SCALABLE DATA MANAGEMENT FOR WEB APPLICATIONS

ZHOU WEI

Parts of Chapter 3 have been published in *Proceedings of the 17th International World Wide Web Conference 2008*.
Parts of Chapter 4 have been published in *Proceedings of the 15th International Euro-Par Conference on Parallel Processing 2009*.
Parts of Chapter 4 will also be published in *IEEE Transactions on Services Computing, Special Issue on Cloud Computing 2012*.
Parts of Chapter 5 have been published in *Proceedings of the 12th IEEEACM International Symposium on Cluster, Cloud and Grid Computing 2012*.

VRIJE UNIVERSITEIT

SCALABLE DATA MANAGEMENT FOR WEB APPLICATIONS

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. L.M. Bouter,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Exacte Wetenschappen
op dinsdag 4 december 2012 om 11.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Zhou Wei

geboren te Zhejiang Province, China

promotor:	prof.dr.ir. M.R. van Steen
copromotoren:	dr.ir. G.E.O. Pierre
	prof.dr. C.H. Chi

Acknowledgements

Upon completion of this thesis, it is time for me to express my thanks to all the people who have helped me through this journey.

Let me start by thanking my advisors: Maarten van Steen, Guillaume Pierre and Chi-Hung Chi. Maarten has always been kind and supportive. He inspired me to look into my research in different perspectives so that I could see the full picture. I owe him many thanks for his constructive advices on how to organize this thesis as well as for his reviews which help me to correct some weak arguments.

Guillaume has been a terrific advisor. I owe the most to him for this thesis. He has taught me so many things about how to do good research. The first lesson about research I learnt from him is: "understand the problem well before trying to find a solution." I still remember vividly that in the first a few weeks when I started my Ph.D. study, he almost always began our discussion with the question: "what's the problem?" I really appreciate his guidance and training, which transfers me from an exam-oriented student into a researcher addressing real problems. Throughout the years of my Ph.D. study, he has always been ready and energetic to help me with my research. I am particularly thankful for his remarkable patience and significant effort in reading my poorly-written drafts and making them more readable. Without his guidance and constant support, completing this thesis would be impossible.

Beyond an advisor, Guillaume has also been a great friend to me. He helped me handling all kinds of living problems in Amsterdam, making my stay comfortable and pleasant. Particularly, I want to thank him and Caroline for their kindnesses and hospitality of inviting me to dinner at their home and treating me to the delicious French cuisine every year!

I have to thank Chi-Hung Chi, who has been my mentor and great friend throughout my graduate-school days, including both Master and Ph.D. study. I owe him a lot of thanks for offering me the great opportunity to study in Amsterdam. He has taught me not only a lot about research, but also many more about life and career. I really appreciate his valuable suggestions which guide me to plan my life and career.

I would like to thank the other members in my thesis committee, Dag Johansen, Henri Bal, Peter Boncz, and Thorsten Schütt, for their effort and time in reading and reviewing this thesis.

I have to thank Jiang Dejun and Chen Shuo, my colleagues and great friends, not only for our successful cooperation in research, but also for all the fun that we

have shared together as well. I owe thanks to Chen Shuo for organizing the exciting trips of visiting Brussels and Paris. I would also like to thank Konrad, Guido, Jeff, Corina, Albana, Melanie, Suhail and all my colleagues in the group for the help they gave me as well as the fun we shared. I particularly want to thank Ben van Werkhoven for translating the summary of this thesis into Dutch for me. I owe thanks to Björn Patrick Swift, Plamen Nikolov and Enes Göktaş for their works on improving CloudTPS. Working together with them is fun and inspiring. Moreover, I would like to thank Liu Bo, Li Peng, Yin Si and the Chinese community in Amsterdam. With these friends, I have a wonderful life in this fantastic city.

Finally, I owe so many thanks to my family that I cannot thank them enough with words. I owe a lot to my wife Shi for her understanding and support while I was away studying in Amsterdam. She has offloaded many family duties from me so that I could focus on my research. My parents have given me so much that I cannot repay them in this life. They have always encouraged me to pursue my interests and would unconditionally offer me any support as soon as I need it. Without their steadfast support, I can never be what I am today.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Data are at the heart of all computer applications, as most programs spend a majority of their time reading, processing and writing data stored on a variety of devices. For example, Web applications answer each incoming user request using parameters contained in the request as well as the current state of application data, usually stored in a database. Some requests trigger data updates, which must be reflected in the responses of following user requests. Modern Internet Web applications encourage users to create their own Web content dynamically (e.g., by writing comments or publishing blogs). This potentially introduces a high update load. For example, in Facebook, establishing a friend relationship generates one or more updates in the underlying data store. In 2010, Facebook claimed to have established as many as 1,972,000 such relationships in just 20 minutes [77]. To support efficient Web application operations, data must be organized and stored properly to support information retrieval and data modifications.

An important challenge for Web-based businesses is to maintain good application performance while keeping costs under control. This is true in particular for data management, which is an essential and difficult challenge. For a Web application, the costs are mainly twofold: the development cost of building the application, and the operational cost of maintaining reasonable performance under arbitrary workload.

The time to develop Web applications is preferably small. An effective way to reduce development time is to provide high-level tools that improve the efficiency of programmers. Such tools can range from design techniques such as UML, to high-level platforms such as advanced database systems. When it comes to application implementation, a data store with high-level database functionalities such as complex queries and strong data consistency can greatly improve programming productivity [100]. Indeed, even simple business logic often leads to complex queries that join data from a number of separate tables. Join queries often originate in data normalization which aims to reduce data redundancy and ensure data integrity. Such operations may require simultaneous access to arbitrary data items across the data store. They must be carried out atomically to guarantee operation

correctness. A data store providing strong data consistency can relieve programmers from complex concurrency bugs, and allow them to focus on implementing the business logics of Web applications.

Once a Web application is developed and published online, it is critical to maintain short response times. Customers of eCommerce applications have been shown to start leaving the site if download time exceeds two seconds [28]. Studies on man-computer conversation impose even more demanding goals where one second is the limit for a user's flow of thought to stay uninterrupted [22, 70]. The response time of a Web application includes the time of both executing application code and obtaining data from the data store. To maintain reasonable performance, operators must provide sufficient resources for application servers and the data store according to current workload. However, one challenge is that workloads fluctuate widely over several orders of magnitude according to predictable as well as unpredictable patterns [40, 108]. Under such circumstances, on-demand resource provisioning is much more cost-effective than static over-provisioning. To enable on-demand resource provisioning, the underlying data store should be elastic such that its capacity can grow or shrink efficiently. Adding or removing machines from the data store should require only minimum human effort such that the whole process can be entirely automated.

A related challenge is that Web applications may become extremely popular over short time periods. For example, the workload of Twitter.com increased by a factor of 100 from 2009 to 2011 [107]. Similar workload explosions has also been experienced by many now-famous websites such as Facebook, Google, Amazon, eBay etc. The data store of such applications must therefore be highly scalable. This means that capacity must preferably increase linearly as more resources are provisioned, such that the increased workload can be handled by adding a proportional number of machines. Otherwise, the data store will eventually become a bottleneck.

Scaling to large number of machines introduces a new issue of fault tolerance. In a large-scale distributed system, machine failures happen frequently. The data store must not compromise data consistency in case of such failures. It should recover automatically without interrupting Web applications.

In this perspective, an ideal data store for Web applications should have the following properties.

Ability to store large amounts of data. Web application workloads can trigger a very high load of data updates, which in turn can generate large amounts of data. The data store should be able to store fast-increasing amounts of data, while remaining efficient in executing queries.

Complex query support. Complex queries, such as secondary-key queries and join queries, are necessary to implement business logic efficiently. A data store supporting complex queries can greatly improve the productivity of programmers.

Strong data consistency. To ensure the correctness of business operations, applications should observe a consistent view of the data even under concurrent data updates. We consider that program correctness should not be an optional feature left under the sole responsibility of the programmers. A good data store should therefore realize strong data consistency, so that programmers can focus their attention on implementing business logics.

Elasticity and scalability. Web applications are response-time sensitive. To maintain reasonable performance under widely-fluctuating workloads, the data store should be elastic so that the process of adjusting capacity requires only minimum human efforts. In addition, it should scale linearly so that additional machines can handle any increase of the workload.

Fault tolerance. In large-scale distributed systems, machine failures are common, and network partitions will eventually happen. The data store should be able to recover from such failures automatically. During recovery, the correctness of the application must not be compromised. Data loss is obviously not acceptable.

Web application developers have access to two main families of data stores. First, relational databases such as MySQL and Oracle have been widely used by many types of applications, including Web applications. Relational databases support the SQL language, which allows programmers to easily express business logics into data queries, such as joins, secondary-key queries, aggregations etc. Programmers can group multiple queries into a transaction to guarantee strong data consistency. However, as we shall see further in this thesis, relational databases are not elastic. Scaling relational databases to any number of machines is difficult and often requires careful manual intervention [86]. A second class of systems, called NoSQL data stores, such as Cassandra [61], Amazon Dynamo [35] and Google Bigtable [27], provide excellent scalability and elasticity. However, they lack support for complex queries and strong data consistency [112]. We can see that the two system families have complementary strengths and weaknesses but that none of them completely fulfills all the requirements of an ideal data store for Web applications.

This thesis aims to answer the following question: *is it possible to build a data store that accommodates all these requirements at the same time*? We use two different approaches to achieve this goal: first, we explore to which extent Web applications based on relational databases can be made elastic and scalable. We establish a systematic approach to restructure a Web application and its data, and show that the results exhibit linear scalability. However, this approach requires significant efforts in reimplementing Web applications.

Second, we explore how one may extend the existing NoSQL data stores with high-level database functionalities, such that their properties of scalability and elasticity are not compromised. We implement the missing features in a middleware layer, called CloudTPS which sits between the application and its data store. Our

prototype creates a temporary copy of the application data in the memory of its participant machines. All the added functionalities, such as transactions and complex queries, operate directly on this copy of the data. All updates are checkpointed back to the underlying data store in a lazy fashion such that users observe strong ACID properties even in case of machine failures or network partitions. CloudTPS follows the system model of typical NoSQL data stores, which automatically manages data partitioning across any number of machines. CloudTPS also replicates data items to a specified number of machines. When encountering machine failures or network partitions, CloudTPS can recover automatically without compromising data consistency. By presenting the detail design and implementation of CloudTPS, we believe that this thesis makes an original contribution towards the ideal data store as stated before.

This thesis is organized as follows.

Chapter 2 presents related work and compares various techniques of scalable data management from the perspectives of distributed system management, data partition and replication, transaction management, and fault tolerance respectively.

Chapter 3 presents a systematic way to restructure the relational data of a Web application, such that it achieves linear scalability. The so-called data denormalization process results in separated data partitions stored in independent relational database instances. This approach maintains the same ACID properties as the original centralized application. We also show how to implement complex queries such as join queries and aggregations in this architecture. The results demonstrate that monolithic Web applications can be made fully elastic and scalable using relational databases. However, this approach requires major modifications to the application and also requires manual partitioning of its data.

Chapters 4 and 5 present the design and evaluation of CloudTPS, a middleware between Web applications and NoSQL data store. Chapter 4 presents the general architecture of CloudTPS and the techniques to impose ACID properties in CloudTPS without compromising scalability. CloudTPS supports multi-item transactions, which can atomically read or write arbitrary data items across the data store. To ensure ACID properties, CloudTPS implements decentralized mechanisms to manage atomicity and concurrency control across multiple machines. This chapter also describes the fault tolerance mechanism and membership management protocol which enable CloudTPS to tolerate machine failures and network partitions.

Chapter 5 details the implementation of high-level queries such as join queries and secondary-key queries. Finding matched records for a join query across partitioned data of CloudTPS may require table scanning or even transferring data across the network. This would incur unacceptable long execution times. To address the issue, we create indexes to link the matched records. CloudTPS automatically synchronizes indexes with the application data and executes join queries as transactions.

Finally, chapter 6 concludes the thesis and discusses open issues.

Chapter 2

Related Work

A large number of techniques have been proposed to support scalability and elasticity in data stores. A scalable data store must maintain reasonable performance even under high workload. In general, a data store carries out two types of queries: read-only and read-write queries. To scale read-only queries, data replication is an effective technique where large workloads can be addressed by adding more replicas. However, data replication cannot scale under write-intensive workloads, as consistency demands that each replica be updated upon each read-write query.

