Bond, James Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proc. of CIDR*, pages 223–234, 2011.

[39] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[40] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In *Proc. of SIGMOD*, pages 1–10, 1995.

[41] Philip A. Bernstein, Istvan Cseri, Nishant Dani, Nigel Ellis, Ajay Kalhan, Gopal Kakivaya, David B. Lomet, Ramesh Manne, Lev Novik, and Tomas Talius. Adapting microsoft sql server for cloud computing. In *Proc. of ICDE*, pages 1255–1263, 2011.

[42] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand Aiyer. Apache hadoop goes realtime at facebook. In *Proc. of SIGMOD*, pages 1071–1080, 2011.

[43] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. Building a database on s3. In *Proc. of SIGMOD*, pages 251–264, 2008.

[44] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable isolation for snapshot databases. In *Proc. of SIGMOD*, pages 729–738, 2008.

[45] David G. Campbell, Gopal Kakivaya, and Nigel Ellis. Extreme scale with full sql language support in microsoft sql azure. In *Proc. of SIGMOD*, pages 1021–1024, 2010.

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

[47] Rick Cattell. Scalable sql and nosql data stores. *SIGMOD Rec.*, 39(4):12–27, 2011.

[48] Ronnie Chaiken, Bob Jenkins, Per-Ake Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2):1265–1276, 2008.

[49] 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 *Proc. of OSDI*, pages 205–218, 2006.

[50] David Chappell. A short introduction to cloud platforms: an enterprise-oriented overview. White paper, Chappell & Associates, 2008.

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

[52] Gang Chen, Ke Chen, Dawei Jiang, Beng Chin Ooi, Lei Shi, Hoang Tam Vo, and Sai Wu. E3: an elastic execution engine for scalable data processing. *JIP*, 20(1):65–76, 2012.

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

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

[55] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proc. of SoCC*, pages 143–154, 2010.

[56] Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, 3(1):48–57, 2010.

[57] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Elastras: An elastic transactional data store in the cloud. *CoRR*, abs/1008.3751, 2010.

[58] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *Proc. of SOCC*, pages 163–174, 2010.

[59] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proc. of SCG*, pages 253–262, 2004.

[60] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proc. of OSDI*, pages 10–10, 2004.

[61] 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 *Proc. of SOSP*, pages 205–220, 2007.

[62] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):44–62, 1990.

[63] David J. DeWitt and Jim Gray. Parallel database systems: The future of database processing or a passing fad? *SIGMOD RECORD*, 19(4):104–112, 1991.

[64] 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). *PVLDB*, 3(1-2):515–529, 2010.

[65] Shel Finkelstein, Dean Jacobs, and Rainer Brendle. Principles for inconsistency. In *Proc. of CIDR*, 2009.

[66] Eric Friedman, Peter Pawlowski, and John Cieslewicz. Sql/mapreduce: a practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *PVLDB*, 2(2):1402–1413, 2009.

[67] Shinya Fushimi, Masaru Kitsuregawa, and Hidehiko Tanaka. An overview of the system software of a parallel relational database machine grace. In *Proc. of VLDB*, pages 209–219, 1986.

[68] Eran Gabber, Jeff Fellin, Michael Flaster, Fengrui Gu, Bruce Hillyer, Wee Teck Ng, Banu Özden, and Elizabeth A. M. Shriver. Starfish: highly-available block storage. In *USENIX Annual Technical Conference, FREENIX Track*, pages 151–163, 2003.

[69] Prasanna Ganesan, Mayank Bawa, and Hector Garcia-molina. Online balancing of range-partitioned data with applications to peer-to-peer systems. In *Proc. of VLDB*, pages 444–455, 2004.

[70] Prasanna Ganesan, Beverly Yang, and Hector Garcia-Molina. One torus to rule them all: multi-dimensional queries in p2p systems. In *Proc. of WebDB*, pages 19–24, 2004.

[71] David K. Gifford. Weighted voting for replicated data. In *Proc. of SOSP*, pages 150–162, 1979.

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

[73] Vijay Gopalakrishnan, Bujor Silaghi, Bobby Bhattacharjee, and Pete Keleher. Adaptive replication in peer-to-peer systems. In *Proc. of ICDCS*, pages 360–369, 2004.

[74] Craig Gotsman and Michael Lindenbaum. On the metric properties of discrete space-filling curves. *IEEE Transactions on Image Processing*, 5(5):794–797, 1996.

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

[76] Jim Gray. The transaction concept: virtues and limitations. In *Proc. of VLDB*, pages 144–154, 1981.

[77] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proc. of SIGMOD*, pages 173–182, 1996.

[78] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudré-Mauroux, and Samuel Madden. Hyrise - a main memory hybrid storage engine. *PVLDB*, 4(2):105–116, 2010.

[79] Morton Hamermesh. *Group Theory and Its Application to Physical Problems*. Dover Publications, 1989.

[80] Richard A. Hankins and Jignesh M. Patel. Data morphing: an adaptive, cache-conscious storage technique. In *Proc. of VLDB*, pages 417–428, 2003.

[81] 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 *Proc. of ICDE*, pages 1199–1208, 2011.

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

[83] Yixiu Huang and Ouri Wolfson. A competitive dynamic data replication algorithm. In *Proc. of ICDE*, pages 310–317, 1993.

[84] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proc. of USENIX*, pages 11–11, 2010.

[85] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, 2007.

[86] H. V. Jagadish, Beng Chin Ooi, and Quang Hieu Vu. Baton: a balanced tree structure for peer-to-peer networks. In *Proc. of VLDB*, pages 661–672, 2005.

[87] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. The performance of mapreduce: an in-depth study. *PVLDB*, 3(1-2):472–483, 2010.

[88] Hyungsoo Jung, Hyuck Han, Alan Fekete, and Uwe Röhm. Serializable snapshot isolation for replicated databases in high-update scenarios. *PVLDB*, 4(11):783–794, 2011.

[89] Alfons Kemper and Thomas Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *Proc. of ICDE*, pages 195–206, 2011.

[90] Donald Kossmann. Building web applications without a database system. In *Proc. of EDBT*, page 3, 2008.

[91] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency rationing in the cloud: pay only when it matters. *PVLDB*, 2(1):253–264, 2009.

[92] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6:213–226, 1981.

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

[94] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):1825, 2001.

[95] Per-Ake Larson. Grouping and duplicate elimination: Benefits of early aggregation. Technical report, Microsoft Research, 1997.

[96] Jonathan K. Lawder and Peter J. H. King. Using space-filling curves for multi-dimensional indexing. In *Proc. of BNCOD*, pages 20–35, 2000.

[97] Mong Li Lee, Masaru Kitsuregawa, Beng Chin Ooi, Kian-Lee Tan, and Anirban Mondal. Towards self-tuning data placement in parallel database systems. In *Proc. of SIGMOD*, pages 225–236, 2000.

[98] Philip L. Lehman and S. Bing Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6(4):650–670, 1981.

[99] Justin J. Levandoski, David B. Lomet, Mohamed F. Mokbel, and Kevin Zhao. Deuteronomy: Transaction support for cloud data. In *Proc. of CIDR*, pages 123–133, 2011.

[100] Yi Lin, Bettina Kemme, Marta Patiño Martínez, and Ricardo Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *Proc. of SIGMOD*, pages 419–430, 2005.

[101] 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 *Proc. of SIGMOD*, pages 961–972, 2011.

[102] David Lomet and Mohamed F. Mokbel. Locking key ranges with unbundled transaction services. *PVLDB*, 2(1):265–276, 2009.

[103] David B. Lomet, Alan Fekete, Gerhard Weikum, and Michael J. Zwilling. Unbundling transaction services in the cloud. In *Proc. of CIDR*, 2009.

[104] Peng Lu, Sai Wu, Lidan Shou, and Kian-Lee Tan. An efficient and compact indexing scheme for large-scale data store. In *Proc. of ICDE*, 2013. To appear.

[105] Yijun Lu, Ying Lu, and Hong Jiang. Adaptive consistency guarantees for large-scale replicated services. In *Proc. of International Conference on Networking, Architecture, and Storage*, pages 89–96, 2008.

[106] Mihai Lupu, Beng Chin Ooi, and Y. C. Tay. Paths to stardom: calibrating the potential of a peer-based data management system. In *Proc. of SIGMOD*, pages 265–278, 2008.

[107] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Multi-probe lsh: efficient indexing for high-dimensional similarity search. In *Proc. of VLDB*, pages 950–961, 2007.

[108] Rimma Nehme and Nicolas Bruno. Automated partitioning design in parallel database systems. In *Proc. of SIGMOD*, pages 1137–1148, 2011.

[109] Wee Siong Ng, Beng Chin Ooi, Kian-Lee Tan, and Aoying Zhou. Peerdb: A p2p-based system for distributed data sharing. In *Proc. of ICDE*, pages 633–644, 2003.

[110] Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. Mrshare: sharing across multiple queries in mapreduce. *PVLDB*, 3(1-2):494–505, 2010.