Many systems exist to replicate a complete relational database across multiple servers within a cluster [25, 59, 82]. Database replication allows one to distribute read-only queries among the replicas. However, in these solutions, all read-write queries must first be executed at a master database, then propagated and re-executed at all other "slave" databases. A few database systems such as Oracle and PostgreSQL [83] allow one to optimize the re-execution of read-write queries at the slaves by transferring a log of the execution at the master. However, these techniques do not improve the maximum throughput as they require a single master server to execute all read-write queries. The throughput of the master server then determines the total system's throughput.

New techniques exploit knowledge of the application data access behavior. Database query caching relies on high temporal locality, and uses prior knowledge of data overlap between different query templates to efficiently implement invalidations [8, 16, 75, 96]. A query template is a parameterized SQL query whose parameter values are passed to the system at runtime. Partial replication techniques use similar information to reduce the data replication degree and limit the cost of database updates [48, 95]. However, we observe that these techniques work best under very simple workloads composed of only a few different query templates. When the number of templates grows, an increasing number of constraints reduces their efficiency: database caching mechanisms need to invalidate more cached queries upon each update to maintain consistency, and partial replication is increasingly limited in the possible choices of functionally correct data placements.

To scale under write-intensive workloads, it is necessary to partition the data. By splitting data into smaller parts and placing them across different machines, the system can scale better as each machine processes only a fraction of all data updates. However, data partitioning also imposes a number of challenges in implementing database functionalities.

The first challenge toward building a partitioned and scalable data store is the fact that data queries may span multiple partitions. A query may be as simple as reading one data item, or as complex as scanning multiple tables, merging their contents, and returning some aggregation from the result. Efficient query processing requires careful data organization and well-designed data access algorithms, which is a well-understood problem in centralized databases [94]. However, in the context of partitioned and distributed data stores, implementing complex queries in a scalable fashion is much more difficult.

Secondly, the data store has to provide a number of non-functional properties such as high availability and strong data consistency. Providing these properties requires extra mechanisms, which introduces overhead compared to centralized databases. For example, maintaining strong data consistency requires executing specific protocols to synchronize concurrent access to distributed data, while achieving high availability requires data replication and fault tolerance mechanisms.

Thirdly, some desirable properties conflict with each other and cannot be achieved simultaneously. For example, according to the CAP dilemma, one cannot achieve both prefect availability and consistency in the case of network partitions [44]. One must therefore carefully trade off between the properties that a data store should provide to satisfy application requirements.

This chapter surveys research aiming towards the ideal scalable and elastic data store described in Chapter 1. These efforts cover topics ranging from distributed transaction management to distributed query processing techniques. However, analyzing and comparing these efforts is difficult as these works differ not only in their designs but also in the properties they aim to provide. To better understand and compare these works, we propose a framework that covers the important design issues of scalable data stores.

We consider a scalable data store deployed within a data center where machines are connected by high-speed networks and the latency is low. Environments where these properties are not met require different solutions that we discuss separately in Section 2.6.3. When designing a scalable data store, the following five issues need to be addressed:

1. How do we model application data and queries?

2. How do we partition and distribute data across machines?

3. How do we execute complex queries across partitioned and distributed data?

4. How do we ensure data consistency across multiple data items?

5. How do we tolerate machine failures and network partitions?

Although we will discuss these aspects separately, there are many relationships and dependencies between them. For example, fault tolerance mechanisms often require maintaining multiple data replicas across machines. Such design implies overhead in enforcing strong data consistency when these replicas are updated. Furthermore, an efficient solution to one issue may rely on a specific design related to another issue. For example, executing complex queries could require accessing considerable numbers of data items. Placing these data items in the same machine can significantly improve query processing efficiency, but it demands specific support in the data partition method.

These five issues constitute a general framework to understand the design of scalable and elastic data stores. This framework allows us to compare different data store and expose the tradeoffs which lead to these different systems. Compared to centralized databases, the key here resides in the cooperation of multiple nodes rather than the implementation of local operations within a node. The issues of implementing local operations, such as data buffer management, file organization and logging, are well-understood in centralized databases. Many techniques originally developed for centralized databases can be reused in this new context [94]. Therefore, although single-node performance is still an important aspect in designing a scalable data store [101], we will not discuss these issues here, and rather focus on efforts aimed at achieving linear scalability.

A complete scalable data store should also consider other aspects such as security and privacy. These issues are, however, not considered in this thesis, as we mainly focus on solutions to improve the scalability of data stores belonging to a single organization.

This chapter is organized as follows: in Section 2.1 we present the general framework of scalable data stores. In Sections 2.2 to 2.6, we discuss each of the five issues constituting this framework separately and show how various significant research efforts address these problems. We conclude this chapter in Section 2.7.

2.1 Framework

The goal of a scalable data store is to improve the productivity of application programmers and also to reduce runtime operational costs. Supporting complex queries and strong consistency can help programmers to focus on developing business logics, while scalability and elasticity can automate the process of managing the performance of Web applications. We present this framework as a guideline to discuss the issues in providing these desired properties, and compare the solutions in the perspective of these issues.

2.1.1 Desired Properties

To build a data store, one must first understand which properties are required by the targeted applications, and to what extent we need to provide them. As discussed in Chapter 1, a data store should ideally have all of the following properties: *performance, scalability, elasticity, supporting complex queries, strong data consistency, high availability, supporting large data sizes*. In reality, it is very difficult to enforce all these properties simultaneously. One has to trade off between these properties based on the specific requirements of an application.

Scalability and elasticity are two essential properties required by Web applications. Ideally, the data store should allow one to address any increase of workload by adding more machines. Linear scalability is preferred where the maximum supportable throughput of a system increases proportionally at a constant rate as resources are added to the system. However, in reality, infinite scalability is neither possible nor necessary. For large-scale Web applications, a data store should scale to at least thousands of machines. For example, Facebook has split its MySQL database into 4,000 shards in order to handle the site's massive data volume, and is running 9,000 instances of memcached in order to keep up with the number of transactions the database must serve [51]. If Facebook used a cloud data store instead, this data store would have to scale linearly to at least this number of nodes. In addition, when scaling the system to such sizes, administrating the partitioned database would cause a lot of operational costs and being *elastic* would become even more difficult.

Performance requirements of applications are two-fold: throughput requirements and response-time constraints. As a scalable data store can address the throughput requirement by provisioning a sufficient number of machines, the response-time constraint becomes the core issue. Applications may differ in their response time constraints by several orders of magnitudes. For Web applications serving online customers, the constraint for loading an entire page is in the order of 1 second [22, 70], which includes the time for query execution, but also the time for network transfers, load balancing, Web-server and application-server processing. In contrast, for applications carrying out background tasks, response times of tens of seconds may be acceptable [80].

Implementing *complex queries* that access considerable numbers of data items in a scalable fashion is challenging, especially with low latency constraints. An extreme case of an expensive complex query is a full join of multiple tables, each containing tens of millions of records. Some applications trade off the semantics of *complex queries* for *performance*. For example, Web-application programmers may be asked to carefully design their queries such that even complex queries access only a small number of data items [29]. In fact, supporting both *complex queries* and *low latency* may be possible, at the cost of relaxing *data consistency* so that precomputed and possibly outdated results can be stored and used to answer queries.

High availability is critical for web applications, as website down time causes

direct monetary lost and potential loss of customers. However, the property of *high availability* often contradicts the property of *strong consistency* in distributed systems. In the case of network partitions, achieving perfect *availability* means that one must relax *strong consistency* [44]. For the applications which prefer *availability* over *consistency*, it is crucial to ensure application correctness in relaxing *data consistency*.

The size of data is an important factor in designing algorithms and mechanisms to achieve all other properties. For example, with increasingly larger data sizes, ensuring *elasticity* requires that the costs of adding or removing a node should remain largely constant irrespective of the system size. Similarly, *complex queries* may access more data items as data sizes grow; designing an efficient solution satisfying low-latency constraints becomes more difficult. Therefore, one should estimate the size of application data and design the data store accordingly.

2.1.2 Framework Elements

We identify five issues that have to be considered to achieve the desired properties of a scalable data store: data and query model, data-partition mechanism, data-consistency enforcement, complex-query implementation and fault tolerance.

Data and query models define data structures to store application data and query semantics to access the stored data. In general, a data store with a sophisticated data model and rich query semantics can improve programmer efficiency for the applications containing complex business logics. However, more features may also bring more issues in achieving scalability, which often results in more overheads. Therefore, the data store should balance the richness of its data and query model so that targeted applications can be developed efficiently, while keeping costs low. We compare the data and query models of current NoSQL data stores and partitioned relational databases in Section 2.2.

Data partitioning is necessary to be scalable under update-intensive workloads. Based on the data model, data-partitioning mechanisms define how structured data are partitioned and assigned to different nodes. Data-partitioning mechanisms also define how to route queries cross nodes to locate and access the required data items. The performance of query routing is critical for efficient query execution, and in particular complex queries. In addition, data-partitioning mechanisms also affect elasticity as partitioned data should be reorganized when adding or removing nodes. An elastic data store should relocate only a small part of the data during this process. We discuss data-partitioning mechanisms in Section 2.3.

Implementing complex queries in a partitioned and distributed data store introduces new issues compared to centralized databases. For example, complex queries may access considerable numbers of data items across multiple machines. In this context, enforcing strong data consistency to distributed complex queries while preserving the scalability property is even more challenging. As for performance issues, the data store should locate all accessed data items efficiently and

avoid transferring large amount of data across networks in query execution. We discuss how to implement distributed complex queries efficiently in Section 2.4.

Data consistency enforcement requires defining the data consistency model carefully to present programmers with predictable behaviors. Data stores may use various consistency levels, which allow programmers to trade off data consistency and performance while still ensuring application correctness. To implement these consistency levels, specific protocols and mechanisms must be followed to ensure atomicity and manage concurrency control. We discuss the issues of enforcing data consistency in Section 2.5.

Fault tolerance of the data store requires the system to functionally normally even under machine failures and network partitions. The correctness of the application must not be compromised during failures, which means that the promised data consistency must be maintained and data loss is not acceptable. The system should be able to recover from failures automatically without requiring human intervention such as manual data recovery or data re-partitioning. We discuss fault tolerance mechanisms in Section 2.6.

2.2 Data and Query Model

The data model of a data store defines the structure of stored data and presents a logical view for defining the query model for applications to access data. Many NoSQL data stores and horizontally partitioned relational databases have been designed with different data models. We follow the category definition from [24] and classify data stores by their data models into four categories: key-value data stores, document data stores, tabular data stores and relational databases.

2.2.1 Data-Store Classification

Key-value data stores use a simple data model that maps keys to values. The keys are indexed and applications can access a value by giving its associated key. Key-value data stores commonly support the interface of $GET(Key)$/$PUT(Key)$. There are also some further distinctions. For example, Dynamo stores values as uninterpretable binary objects [35], while Voldemort supports values to be stored as structured objects such as JavaScript Object Notation (JSON) objects [113]. However, they both do not index values and support queries based only on keys. Scalaris supports range queries on keys by arranging keys in an ordered fashion [91].

Document data stores also store values by keys, named Primary-Key (PK), but values are stored as documents that can be indexed and queried. A document is a collection of attribute-value pairs. Attribute names can be defined in a document dynamically and different documents need not share the same set of attributes. These data stores support accessing documents by their PKs and attributes. Examples of document data stores are Amazon SimpleDB [6] and MongoDB [1].

Tabular data stores define their data model as tables of columns [1] and rows. The columns of a table are grouped into column families, which form the basic unit of system configuration (for example, access control is typically realized per column family). A column is prefixed with its column family name. For example, Bigtable names a column using the following syntax: "family:qualifier" [27]. The column-families must be defined in advance while applications can define new qualifiers dynamically. Similar to Bigtable, Cassandra also groups columns into column families [61]. Cassandra extends this data model with "super -columns" within a regular column family. In this case, the column syntax is: "column_family:super_column:column." Tabular data stores support similar single-table queries like document data stores and further allow applying the predicates on specific column families. Bigtable and Cassandra have an API with support for inserting, getting and deleting data items, which are all single-row operations. Bigtable also supports scanning a subset of a table, iterating over multiple column families, including mechanisms for limiting the rows, columns, and timestamps produced by a scan. As for Cassandra, its latest version now supports secondary-key queries on indexes it maintains [23]. However, neither Cassandra nor Bigtable support complex queries such as joins.

Relational databases store well-defined tables where all column names and types must be defined in advance [94]. All rows of a table should contain the same set of columns. The properties of each column, such as uniqueness, primary-key, nullable and creating indexes, must also be defined in advance. Relational databases require an explicit definition of application data to maintain data integrity for applications. In this context, modifying a data schema without stopping the application remains a challenge. Another specificity of the relational data model is that it defines foreign-key relationships between tables, which allows them to support complex queries such as joins. Centralized relational databases support the SQL language fully, which provides rich query semantics such as range queries, join queries, secondary-key queries and aggregations. However, to scale a relational database, horizontal data partition is required. An example of a relational database which applies horizontal data partition is VoltDB [56]. This database however supports only a subset of the SQL semantics and aims primarily at OLTP workloads. The data integrity feature may also be relaxed to achieve online schema update.

PNUTS defines its data model as a hybrid between a tabular data store and a relational database [29]. PNUTS presents a simplified relational data model where attributes should be declared in advance. However, like tabular data stores, its schemas are flexible so new attributes can be added dynamically at any time without halting query or update activity. Records are not required to have values for all attributes. PNUTS's query language supports selections and projections from a single table. Updates and deletes must specify the primary key. PNUTS also supports retrieving multiple records from multiple tables, again by primary key.

[1] We consider the terms "column" and "attribute" as interchangeable.

2.2.2 Discussion

To achieve scalability and elasticity, there are two issues to be considered in designing the data and query model.

The first issue is *online schema modification*: modifying the schema should not halt the execution of queries and updates. Some Web applications need to be modified frequently to cope with dynamic business requirements. For example, the programmers of eBay add 300+ features per quarter of a year and produce 100,000+ lines of code every two weeks [93]. Such application modifications often result in updates of the underlying data model. However, many relational databases require all nodes to share the same consistent data model information to execute queries correctly. Updating the schema across all nodes without halting the system or affecting performance is therefore challenging. For such applications, a flexible data model is preferable so that the schema rarely needs to be updated.

NoSQL data stores such as key-value data stores, document data stores and tabular data stores provide flexible schemas that allow to update without halting system activities. Key-value data models are simple and do not need to be modified. Document data stores, such as SimpleDB, are featured by their "schema-less" data model where no attribute needs to be defined in advance. Tabular data stores require to define a few column families for a table in advance, but the columns within a column-family can be defined dynamically.

The second issue is *limiting performance interference between concurrent queries*. Web applications serve large number of concurrent users simultaneously, and each user request can trigger any number of queries to the data store. To satisfy the response time constraint of each individual user, the data store should limit the total impact of any single query, and prevent any query from overloading the shared environment [10]. One method is to restrict the query semantics by supporting only simple data operations such as read or write well-identified data items where their primary keys are given. Another method is to place constraints on the number of accessed data items or the query execution time.

As described in Section 2.2.1, NoSQL data stores typically choose to restrict the query semantics that support only single-table queries but no complex queries across multiple tables. In addition to that, many NoSQL data stores also impose other constraints on queries. For example, PNUTS is designed primarily for workloads that consist mostly of queries that read and write single records or small groups of records [29]. PNUTS expects the number of records accessed by most queries to be no more than a few hundred.

Amazon SimpleDB [6] is a public online database service, where the issue of limiting performance interference between concurrent queries is more important as it is not only among concurrent users of an application but also among multiple unrelated applications. To address this multitenancy issue, SimpleDB restricts queries within one domain only. A domain in SimpleDB can be viewed as a table or a horizontal table partition. Different domains in SimpleDB are deployed on different machines, so queries on different domains will not interfere with each other.

In addition, SimpleDB also imposes various restrictions to limit the runtime impact of a query. For example, the maximum number of comparisons per `SELECT` expression selection is 20; the maximum query execution time is 5 seconds; `SELECT` operations can retrieve at most 2500 rows, a row can have at most 256 attributes [2].

Unlike PNUTS and SimpleDB, Google Bigtable was originally designed for batch processing tasks such as Web page ranking. Limiting query impact was therefore not considered as a major issue in the design. Bigtable provides APIs to write, delete, look up values from individual rows, or iterate over a subset of the data in a table [27]. As the targeted workloads may contain many scan-based operations, Bigtable places no restriction on the number of accessed data items or execution time.

As opposed to NoSQL data stores, traditional relational databases support rich query semantics such as range queries, join queries, secondary-key queries and aggregations. They accept ad hoc queries with no constraint on the number of data items accessed or the length of execution time. With these features, relational databases can satisfy the functional requirements of a wide range of applications. However, complex queries may risk overloading the system and significantly impact the rest of the workload. Such scenarios are not acceptable for hosted data service providers which have Service-Level Agreements (SLA) on guaranteed query response times.

Partitioned relational databases often place limits on the impact of queries. VoltDB supports only stored procedures and does not accept ad hoc queries [56]. It aims for OLTP workloads, which consist of short-lived transactions accessing a small number of records.

2.3 Data Partitioning

There are three main approaches to partition data: horizontal partitioning, vertical partitioning, and hybrid partitioning [72]. Horizontal partitioning splits a table according to the value of a specific column, and clusters rows into a number of subtables. In contrast, vertical data partition splits a table by column names, and each partition contains a subset of columns. A hybrid approach splits a table into table fragments of which each contains a subset of rows and columns.

To scale a data store to thousands or more nodes, the data-partitioning mechanism should be able to create a sufficiently large number of data partitions so that each node can be assigned at least one data partition. When partitioning a table horizontally, the maximum number of data partitions depends on the number of rows in the table. For Web applications, a table can contain millions or even hundreds of millions of records [93]. With such a large number of rows, horizontal partitioning can create sufficiently large number of partitions to scale the data store. On the other hand, with vertical data partitioning, the number of columns defines the scalability's upper bound. The number of columns of a table is usually smaller than the

[2] These numbers are valid as of December 2011.

number of rows by several orders of magnitude. Vertical partitioning is therefore not used for scalability but rather to optimize performance based on the data access pattern of specific applications. Hybrid data partitioning with both horizontal and vertical data partitioning can potentially achieve both scalability and optimized performance. This method may be suitable for specific applications but is hard to apply in practice as a general-purpose measure. In this section we therefore focus on techniques for horizontal data partitioning.

There are three aspects to be considered in horizontal data partitioning: the choice of a partition key, the data-partitioning algorithm and the way queries are routed to the right location.

2.3.1 Partition Key Selection

The first step is to select a key for each table to partition the table horizontally. For key-values data stores, the choice is straightforward as the system is indexed by a single key. But for data stores of the other three categories, the primary key or any other column could be selected as the partition key. One criterion is to optimize performance so that the system can efficiently locate the accessed rows. Therefore, the most commonly used attribute for identifying the accessed rows should be chosen as the partition key. Otherwise, most queries would need to reach every node of the data store to locate the accessed rows, unless the data store has maintained indexes on the demanded attributes. For workloads composed mainly of primary-key queries such as `SELECT * FROM table WHERE pk='value'`, the primary key should be the partition key. On the other hand, if the workload contains a majority of secondary-key queries, such as `SELECT * FROM table WHERE sk='value'`, that the secondary-key presented in queries may be the partition key.

Relational databases are designed to support ad hoc queries, including complex read queries containing predicates on one or multiple secondary keys. For example, MySQL Cluster allows programmers to specify a column or even a user-defined expression as the partition key, which allows performance optimizations for complex read queries [71]. On the other hand, NoSQL data stores such as Bigtable, Cassandra and SimpleDB typically partition data by primary key only. An important reason is that NoSQL data stores often target write-intensive workloads, where most write queries are based on primary key. Another reason is the fact that flexible data models impose no constraint on columns other than the primary key. It means that a column might be missing values in many rows or have only a limited set of possible values (e.g., a column containing country names). Selecting such a column as the partition key would lead to load imbalance.

2.3.2 Data-Partitioning Algorithms

Once one has chosen a partition key, rows can be partitioned accordingly. In general, horizontal data-partitioning algorithms define the mapping relationship between any partition-key value and the responsible nodes of current system.

In general, current algorithms for horizontal data partitioning can be categorized into three classes: range-based, hash-based and workload-specific algorithms.

There are three issues to be considered in data-partitioning algorithms. The first issue is *optimizing query performance* so that the system can scale linearly. To achieve this goal, queries should access only a small number of nodes in the partitioned data store. However, complex queries can access a considerable number of rows. To execute these queries efficiently, the algorithm should cluster the rows which are accessed together into the same partition so queries can be executed within a single node. For example, range queries access data by key ranges. If rows are assigned to nodes in a random fashion, range queries must be issued to every node to obtain a complete result. However, if continuous key ranges are assigned to nodes, range queries need to access only the nodes responsible for the accessed ranges.

The second issue is *load balancing*: the workload should be distributed evenly across all partitions. An imbalanced data store where most of the workload access a small number of data partitions cannot scale linearly. For well-balanced workloads which address uniform numbers of queries to all data items, load balancing can be achieved by splitting data into partitions with equal number of data items. However, many workloads do not distribute queries uniformly across data items. Therefore, the algorithm must ensure that hot data are split and distributed evenly across all partitions.

The third issue is *efficient data reorganization* across the data store when adding or removing machines. The goal of addressing this issue is to achieve elasticity. The process of data reorganization should be automatic and it should cause only minor financial and performance overheads. Considering a system with large data volume, data reorganization should involve only a small part of the data.

Range-based algorithms order tables by their partition keys and split them into ranges. These ranges are the units of data distribution and load balancing across machines. Reads of short row ranges are efficient and typically access only a small number of machines. When adding or removing a machine, only the ranges held by this machine need to be relocated. For example, Bigtable maintains data in lexicographic order by row key and partitions tables into ranges called tablets [27]. Each tablet is assigned to one machine at a time. The system meta-data is partitioned along the same principles. A `METADATA` table maintains the location of tablets. This `METADATA` table itself is partitioned into tablets and treated like a regular table.

Hash-based data structures are widely used to address random-access workloads. This technique can also be applied to data partitioning. Hash-based algo-

rithms split a table not by the values of the partition key directly, but by the values of its hashed partition key. Consistent hashing has been widely used to design partitioning algorithms [57]. For example, Dynamo partitions data using consistent hashing with MD5 as the hash function [35]. It extends consistent hashing with virtual nodes to address non-uniform data and load distribution. Instead of mapping each node to a single hash-key range, Dynamo assigns multiple ranges (called virtual nodes) per node. The number of virtual nodes assigned to a node can be adjusted according to its processing capacity. This organization also reduces the overhead for adding or removing nodes to the system as the load on the other nodes can be distributed.

Workload-based algorithms partition data based on access patterns. The idea is to minimize the number of nodes accessed for executing queries. This can be achieved by grouping the data items that are often accessed together. Data fragmentation techniques have been commonly used in the design of distributed relational databases [72, 73, 79, 41]. In these works, tables are partitioned either vertically or horizontally into smaller fragments and then allocated to geographically distributed nodes. These systems often assume that the partitions are deployed across a WAN so communications between partitions must be minimized. These algorithms therefore partition data according to a workload analysis which aims to execute as many queries locally as possible.

New workload-based algorithms have emerged recently to partition data in large-scale data stores. For example, Schism analyzes a query log to propose a data placement which minimizes the number of partitions involved in transactions [31]. Schism requires no exact prior knowledge of queries, but directly analyzes the query and transaction log. Schism synthesizes this information into a graph where each node represents a record and edges connect records that are used within the same transaction. Edge weights account for the number of transactions that accesses the same pair of records. Schism applies graph-partitioning algorithms to split a graph into k non-overlapping partitions such that the overall cost of the cut edges is minimized, while keeping the weight of partitions within a constant factor of perfect balance. This work is promising, however, the analysis of the transactions access log is done offline in a centralized and unscalable manner. It is therefore suitable only for applications with sufficiently stable workloads. Similar techniques have been applied for online analysis and dynamic re-partitioning in the context of CloudTPS [103].