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

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

[113] 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 *Proc. of SIGMOD*, pages 165–178, 2009.

[114] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with sawzall. *Sci. Program.*, 13(4):277–298, 2005.

[115] Hasso Plattner. A common database approach for oltp and olap using an in-memory column database. In *Proc. of SIGMOD*, pages 1–2, 2009.

[116] Dan Pritchett. Base: An acid alternative. *Queue*, 6(3):48–55, 2008.

[117] Changtao Qu, Wolfgang Nejdl, and Matthias Kriesell. Cayley dhts - a group-theoretic framework for analyzing dhts based on cayley graphs. In *Proc. of ISPA*, pages 914–925, 2004.

[118] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 3 edition, 2003.

[119] Jun Rao, Eugene J. Shekita, and Sandeep Tata. Using paxos to build a scalable, consistent, and highly available datastore. *PVLDB*, 4(4):243–254, 2011.

[120] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy Lohman. Automating physical database design in a parallel database. In *Proc. of SIGMOD*, pages 558–569, 2002.

[121] D. Ratajczak and J. M. Hellerstein. Deconstructing dhts. Technical report, Intel Research Berkeley, 2003.

[122] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proc. of SIGCOMM*, pages 161–172, 2001.

[123] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.

[124] Gunter Schlageter. Optimistic methods for concurrency control in distributed database systems. In *Proc. of VLDB*, pages 125–130, 1981.

[125] Mario Schlosser, Michael Sintek, Stefan Decker, and Wolfgang Nejdl. Hypercup: hypercubes, ontologies, and efficient search on peer-to-peer networks. In *Proc. of AP2PC*, pages 112–124, 2002.

[126] S. Seshadri and Jeffrey F. Naughton. Sampling issues in parallel database systems. In *Proc. of EDBT*, pages 328–343, 1992.

[127] Xipeng Shen and Chen Ding. Adaptive data partition for sorting using probability distribution. In *Proc. of ICPP*, pages 250–257, 2004.

[128] Adam Silberstein, Brian F. Cooper, Utkarsh Srivastava, Erik Vee, Ramana Yerneni, and Raghu Ramakrishnan. Efficient bulk insertion into a distributed ordered table. In *Proc. of SIGMOD*, pages 765–778, 2008.

[129] Adam E. Silberstein, Russell Sears, Wenchao Zhou, and Brian Frank Cooper. A batch of pnuts: experiences connecting cloud batch and serving systems. In *Proc. of SIGMOD*, pages 1101–1112, 2011.

[130] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[131] Michael Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

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

[133] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-store: A column-oriented dbms. In *Proc. of VLDB*, pages 553–564, 2005.

[134] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating systems: design and implementation*. Pearson Prentice Hall, Upper Saddle River, NJ 07458, USA, third edition, 2009.

[135] Alexander Thomasian. Distributed optimistic concurrency control methods for high-performance transaction processing. *IEEE Trans. on Knowl. and Data Eng.*, 10(1):173–189, 1998.

[136] Steven K. Thompson. Sample size for estimating multinomial proportions. *The American Statistician*, 41(1):42–46, 1987.

[137] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. Hive - a petabyte scale data warehouse using hadoop. In *Proc. of ICDE*, pages 996–1005, 2010.

[138] Dimitrios Tsoumakos and Nick Roussopoulos. An adaptive probabilistic replication method for unstructured p2p networks. In *Proc. of the Confederated international conference On the Move to Meaningful Internet Systems: CoopIS, DOA, GADA, and ODBASE*, pages 480–497, 2006.

[139] Hoang Tam Vo, Chun Chen, and Beng Chin Ooi. Towards elastic transactional cloud storage with range query support. *PVLDB*, 3(1):506–517, 2010.

[140] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009.

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

[142] Shiyuan Wang, Beng Chin Ooi, Anthony K. H. Tung, and Lizhen Xu. Efficient skyline query processing on peer-to-peer networks. In *Proc. of ICDE*, pages 1126–1135, 2007.

[143] Ouri Wolfson and Sushil Jajodia. Distributed algorithms for dynamic replication of data. In *Proc. of PODS*, pages 149–163, 1992.

[144] Ouri Wolfson, Sushil Jajodia, and Yixiu Huang. An adaptive data replication algorithm. *ACM Trans. Database Syst.*, 22(2):255–314, 1997.

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

[146] Sai Wu, Feng Li, Sharad Mehrotra, and Beng Chin Ooi. Query optimization for massively parallel data processing. In *Proc. of SOCC*, pages 12:1–12:13, 2011.