Compared to range-based and hash-based algorithms, workload-based algorithms can improve performance in the case of complex query workloads where an efficient data-partitioning scheme is difficult to design by simpler methods. With better data locality, complex queries such as join queries can be executed efficiently as many network communications can be avoided. However, workload-based algorithms must address the issue of workload changes in dynamic environments. Changes in the workload require to constantly re-evaluate the data-partitioning scheme [58]. These associated costs of data reorganization can potentially compromise the elasticity of data stores.

2.3.3 Query Routing

Data-partitioning algorithms need to route queries to the nodes which hold the concerned data items.

For algorithms based on consistent hashing, there is a tradeoff between performance and robustness. For example, Chord has been to designed for unstable P2P environments where nodes can join and leave freely [98]. Chord improves the robustness of consistent hashing by avoiding the requirement that every node knows about every other node. In an N-node network, each node maintains information about $O(logN)$ other nodes, which can route any query. A query may need to route between multiple nodes before reaching the destination nodes, and a lookup requires $O(logN)$ network messages. On the other hand, joining and leaving of nodes only affects a small part of the system.

In data centers where node failures are rare, optimizing for performance is often preferred to robustness. Dynamo targets latency-sensitive applications that require at least 99.9% of read and write operations to be performed within a few hundred milliseconds [35]. Instead of using DHT routing, it maintains the complete membership information across all nodes of the system, such that queries are routed in 1 hop only.

For range-based algorithms, the key-range assignment information has to been known to route queries. Bigtable employs a master node to manage the key-range assignment information. The master node is responsible for assigning tablets to nodes, detecting the addition and expiration of nodes, and balancing load across nodes. Nodes cache this key-range assignment locally to route queries without looking up locations remotely.

PNUTS [29] supports both hash-based and range-based data partitioning. Tables are ordered by the partition key, then partitioned into tablets and assigned to different servers. A binary search tree of tablets is maintained to efficiently map each record to a tablet. PNUTS uses standalone query routers, which maintain a cached copy of the mapping information and can directly route queries to the right node. The mapping itself is owned by the table controller. Routers maintain only soft states. If a router fails, PNUTS can simply start a new one instead. Query routers detect changes in data locations when query routing fails. They can simply reload the new mapping from the table controller.

2.4 Complex Query Support

Centralized relational databases commonly support the SQL language, which provides rich semantics for complex queries such as secondary-key queries, range queries, join queries and aggregations. Implementing these complex queries in a centralized database is a well-understood topic [94]. However, supporting them in distributed and partitioned data stores remains a challenge.

A number of research efforts have been conducted to implement specific types of complex queries in distributed databases. However, they typically rely on a spe-

cific data-partitioning design targeted at one particular type of complex query. For example, the hash-join algorithm is widely used for join queries and it can be naturally parallelized for a distributed environment [90]. Parallel hash-join algorithms partition tables by the hash value of the join attribute. Records that have the same hashed value of the join attribute are assigned to the same machine, so identifying matching records can be executed by each machine locally. However, as the partitioning key is the join attribute rather than the primary key, such data-partitioning design is efficient only for queries that give the value of the join attribute in advance or request a full join of two tables. This partitioning scheme implies large overheads for executing other queries such as primary-key queries and other join queries.

Centralized databases typically rely on indexes to implement complex queries efficiently. In a partitioned and distributed data store, one option is to maintain scalable and distributed B-trees of indexed columns [3]. By replicating all inner nodes and partitioning the leaf nodes of the B-tree, such a distributed data structure allows efficient data location and retrieval. However, this technique faces challenges in the case of update-intensive workloads, as this causes frequent updates of the inner nodes. In addition, for large data sizes, replicating all inner nodes at all machines may be prohibitively expensive.

Another approach to implement a scalable general-purpose database is to run any number of database engines in the cloud, and use the cloud's file system as the shared storage medium [17]. Each engine has access to the full data set and can therefore support any type of SQL queries. However, cloud file systems usually have very high latency compared to an ordinary local disk drive. A complex query accessing many data items would require accessing the cloud file system multiple times, resulting in very long query execution times.

As described in Section 2.3, NoSQL data stores typically partition data by their primary key. Such data-partitioning design is therefore not efficient for complex queries where the primary keys of accessed records are not given in advance. Bigtable [27] supports secondary-key and range queries by scanning the ordered table with user-defined filters on values of columns. However, this algorithm is not efficient for response-time-sensitive Web applications. SimpleDB [6] supports secondary-key and range queries automatically by indexing data. This clearly improves performance compared to Bigtable. However, such queries are restricted within a single table partition (or "domain" in SimpleDB terminology). Cassandra [23] maintains indexes of user-specified columns and supports secondary-key queries over these indexed columns. The indexes are maintained in a new column-family with the value of the indexed column as the primary key. Cassandra partitions these index records in a hash-based approach so range queries cannot be supported. To our best knowledge, no scalable data store supports complex queries across tables, such as join queries. We will return to this topic in Chapter 5 where we discuss join query support in CloudTPS.

2.5 Consistency Enforcement

A data store with strong data consistency can help programmers to ensure application correctness effectively. Centralized relational databases commonly provide strong data consistency in the form of ACID properties: Atomicity, Consistency, Isolation and Durability. Programmers can group queries into ACID transactions, which are executed atomically. Implementing ACID transactions in centralized databases is a well-understood topic [94]. However, providing strong data consistency in a distributed and partitioned data store remains a challenge.

According to the CAP theorem, in a distributed and partitioned data store, one cannot achieve two of the three properties of perfect Availability, strong Consistency and network-Partition tolerance [44]. It is obviously unrealistic to assume the absence of partitions. However, as discussed in [18], the CAP theorem is more subtle than a simple binary choice between availability and consistency. For specific applications, one can trade off these two properties in several ways. For example, during a partition, one may accept incoming client requests but delay their execution until the network partition is restored. By entering such an offline mode, one can maintain strong data consistency while only slightly reducing availability. Other applications may tolerate temporal data inconsistency but reconcile inconsistent updates at the end of the partition with techniques such as version vectors and commutative operations. This hides the temporary inconsistency for the users.

However, these techniques which trade off availability and consistency come at a cost. They heavily rely on the semantics of specific applications. For example, to reconcile inconsistent data updates, programmers need to define application-specific rules in merging the inconsistent versions of data. Furthermore, the applications should also be implemented in a partition-aware way to cope with potentially inconsistent data during a partition. These tasks require programmers to understand the subtle issues of distributed systems as well as the tradeoff between consistency and availability in the presence of network partitions, which is not trivial at all. Therefore, it would require significant efforts from programmers to apply these techniques correctly and effectively. On the other hand, these techniques cannot provide the Durability property of the ACID properties, as queries running in one partition would not reflect updates realized in another partition. This introduces an extra burden on programmers to manage data consistency and ensure application correctness.

This section discusses techniques for enforcing strong data consistency in partitioned and distributed data stores, with no tradeoff of consistency for availability in the presence of network partitions. These techniques do not require the knowledge of the semantics of specific applications. As the design of the tradeoff techniques focuses on fault tolerance rather than data consistency, we will describe them separately in Section 2.6.

2.5.1 Distributed Transactional Systems

There have been decades of research efforts in efficiently implementing distributed ACID transactions in distributed database systems [79]. A number of distributed commit protocols [50, 65, 99] and concurrency control mechanisms [13, 14] have been proposed to maintain the ACID properties of distributed transactions. However, as distributed databases use the same relational data model as RDBMSs, they also cannot partition the data automatically and thus lack scalability. On the other hand, we will see in Chapter 4 how we can apply these techniques as building blocks for CloudTPS. For example, we rely on 2-Phase Commit [65] as the distributed commit protocol for ensuring Atomicity, and on timestamp-ordering [12] for concurrency control.

H-Store is a distributed main-memory OLTP database, which executes on a cluster of shared-nothing main-memory executor nodes [56, 102]. H-Store supports transactions accessing multiple data records with SQL semantics, implemented as predefined stored procedures written in C++. It also replicates data records to tolerate machine failures. H-Store focuses on absolute system performance in terms of transaction throughput, and achieves very high performance on each executor node. However, H-Store's scalability relies on careful data partitioning, such that most transactions access only one executor node. H-Store does not maintain persistent logs in non-volatile storage. H-Store thus faces risks of losing data in the case of system wide outage.

Sinfonia is a distributed message-passing framework which supports transactional access to in-memory data across a distributed system [4]. It addresses fault tolerance by primary-copy replication and by writing transactionally consistent backups to disk images. In contrast with our work, Sinfonia provides a low-level data access interface based on memory address and supports only transactions with restricted semantics. Besides, it requires applications to manage data placement and caching themselves across the distributed system. Sinfonia targets infrastructure applications which require fine-grained control of data structures and placement to optimize performance. On the other hand, Web applications usually require quick and flexible development, so Web developers prefer accessing logical and location-transparent data structures with rich-semantic transactions.

Transactional memory (TM) systems support transactional access to in-memory data. They traditionally target single multiprocessor machines [53], but recent research works extend them to distributed systems and support distributed transactions across in-memory data of multiple machines [15, 60, 66]. Distributed TM systems however provide no durability for transactions and do not address machine failures. The reason is that TM systems are mainly designed for parallel programs that solve large-sized problems. In this case, only the final results are valuable and required to be durable. On the other hand, TM systems usually execute in a managed environment where machine failures are rare. They thus provide no durability for intermediate transactions and do not address machine failures to maximize the

system's performance. However for interactive Web applications, the result of each transaction is critical. TM systems are therefore not suitable for such applications.

2.5.2 Transactions in NoSQL data stores

Typical NoSQL data stores, such as Bigtable, SimpleDB and PNUTS, do not provide full ACID consistency as this allows to execute queries more efficiently. Therefore, these systems support only single-row transactions. Every read or write of data under a single row key is atomic (regardless of the number of different columns being read or written in the row). They do not support ACID transactions across multiple row keys, although they typically provide an interface for batching writes across row keys at the client side.

A number of recent systems support multi-item ACID transactions, each with their own focus.

Google Megastore is a transactional storage system built on top of Bigtable [11]. Megastore supports transactional consistency within fine-grained partitions of data, but only limited consistency guarantees across them. Megastore is designed to provide transactional consistency for Web applications. However, join queries are not supported: join queries require one to be able to access any set of data items within a transaction, whereas Megastore restricts transactions within partitions.

Microsoft SQL Azure Database is a scalable cloud data service which supports the relational data model and ACID transactions containing any SQL query [69]. However, similar to Megastore, it requires manual data partitioning and does not support distributed transactions or queries across data partitions located in different servers.

Deuteronomy is a transactional layer that can operate on top of a wide range of heterogeneous data sources, including cloud data stores [64]. Deuteronomy mostly focuses on issues deriving from the multiplicity of data sources rather than on supporting complex types of operations.

Elastras supports scalable transactions in the cloud [32]. It splits the transaction manager into multiple ones, where each manager loads a specific data partition from the cloud storage service and owns exclusive access to it. However, the system does not address the problem of maintaining ACID properties in the presence of machine failures. Furthermore, similar to Sinfonia [4], it allows only restricted transactional semantics for distributed transactions across multiple data partitions.

G-Store provides transactional access to multiple records in the underlying key-value data store [33]. G-store allows applications to select arbitrary records to form a group. A record can belong only to a single group at any instant of time. G-store assigns the "ownership" of this group to one member node, which coordinates atomic access to this group of records. Such design is suitable for workloads that access relatively fixed sets of records together. Otherwise, frequent creation and modification of record groups may introduce extra costs.

Scalaris is a transactional DHT which splits data across any number of DHT nodes, and supports transactional access to any set of data items addressed by pri-

mary key [81]. It is a purely in-memory system so it does not support durability for the stored data. In contrast, CloudTPS provides durability for transactions by checkpointing data updates into the cloud data service. Scalaris relies on the Paxos algorithm to implement transactions, which can address Byzantine failures, but introduces high costs for each transaction. Moreover, each query requires one or more requests to be routed through the DHT, potentially adding latency and overhead. Cloud computing environments are also expected to be much more reliable than typical peer-to-peer systems, which allows us to use more lightweight mechanisms for fault tolerance.

Google Percolator provides multi-row ACID transactions on top of Bigtable [80]. Percolator employs Bigtable as a shared memory for all instances of its client-side library to coordinate transaction management. The data updates and transaction coordination information, such as locks and primary node of a transaction, are directly written into Bigtable. Using Bigtable's single-row transactions, Percolator can perform multiple actions atomically on a single row, such as locking a data item and marking the primary node of the transaction. Activities such as deadlock handling may delay responses up to minutes, but this is acceptable for Percolator as it is designed for incremental processing of massive data processing tasks which typically have relaxed latency requirements.

2.6 Fault Tolerance

High availability is an important property for many Web applications as any service interruption may negatively impact businesses. However, providing uninterrupted accesses to Web applications faces many technical challenges. One of them is that machines or any other devices that support the execution of Web applications may fail. An ideal data store should tolerate various types of failures and always execute queries correctly (with strong consistency) while remaining always available. However, according to the CAP theorem, such an ideal data store cannot be realized in the case of network partitions [44]. Section 2.5 discussed research efforts in achieving strong consistency even in the case of network partitions, possibly at the expense of high availability. This section turns to techniques to tolerate various failures, but might relax consistency in the case of a network partition.

2.6.1 Replication and Consistency

The main idea to achieve fault tolerance is to replicate application data and system states, so that the state of a failed machine can be recovered from another machine. With redundant information, fault tolerance mechanisms can be developed to recover failed machines automatically and the system can continue working even in the case of machine failures.

Replication is an efficient technique for fault tolerance, but it also introduces issues of maintaining consistency between replicas. When data are updated, all

replicas should be synchronized. The replication mechanism should ensure that updates are stored and correctly reflected in future reads, even when some replicas fail.

There are in general two approaches to enforce data consistency across replicas: the primary-replica approach and the decentralized approach.

The primary-replica approach assigns one replica to process all read-write transactions and then propagates updates to the slave replicas, while read-only transactions can be balanced across all replicas. A typical example is master-slave replication in relational databases. In this case, each replica holds a full copy of the database. Ganymed offers a master-slave replication middleware which addresses update propagation, transaction scheduling and automatic failure recovery [82]. However, such full database replication is not scalable.

Some scalable data stores also apply primary-based replication but at a finer granularity, such as file chunks or records, rather than the complete database. For example, Bigtable relies on the Google File System (GFS) [43] to achieve fault tolerance. GFS divides files into fixed-size chunks and applies primary-based replication at the granularity of chunks. Another example is PNUTS, which applies a record-level mastering mechanism for asynchronous replication across wide-area clusters.

In contrast to the primary-based approach, the decentralized approach maintains no primary replica and all replicas have the same role and responsibility. As a result, to obtain the consistent value of a data item, additional mechanisms are required to reach consensus among all replicas.

Quorum-based protocols are typically used to reach consensus between replicas. These protocols maintain N replicas for each data item. The system is consistent if $N_R + N_W > N$ and $N_W > N/2$, where N_R is the number of replicas to be accessed for a read operation, while N_W is the number of replicas to be accessed for a write operation. For example, Dynamo relies on this technique to perform read and write operations consistently [35]. Dynamo extends the protocol to tolerate machine failures: in the case of a machine failure, another node is selected to temporarily act as the failed replica. To provide high availability, Dynamo modifies the protocol and allows applications to set $N_W < N/2$, which may compromise data consistency. For instance, applications can have the highest level of availability by setting $N_W = 1$ to ensure that a write is accepted as long as a single node has successfully committed the updates.

A more general fault-tolerant approach based on quorums is the Paxos protocol, which solves the general problem of reaching consensus on the state of 2F+1 replicas, while tolerating up to F failures [63]. Paxos is used by various data stores to achieve fault tolerance [11, 84, 88]. Another example is Google Chubby, which is a fault-tolerant system providing a distributed locking mechanism and storing small files [20, 26]. Bigtable uses the Chubby system to store its metadata and act as the coordinator for global system mechanisms such as data partition assignment.

Primary-based approaches typically achieve better performance than Paxos as

Paxos involves multiple network round-trips to reach consensus between replicas. However, in synchronous master-slave replication, some failure sequences may compromise consistency even with just one node failure at a time [84].

2.6.2 Eventually Consistent Systems

Some systems choose to maintain high availability even in the case of network partitions, at the cost of relaxing data consistency.

Dynamo is an always-writable data store which therefore processes updates even in the case of network partitions [35]. During network partitions, each partition still receives data updates from its clients but cannot synchronize with other disconnected partitions. In a system which guarantees strong data consistency, the system should reject these updates to remain consistent until the network is restored and synchronization is possible. However, Dynamo prefers high-availability and instead chooses to relax strong data consistency and provide *eventual consistency*. This means that all replicas will eventually become consistent after an indeterministic amount of time [112]. Dynamo tags each update with version numbers and vector clocks [62] to help clients to reconcile divergent versions.

To support flexible tradeoffs between high-availability and consistency, some scalable cloud data stores such as Amazon SimpleDB and Cassandra support eventual consistency as an option. For example, Amazon SimpleDB supports eventual consistency as its default consistency level but also supports single-row transactions.

2.6.3 Replication Across Data Centers

The motivations for deploying a data store across multiple data centers are twofold: 1) tolerating possible outages of an entire data center, which might be caused by power outage or natural disasters; 2) improving performance by placing data at the data center which is the closest to the clients. However, operating across multiple data centers introduces new issues for synchronizing replicas as the latency between data centers is much greater than within a data center. The common solution is to relax consistency for ensuring faster reads which obtain the value directly from the local data center without synchronization across multiple data centers.

Dynamo maintains a preference list of nodes for replication of each data item [35]. Dynamo is configured such that the nodes in the preference list of each data item reside in multiple data centers. As Dynamo employs a quorum-based protocol to synchronize replicas, programmers can trade off consistency and performance by tuning the configurations of parameter N_R and N_W, as long as $N_R + N_W \geq N$. For example, setting $N_R = 1$ means all reads are performed locally. In cases such as network partitions where replicas might not be synchronized, stale data could be returned. While setting $N_R = N$ means all N replicas should be accessed and the latest value is returned.

Similar to Dynamo, Cassandra provides various replication policies such as "Rack Unaware", "Rack Aware" (within a data center), "Datacenter Aware" [61]. With the latter two policies, Cassandra performs data partitioning by ensuring that replicas of a data item reside in different racks of data centers.

PNUTS uses a primary-replica approach to replicate application data across multiple data centers [29]. PNUTS is mainly designed for social-network applications which typically observe significant write locality on a per-record basis. For each record, PNUTS selects the data center receiving the most workload as its master. This master handles all updates on the record, even if the update was received by another data center. The committed updates are then propagated asynchronously to slave replicas in other data centers via a topic-based publish/subscribe system that guarantees correct message delivery. PNUTS ensures that all replicas will be synchronized even in case of single broker machine failure. To address performance issues of cross-data-center operations, it provides the consistency level of "Read-any" which directly obtains the value stored in the local data center.

Unlike PNUTS, Megastore applies Paxos to synchronize replicas across multiple data centers [11]. Megastore uses Bigtable [27] for scalable fault-tolerant storage within a single data center, and provides a middleware to coordinate operations across data centers.

2.7 Conclusion

This chapter discussed the large variety of techniques used for a partitioned and distributed data store. We provided a generalized framework for such systems, which consists of five components: data and query model, data partition mechanism, data consistency enforcement, complex query implementation and fault tolerance.

We have seen that the desired properties of an ideal data store often contradict each other. Achieving all of them simultaneously is not possible, so it is important for each data store to clearly identify the desired properties and the priorities of the targeted applications. The most important contradiction is between availability and consistency in the case of network partitions. Some data stores relax data consistency to achieve better availability as well as performance and scalability. Other data stores maintain strong data consistency but trade off other properties such as scalability, elasticity, query semantics and fault tolerance.

In subsequent chapters, we propose two different middleware systems which aim to satisfy the following properties simultaneously: scalability, elasticity, supporting complex queries, strong data consistency and fault tolerance. In the case of network partitions, our systems choose strong data consistency over high availability.

Chapter 3

Relational Data Denormalization for Scalable Web Applications

This thesis discusses techniques to enable scalable data management for Web applications. To maintain reasonable performance under dynamic workloads, Web applications demand a data store which is scalable to both read-only and read-write queries.

Two main families of databases exist for this purpose. On the one hand, relational databases offer high-level query semantics and strong consistency, but often lack scalability. On the other hand, NoSQL databases are highly scalable and fault tolerant but they support only very simple queries and weak forms of consistency. In this chapter we will see how one may build scalable Web applications while using relational databases. This requires restructuring application data into multiple data partitions and storing them across separate relational databases. We present a methodology for restructuring an application's data schema and show that the restructured Web application scales linearly with no loss of transactional properties. The next chapters follow the opposite approach and discuss how inherently-scalable NoSQL databases can be extended to support strong consistency and complex queries.

Centralized relational databases provide many useful features to improve programmer efficiency. For example, they provide a rich-semantic query model, supporting many types of complex queries. In addition, they also support executing any set of queries atomically by grouping them into an ACID transaction. To scale relational databases across multiple machines while preserving these features, one may apply classical techniques such as master-slave database replication. However, Web applications issue a mix of read-only and UDI (Update-Delete-Insert) queries. Each UDI query must be issued to all database replicas so that replicas can remain synchronized. Therefore, database replication is not scalable to update workloads.

Besides master-slave database replication, new techniques exploit knowledge of the application data access behavior. As discussed in Chapter 2, database query

caching relies on high temporal locality, and uses prior knowledge of data overlap between different query templates to efficiently implement invalidations [8, 16, 75, 96]. A query template is a parameterized SQL query whose parameter values are passed to the system at runtime. Partial replication techniques use similar information to reduce the data replication degree and limit the cost of database updates [48, 95]. However, we observe that these techniques work best under very simple workloads composed only of a handful of different query templates. When the number of templates grows, an increasing number of constraints reduces their efficiency: database caching mechanisms need to invalidate more cached queries upon each update to maintain consistency, and partial replication is increasingly limited in the possible choices of functionally correct data placements.

Vertical and horizontal data fragmentation techniques have been commonly used in the design of distributed relational database systems [72, 73, 79, 41]. In these works, tables are partitioned either vertically or horizontally into smaller fragments and then allocated to distributed nodes. However, these systems often assume that the partitions are deployed across a WAN so they strive to optimize access time, while changes in the workload require to constantly re-evaluate the data-fragmentation scheme [58]. Dynamic environments such as Web applications would make such an approach impractical.

In [42], the authors propose an edge-computing infrastructure where the application programmers can choose the best-suited data replication and distribution strategies for the different parts of application data. By carefully reducing the consistency requirements and selecting the replication strategies, this approach can yield considerable gains in performance and availability. However, it requires that the application programmers have significant expertise in domains such as fault-tolerance and weak data consistency.

In this chapter, we propose to restructure the application data according to prior knowledge of queries and transactions, so that the data store can be scaled. We establish a systematic approach to vertically partition the application data into independent data services, each of which having exclusive access to its private data store. This restructuring by itself does not lead to linear scalability directly. However, each of the data services has reduced workload complexity, which allows for a more effective application of the optimization techniques such as database replication, query caching and horizontal data partitioning, thus leading to significantly better scalability. For example, read-only data services can be scaled simply by database replication, while update-intensive data services can be scaled more effectively by horizontal data partitioning. Importantly, the restructuring does not imply any loss in terms of transactional or consistency properties. This aspect makes our approach unique compared to the work of [42]. Besides, our approach differs from the data fragmentation techniques, as we propose a one-time modification in the application data structure. Further workload fluctuations can be handled by scaling each service independently according to its own load.

Restructuring a monolithic Web application composed of Web pages that address queries to a single database into a group of independent Web services query-

ing each other requires one to rethink the data structure for improved performance – a process sometimes named *denormalization*. Data denormalization is largely applied to improve the performance of individual databases [87, 92]. It consists of creating data redundancy by adding extra fields to existing tables so that expensive join queries can be rewritten into simpler queries. This approach implicitly assumes the existence of a single database, whose performance must be optimized. In contrast, we apply similar denormalization techniques in order to scale the application throughput in a multi-server system. Denormalization in our case allows one to distribute UDI queries among different data services, and therefore to reduce the negative effects of UDIs on the performance of replicated databases.

To demonstrate the effectiveness of our proposal, we study three Web application benchmarks: TPC-W [67], RUBiS [9] and RUBBoS [85]. We show how these applications can be restructured into multiple independent data services, each with a very simple data access pattern. We then focus on the UDI-intensive data services from TPC-W and RUBiS to show how one can host them in a scalable fashion. For RUBBoS, this is almost trivial. Finally, we study the scalability of TPC-W, the most challenging of the three benchmarks, and demonstrate that the maximum sustainable throughput grows linearly with the quantity of hosting resources used. We were thus able to scale TPC-W by an order of magnitude more than traditional systems.

This chapter is structured as follows. Section 3.1 details our system model and the issues that we need to face. Section 3.2 presents the process of data denormalization, while Section 3.3 shows how individual data services can be scaled. Section 3.4 presents performance evaluations. Finally, Section 3.5 concludes this chapter.

3.1 System Model

3.1.1 Goal

The idea behind our work is that the data access pattern of traditional monolithic Web applications is often too complex to be efficiently handled by a single scalability technique. Indeed, proposed techniques work best under specific simple access patterns. Data replication performs best with workloads containing few or no UDI queries; query caching requires high temporal locality and not too many UDI queries; partial replication or even data partitioning demand that queries do not span multiple partitions.

We claim that major gains in scalability can be obtained by restructuring Web application data into a collection of independent data services, where each service has exclusive access to its private data store. While such restructuring does not provide any performance improvement by itself, it considerably simplifies the data-access pattern generated by each service. This allows one to apply appropriate scaling techniques to each service.

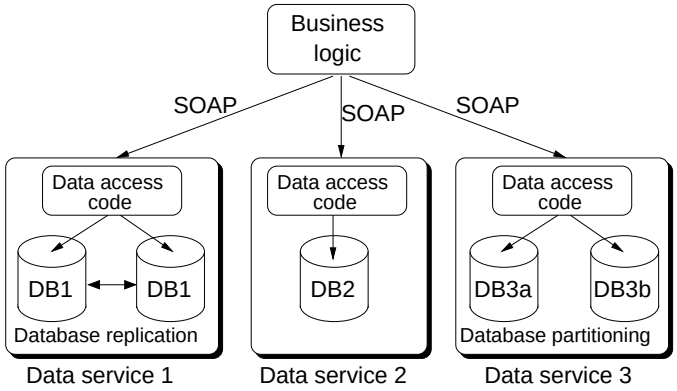


Figure 3.1: System model

Figure 3.1 shows the system model of a Web application after restructuring. Instead of being hosted in a single database, the application data are split into three separate databases DB1, DB2 and DB3. Each database is encapsulated into a data service which exports a service interface to the application business logic. Each data service and its database can then be hosted independently using the technique that suits it best according to its own data access pattern. Here, DB1 is replicated across two database servers, DB2 is hosted by only one server, while DB3 has been further partitioned into DB3a and DB3b. Note that splitting the application data into independent services also improves separation of concerns: details about the internal hosting architecture of a data service are irrelevant to the rest of the application.

3.1.2 Data Denormalization Constraints

Denormalizing an application's data into independent data services requires deep changes to the structure of the data. For example, a table containing fields $\langle key, attr1, attr2 \rangle$ and queried by templates "`SELECT key FROM Table where attr1 = ?`" and "`SELECT key FROM Table where attr2 = ?`" may be split into two tables $\langle key, attr1 \rangle$ and $\langle key, attr2 \rangle$, which may belong to two different data services.

However, not all tables can be split arbitrarily. In practice, data accessed by different queries often overlap, which constrains the denormalization. We identify two types of constraints: transactions and query-data overlap.

Although database transactions are known as an adversary to performance, they sometimes cannot be avoided. An example is a checkout operation in an e-commerce application where a product order and the corresponding payment should be executed atomically. ACID requirements provide a strong motivation for maintaining all data accessed by one transaction inside a single database, and therefore inside a single data service. Splitting such data into multiple services would impose executing distributed transactions across multiple services, for ex-

ample, using protocols such as 2-phase commit. We expect that this would negate the performance gains of the data decomposition.

Another source of constraints is created by ordinary queries executed outside transactions. Similar to constraints created by transactions, it seems logical to cluster data accessed by each query. However, in most cases the overlap of different queries would lead to creating a single data service. Instead, we can apply two other transformations. First, certain complex database queries can be rewritten into multiple, simpler queries. Doing this reduces the data interdependency and allows better data restructuring. Second, data dependencies induced by overlapping queries can be reduced by also replicating certain data to multiple services. However, this implies a trade-off between the gains of splitting the data into more services and the costs of replicating update queries to these data over multiple services.

3.1.3 Scaling Individual Data Services

In all our experiments, we noticed that the services resulting from data denormalization maintain extremely simple data structures and are queried by very few query templates. Such a simple workload considerably simplifies the task of hosting services in a scalable fashion. For example, some data services receive very few or even no UDI queries at all. Such services can therefore benefit from massive caching or replication. On the other hand, some other services are subject to large numbers of UDI queries, often grouped together inside transactions. Such services are clearly harder to scale. However, they at least benefit from the fact that they receive less queries than the database of a monolithic application would. Additionally, we show in Section 3.3.1 that such services can often be *partitioned* so that UDI queries are distributed across multiple database servers.

3.2 Data Denormalization

Service-oriented data denormalization exploits the fact that UDI queries and transactions often access only a part of the columns of a table. Decomposing such tables into multiple smaller ones helps distributing UDI queries and transactions to more data services, and thereby simplifies their workload. As discussed in Section 3.1, two main constraints must be taken into account when denormalizing an application's data. First, one should split the data into the largest possible number of services, such that no transaction or UDI query in the workload spans multiple services. Second, one must make sure that read queries can continue to operate over the then partitioned data.

3.2.1 Denormalization and Transactions

As discussed in previous sections, we need to cluster the data into services such that no transaction overlaps multiple data services. To this end, we first mark which data

columns are accessed by each transaction. Then, simple clustering techniques can be applied to decompose the data into the largest possible number of independent data services.

We distinguish three types of "transactions" that must be taken into account here. First, real database transactions require ACID properties. This means that all the data they access must be accessed atomically and must be placed into the same service. One exception to this rule is formed by data columns that are never updated, neither by the transaction in question nor by any other query in the workload. An example is the table that matches zip codes to local names. Such read-only data does not need to be placed in the same data service, and can be abstracted as a separate data service.

The second type of transaction is a so-called "atomic set," where only the atomicity property of a normal transaction is necessary. Atomic sets appear, for example, in TPC-W, where a query that reads the content of a shopping cart and the one that adds another element must be executed atomically [105]. For such atomic sets, only the columns that are updated must be local to the same data service to be able to provide atomicity. Columns that are only read by the atomic set can reside outside the service, as they are not concerned by the atomicity property[1].

Finally, UDI queries that are not part of a transaction must be executed atomically, and therefore must be considered as an atomic set composed of a single query.

Once one has marked each transaction, UDI query and atomic set with the data columns that should be kept in a single service, simple clustering techniques can provide the first step of decomposition of the database columns into services. However, this step is not functional, as it accommodates only the needs of transactions and UDI queries. To become functional, one must further update this data model to take read queries into consideration.

3.2.2 Denormalization and Read Queries

Clearly, one can consider read queries similarly to UDI queries and transactions, and cluster data services further such that no read query overlaps multiple services. However, applying this method would increase the constraints to the data decomposition and lead to coarse-grain data services, possibly with a single data service for the whole application.

Instead, as shown in Figure 3.2, two different operations can be applied. First, certain read queries can be rewritten into a series of multiple subqueries, where each subquery can execute in one data service. For example, in TPC-W, the `CUSTOMER` and `ORDER` tables are located in different data services, whereas the following query spans both tables with a join operation: "`SELECT o_id FROM customer, orders WHERE customer.c_id = orders.o_c_id AND c_uname = ?`". However, this query can be easily rewritten into two subqueries

[1] In the case of actual database transactions, these data columns must reside inside the data service to be able to provide the Isolation part of ACID properties.

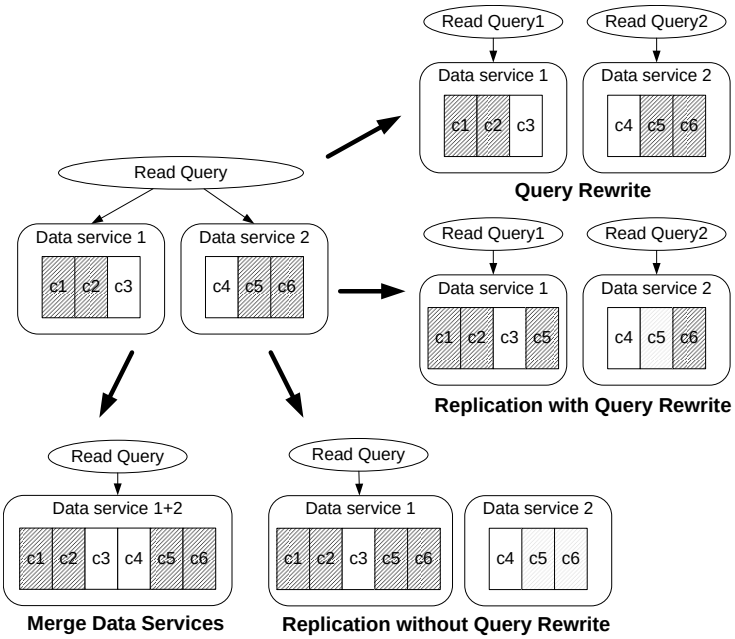


Figure 3.2: Different denormalization techniques for read queries

that access only one table: i) "`SELECT c_id FROM customer WHERE c_uname = ?`"; and ii) "`SELECT o_id FROM orders WHERE o_c_id=?`". The returned result of the first query is used as input for the second one and the final result is returned by the second query.

Another transformation often applied in traditional database denormalization techniques consists of replicating data from certain database tables to other tables. This allows one to transform join queries into simpler queries. Note that traditional denormalization applies this technique to optimize the efficiency of query execution within a single database whereas we apply this technique to be able to split the data into independent data services. For example, the following query accesses two tables in two different data services: "`SELECT item.i_id,item.i_title FROM item,order_line WHERE item.i_id = order_line.ol_i_id AND item.subject = ? LIMIT 50`". Replicating column `i_subject` from table `ITEM` to the other data service allows one to transform this query and target a single data service. The only constraint is that any update to the `i_subject` column must be applied at both data services, preferably within a (distributed) transaction. This scheme is therefore applicable only in cases where the data to be replicated are rarely updated.

To conclude, complex query rewriting should be the preferred option if the semantics of the query allows it. Otherwise, column replication may be applied if the replicated data are never or seldom updated. In last resort, when neither query

rewriting nor column replication is possible, merging the concerned data services is always correct, yet at the cost of coarse-grain data services.

3.2.3 Case Studies

To illustrate the effectiveness of our data denormalization process, we applied it to three standard Web applications: TPC-W, RUBiS and RUBBoS.

TPC-W

TPC-W is an industry standard e-commerce benchmark that models an online bookstore similar to `Amazon.com` [67]. Its database contains 10 tables that are queried by 6 transactions, 2 atomic sets, 6 UDI queries that are not part of a transaction, and 27 read-only queries.

First, the transactions and atomic sets of the TPC-W workload impose the creation of four sets of transactions whose targeted data do not overlap. The first set contains transaction `Purchase`, and the two atomic sets `Docart` and `Getcart`; the second set contains the `Adminconfirm` transaction, the third set contains only the `Updaterelated` transaction. Finally, the last set contains `Addnewcustomer`, `Refreshsession` and `Enteraddress`. This means for example that the original `ITEM` table from TPC-W must be split into five tables: `ITEM_STOCK` contains the primary key `i_id` and the column `i_stock`; table `ITEM_RELATED` contains `i_id` and `i_related1-5`; table `ITEM_DYNAMIC` contains `i_id`, `i_cost`, `i_thumbnail`, `i_image` and `i_pub_date`; the last table contains all the read-only columns of table `ITEM`.

The result of the first denormalization step is composed of five data services: a *Financial* data service contains tables `ORDERS`, `ORDER_ENTRY`, `CC_XACTS`, `SHOPPING_CART`, `SHOPPING-_CART_ENTRY` and `ITEM_STOCK`; data service *Item_related* takes care of items that are related to each other, with table `ITEM_RELATED`; data service *Item_dynamic* takes care of the fields of table `ITEM` that are likely to be updated by means of table `ITEM_DYNAMIC`; finally, data service "Customer" contains customer-related information with tables `CUSTOMER`, `ADDRESS` and `COUNTRY`. The remaining tables from TPC-W are effectively read-only and are clustered into a single data service. This read-only data service can remain untouched, but for the sake of the explanation we split it further during the second denormalization step.

The second step of denormalization takes the remaining read queries into account. We observe that most read queries can either be executed by a single data service, or be rewritten. One read query cannot be decomposed: it fetches the list of the best-selling 50 books that belong to a specified subject. However, the list of book subjects `i_subject` is read-only in TPC-W, so we replicate it to the *Finan-*

Data service	**Data Tables (included columns)**	**Requests**
Financial	ORDERS ORDER_ENTRY CC_XACTS I_STOCK(i_stock) SHOPPING_CART SHOPPING_CART_ENTRY	getLastestOrderInfo createEmptyCart addItem refreshCart resetCartTime getCartInfo getBesterIDs computeRelatedItems purchase
Customer	CUSTOMER ADDRESS COUNTRY	getAddress setAddress getCustomerID getCustomerName getPassword getCustomerInfo login addNewCustomer refreshSession
Item_dynamic	ITEM_DYNAMIC(i_cost i_pub_date i_subject i_image i_thumbnail)	getItemDynamicInfo getLatestItems setItemDynamicInfo
Item_basic	ITEM_BASIC(i_title i_subject) Author	getItemBasicInfo searchByAuthor searchByTitle searchBySubject
Item_related	ITEM_RELATED(i_related1-5)	getRelatedItems setItemRelated
Item_publisher	ITEM_PUBLISHER(i_publisher)	getPublishers
Item_detail	ITEM_DETAIL(i_srp i_backing)	getItemDetails
Item_other	ITEM_OTHER(i_isbn i_page i_dimensions i_desc i_avail)	getItemOtherInfo

Table 3.1: Data services of the denormalized TPC-W

cial data service for this query[2]; `i_subject` is also replicated to the *Item_dynamic* data service for a query that obtains the list of latest 50 books of a specified subject.

The remaining read-only data columns can be further decomposed according to the query workload. For example, the "Search" Web page accesses data only from columns `i_title`, `i_subject` and table `AUTHOR`. We can thus encapsulate them together as the *Item_basic* service. We similarly created three more read-only data services.

The final result is shown in Table 3.1. Note that, although denormalization takes only data access patterns into account, each resulting data service has clear semantics and can be easily named. This result is in line with observations from [46], where examples of real-world data services are discussed.

RUBBoS

RUBBoS is a bulletin-board benchmark modeled after `slashdot.org` [85]. It consists of 8 tables requested by 9 UDI queries and 30 read-only queries. RUBBoS does not contain any transactions. Six tables incur UDI workload, while the other two are read-only. Furthermore, all UDI queries access only one table. It is therefore easy at the end of the first denormalization step to encapsulate each table incurring UDI queries into a separate data service.

All read queries can be executed in only one table except two queries which span two tables: one can be rewritten into two simpler queries; the other one requires to replicate selected items from `OLD_STORIES` into the `USERS` table. The `OLD_STORIES` table, however, is read-only so no extra cost is incurred from such replication. Finally, the two read-only tables are encapsulated as separate data services.

RUBBoS can therefore be considered as a very easy case for data denormalization.

RUBiS

RUBiS is an auction site benchmark modeled after `eBay.com` [9]. It contains 7 tables requested by 5 update transactions. Except for the read-only tables `REGIONS` and `CATEGORIES`, the other five tables are all updated by INSERT queries, which means that they cannot be easily split. This means that the granularity at which we can operate is the table. The transactions impose the creation of two data services: the "Users" data service contains tables `USERS` and `COMMENTS`, while the "Auction" data service contains tables `BUY_NOW`, `BIDS` and `ITEMS`. The final result of data denormalization is shown in Table 3.2.

RUBiS is a difficult scenario for denormalization because none of its tables can be split following the rules described in Section 3.2.1. We note that in such worst-case scenario, denormalization is actually equivalent to the way GlobeTP [48]

[2]Note that we cannot simply move this column into the *Financial* service, as it is also accessed in combination with other read-only tables.

Data Service	Tables	Transactions
User	USERS[U] COMMENTS[C]	Storecomment(U,C) Registeruser(U)
Auction	ITEMS[I] BUY_NOW[N] BIDS[B]	Storebuynow(I,N) Registeritem(I) Storebid(I,B)
Categories	CATEGORIES	-
Regions	REGIONS	-

Table 3.2: Data services of RUBiS

would have hosted the application. We will show however in the next section that scaling the resulting data services is relatively easy.

3.3 Scaling Individual Data Services

In all cases we examined, the workload of each individual data service can be easily characterized. Some services incur either read-only or read-dominant workload. These services can be scaled up by classical database replication or caching techniques [97]. Other services incur many more UDI queries, and deserve more attention as standard replication techniques are unlikely to provide major performance gains. Furthermore, update-intensive services also often incur transactions, which makes the scaling process more difficult. Instead, partial replication or data-partitioning techniques should be used so that update queries can be *distributed* among multiple servers. We discuss two representative examples from TPC-W and RUBiS and show how they can be scaled up using relatively simple techniques.

3.3.1 Scaling the Financial Service of TPC-W

The denormalized TPC-W contains one update-intensive service: the *Financial* service. This service incurs a database update each time a client updates its shopping cart or does a purchase. However, all tables from this service, except one, are indexed by a shopping cart ID and all queries span exactly one shopping cart. This suggests that, instead of replicating the data, one can *partition* them according to their shopping cart ID.

The *Financial* data service receives two types of updates: updates on a shopping cart, and purchase transactions. The first one accesses two tables `SHOPPING_CART` and `SHOPPING_CART_ENTRY`. Table `SHOPPING_CART` contains the description of a whole shopping cart, while `SHOPPING_CART_ENTRY` contains the details of one entry of the shopping cart. If we are to partition these data across multiple servers, then one should keep a shopping cart and all its entries at the same server.

The second kind of update received by the *Financial* service is the

```
1 Insert into ORDER with o_id=id;
2 Insert into CC_XACTS with cx_o_id=id;
3 foreach item i within the order do
4 |   Insert into ORDER_ENTRY with ol_o_id=id, ol_i_id=i;
5 |   Update I_STOCK set i_stock=i_stock-qty(i) where i_id=i;
6 end
7 Update SHOPPING_CART where sc_id=id;
8 Delete from SHOPPING_CART_ENTRY where scl_sc_id=id;
```

Algorithm 1: The purchase transaction

`Purchase` transaction. We present this transaction in Algorithm 1. Similar to the `Updatecart` query, the `Purchase` transaction requires that the order made from a given shopping cart is created at the same server that already hosts the shopping cart and its entries. This allows one to run the transaction within a single server of the *Financial* service rather than facing the cost of a distributed transaction across replicated servers.

One exception to this easy data partitioning scheme is the `ITEM_STOCK` table, in which any element can potentially be referred to by any shopping cart entry. One simple solution would be to replicate the `ITEM_STOCK` table across all servers that host the *Financial* service. However, this would require to run the `Purchase` transaction across all these servers. Instead, we create an `ITEM_STOCK` table in each server of the *Financial* service in which all item details are identical except the available stock which is *divided* by the number of servers. This means that each server is allocated a part of the stock that it can sell without synchronizing with other servers. Only when the stock available at one server is empty, does it need to execute a distributed transaction to re-distribute the available stock.

The *Financial* service receives two more read queries that access data across multiple data clusters. These queries retrieve respectively the 3333 and 10,000 latest orders from tables `ORDERS` and `ORDER_ENTRY` in order to obtain either the list of best-selling items or the items most related to a given other item. We implement these queries in a similar way to distributed databases. Each query is first issued at each server. The results are then merged into a single result set, and the relevant number of most recent orders is re-selected from the merged results.

In our implementation, we wanted to balance the load imposed by different shopping carts across all servers of the *Financial* service. We therefore marked each row of tables `SHOPPING_CART`, `SHOPPING_CART_ENTRY` and `ORDERS` with a key equal to the shopping cart ID. We then partition the tables horizontally by this key and distribute the records of each table evenly across all servers. Records are assigned to these servers using hash partitioning. Our experiments show that this simple approach balances the load effectively in terms of data storage size and computational load.

This example shows that, even for relatively complex data services, the fact

that each service has simple semantics and receives few different queries allows one to apply application-specific solutions. The resulting relative complexity of the service implementation, however, remains transparent to other parts of the application, which only need to invoke a simple service interface.

3.3.2 Scaling RUBiS

The denormalized RUBiS implementation contains two update-intensive services: "Auction" and "User." Similar to the previous example, most queries address a single auction or user by their respective IDs. We were thus able to partition the data rows between multiple servers. A few read-only queries span multiple auctions or users, but we could easily rewrite them such that individual queries would be issued at every server before their results can be merged.

3.4 Performance Evaluation

As we have seen, RUBBoS and RUBiS are relatively simple to host using our denormalization technique. RUBBoS can be decomposed into several rarely updated data services. On the other hand, RUBiS requires coarser-grain update-intensive services, but they can be scaled relatively easily. We present here performance evaluations of TPC-W, which we consider as the most challenging of the three applications.

Our evaluations assume that the application load remains roughly constant, and focus on the scalability of denormalized applications. To support the fluctuating workloads that one should expect in real deployments, a variety of techniques exist to dictate when and how extra servers should be added or removed from each individual data service of our implementations [2, 30, 37, 38, 109].

We compare three implementations of TPC-W. "OTW" represents the unmodified original TPC-W implementation. We then compare its performance to "DTW", which represents the denormalized TPC-W where no particular measure has been taken to scale up individual services. Finally, "STW" (scalable TPC-W) represents the denormalized TPC-W with scalability techniques enabled. All three implementations are based on the Java implementation of TPC-W from the University of Wisconsin [55]. For performance reasons we implemented the data services as servlets rather than SOAP-based Web services.

We first study the performance of OTW and DTW to investigate the costs and benefits of data denormalization with no scalability techniques being introduced. We then study how replication and data partitioning techniques allow us to scale individual data services of TPC-W. Finally, we deploy the three implementations on an 85-node cluster and compare their scalability in terms of throughput.

3.4.1 Experimental Setup

All experiments are performed on the DAS-3, an 85-node Linux-based server cluster [34]. Each machine in the cluster has a dual-CPU / dual-core 2.4 GHz AMD Opteron DP 280, 4 GB of memory and a 250 GB IDE hard drive. Nodes are connected to each other with a gigabit LAN such that the network latency between the servers is negligible. We use Tomcat v5.5.20 as application servers, PostgreSQL v8.1.8 as database servers, and Pound 2.2 as load balancers to distribute HTTP requests among multiple application servers.

Before each experiment, we populate the databases with 86,400 customer records and 10,000 item records. Other tables are scaled according to the benchmark requirements. The client workload is generated by Emulated Browsers (EBs). We use the number of EBs to measure the client workload. The workload model incorporates a think time parameter to control the amount of time an EB waits between receiving a response and issuing the next request. According to the TPC-W specification, think times are randomly distributed with exponential distribution and average value 7 seconds.

TPC-W defines three standard workloads: the browsing, shopping and ordering mixes, which generate 5%, 20% and 50% update interactions respectively. Unless otherwise specified, our experiments rely on the shopping mix.

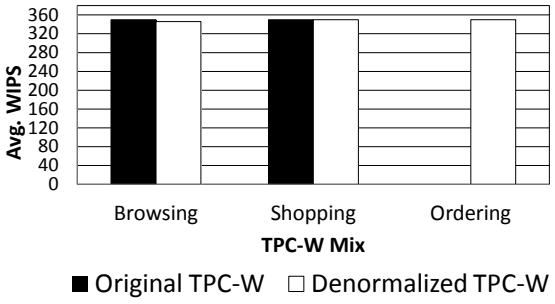
3.4.2 Costs and Benefits of Denormalization

The major difference between a monolithic Web application and its denormalized counterpart is that the second one is able to distribute its UDI workload across multiple machines. Even though such an operation implies a performance drop when hosting the application on a single machine, it improves the overall system scalability when more machines are used. In this section, we focus on the costs and benefits of data denormalization when no special measure is taken to scale the denormalized TPC-W.

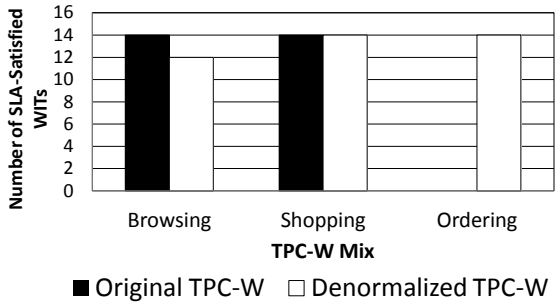
We exercise the OTW and DTW implementations using 2500 EBs, under each of the three standard workload mixes. Both systems are deployed over one application server and 8 database servers. In the case of OTW, the database servers are replicated using the standard PostgreSQL master-slave mechanism. DTW is deployed such that each data service is hosted on a separate database server.

We measure the system performance in terms of WIRT (Web Interaction Response Time) as well as WIPS (Web Interactions Per Second). According to the TPC-W specification, we defined an SLA in terms of the 90th percentile of response times for each type of Web interaction: namely, 90% of Web interactions of each type must complete under 500 ms. The only exception is the “Admin confirm” request type, which does not have an SLA requirement. This request is issued only by the system administrator, and therefore does not influence the client-perceived performance of the system.

Figure 3.3 shows the performance of the different systems under each work-



(a) Average throughput comparison



(b) SLA-satisfied Web Interaction Type number comparison

Figure 3.3: Throughput and performance comparison between original TPC-W and denormalized TPC-W. Note that the Ordering mix for the original TPC-W overloaded and subsequently crashed the application.

load. Figure 3.3(a) shows the achieved system throughput, whereas Figure 3.3(b) shows the number of query types for which the SLA was respected.

The browsing mix contains very few UDI queries. Both implementations sustain roughly the same throughput. However, the denormalized TPC-W fails to meet its SLA for two out of the 14 interaction types. This is due to the fact that the concerned interactions heavily rely on queries that are rewritten to target multiple, different data services. These calls are issued sequentially, which explains why the corresponding request types incur higher latency.

At the other extreme, the ordering mix contains the highest fraction of UDI queries. Here, DTW sustains a high throughput and respects all its SLAs, while OTW simply crashes because of overload. This is due to the fact that DTW *distributes* its UDI queries across all database servers while OTW *replicates* them to all servers. Finally, the shopping mix constitutes a middle case where both implementations behave equally good.

We conclude that data denormalization improves the performance of UDI queries at the cost of a performance degradation of rewritten read queries. We note, however, that the extra cost of read queries does not depend on the number of server machines, whereas the performance gain of UDI queries is proportional to

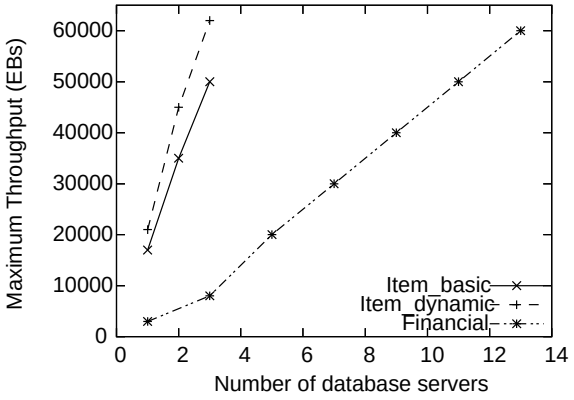


Figure 3.4: Scalability of individual TPC-W services

the size of the system. This suggests that the denormalized implementation is more scalable that the monolithic one, as we will show in the next sections.

3.4.3 Scalability of Individual Data Services

We now turn to study the scalability of each data service individually. We study the maximum throughput that one can apply to each service when using a given number of machines, such that the SLA is respected.

Since we now focus on individual services rather than the whole application, we need to redefine the SLA for each individual data service. As one application-level interaction generates on average five data service requests, we roughly translated the interaction-level SLA into a service-level SLA that requires 90% of service requests to be processed within 100 ms. The *Financial* service is significantly more demanding than other services, since about 10% of its requests take more than 100 ms irrespectively of the workload. We therefore relax its SLA and demand that only 80% of queries return within 100 ms.

We measure the maximum throughput of each data service by increasing the number of EBs until the service does not respect its SLA any more. To generate flexible reproducible workloads for each data service, we first ran the TPC-W benchmark several times under relatively low load (1000 EBs) and collected the logs of the invocation of data service interfaces. We obtained 72 query logs, each representing the workload of 1000 EBs for a duration of 30 minutes. We can thus generate any desired workload, from 1000 EBs to 72,000 EBs step by 1000 EBs, by replaying the right number of elementary log files across one or more client machines concurrently.

Figure 3.4 shows the throughput scalability of three representative data services from the scalable TPC-W. The *Item_basic* data service is read-only. It is therefore trivial to increase its throughput by adding database replicas. Similarly, the *Item_dynamic* service receives relatively few UDI queries, and can be scaled by simple master-slave replication.

On the other hand, the *Financial* service incurs many database transactions and UDI queries, which implies that simple database replication will not produce major throughput improvements. We see, however, that the implementation discussed in Section 3.3.1 exhibits a linear growth of its throughput as the number of database servers increases.

To conclude, we were able to scale all data services to a level where they could sustain a load of 50,000 EBs. Different services have different resource requirements to reach this level, with the *Item_basic*, *Item_dynamic* and *Financial* services requiring 3, 3, and 13 database servers, respectively.

We believe that all the data services can easily be scaled further. We stopped at that point as 50,000 EBs is the maximum throughput that our TPC-W implementation reaches when we use the entire DAS-3 cluster for hosting the complete application.

3.4.4 Scalability of the Entire TPC-W

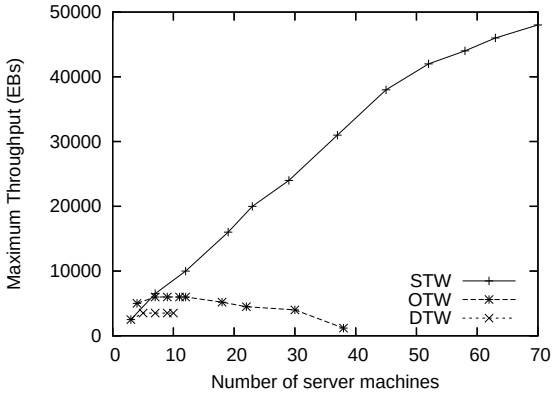
We conclude this performance evaluation by comparing the throughput scalability of the OTW, DTW and STW implementations of TPC-W. Similar to the previous experiment, we exercised each system configuration with increasing numbers of EBs until the SLA was violated. In this experiment, we use the application-level definition of the SLA as described in Section 3.4.2.

Figure 3.5(a) compares the scalability of OTW, DTW and STW when using between 2 and 70 server machines. In all cases we started by using one application server and one database server. We then added database server machines to the configurations. In OTW, extra database servers were added as replicas of the monolithic application state. In DTW, we start with one database server for all services, and eventually reach a configuration with one database server per service. In STW, we allocated the resources as depicted in Figure 3.5(b). Note that in all cases, we deliberately over-allocated the number of application servers and client machines to make sure that the performance bottleneck remains at the database servers.

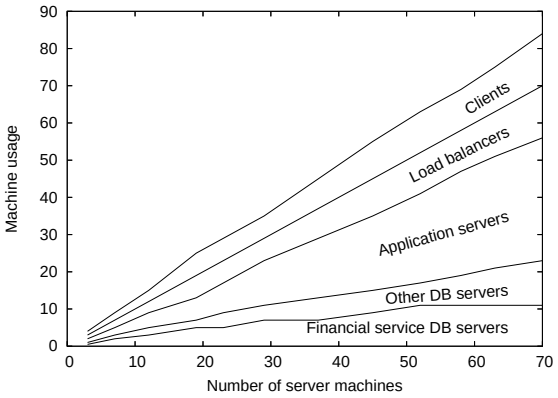
When using very few servers, OTW slightly outperforms DTW and STW. With increasing number of servers, OTW can be scaled up until about 6000 EBs when using 8 servers. However, when further adding servers, the throughput decreases. In this case, the performance improvement created by extra database replicas is counterbalanced by the extra costs that the master incurs to maintain consistency.

As no individual scaling techniques are applied to DTW, it can be scaled up to at most 8 database servers (one database server per service). The maximum throughput of DTW is around 3500 EBs. Note that this is only about half of the maximum achievable throughput of OTW. This is due to the extra costs brought by data denormalization, in particular the rewritten queries. Adding more database servers per service using database replication would not improve the throughput, as most of the workload is concentrated in the *Financial* service.

Finally, STW shows near linear scalability. It reaches a maximum throughput



(a) Maximum system throughput



(b) Allocation of machine resources for STW

Figure 3.5: Scalability of TPC-W hosting infrastructure

of 48,000 EBs when using 70 server machines (11 database servers for the *Financial* service, 12 database servers for the other services, 33 application servers and 14 load balancers). Taking into account the 14 client machines necessary to generate a sufficient workload, this configuration uses the entire DAS-3 cluster. The maximum throughput of STW at that point is approximately 8 times that of OTW, and 10 times that of a single database server.

We note that the STW throughput curve seems to start stabilizing around 50 server machines and 40,000 EBs. This is not a sign that we reached the maximum achievable throughput of STW. The explanation is that, as illustrated in Figure 3.4, 40,000 EBs is the point where many small services start violating their SLA with two database servers, and need a third database server. In our implementation each database server is used for a single service, which means that several extra database servers must be assigned to the small data services to move from 40,000 EBs to 50,000 EBs. We expect that using more resources the curve would grow faster again up to the point where the small data services need four servers.

3.5 Conclusion

Most approaches for building scalable Web applications consider the application code and data structure as constants, and propose middleware layers to improve performance transparently to the application. We take a different stand and demonstrate that major scalability improvements can be gained by allowing one to denormalize an application's data into independent services. While such restructuring introduces extra costs, it considerably simplifies the query access pattern that each service receives, and allows for a much more efficient use of classical scalability techniques. We applied this methodology to three standard benchmark applications and showed that it allows TPC-W, the most challenging of the three, to scale by at least an order of magnitude compared to master-slave database replication. Importantly, data denormalization does not imply any loss in terms of consistency or transactional properties. This aspect makes our approach unique compared to, for example, [42].

Data denormalization exploits the fact that an application's queries and transactions usually target few data columns. This, combined with classical database denormalization techniques such as query rewriting and column replication, allows us to cluster the data into disjoint data services. Although this property was verified in all applications that we examined, one cannot exclude the possible existence of applications with sufficient data overlap to prevent any service-oriented denormalization. This may be the case of transaction-intensive applications, whose ACID properties would impose very coarse-grained data clustering. It is a well-known fact that database transactions in a distributed environment imply important performance loss, so one should carefully ponder whether transactions are necessary or not.

The fact that denormalization is steered by prior knowledge of the application's query templates means that any update in the application code may require to restructure the data to accommodate new query templates. However, the fact that all data services resulting from denormalization have clear semantics makes us believe that extra application features could be implemented without the need to redefine data services and their semantics.

In our experience, designing the data denormalization of an application from its original data structure and query templates takes only a few hours. On the other hand, the work required for the actual implementation of the required changes largely depends on the complexity of each data service. For complex Web applications, re-implementation may require lot of time and development skills. This increases development times and restricts the practical usefulness of this technique. We therefore need a more practical approach where data restructuring is not necessary anymore. In the following chapters, we present such an approach using inherently-scalable NoSQL databases. In Chapter 4, we show how to support ACID transactions on top of NoSQL databases without compromising scalability. In Chapter 5, we further extend this design to support join queries.

Chapter 4

CloudTPS: Transactional Consistency in NoSQL Data Stores

We have seen that it is possible to build scalable Web applications using relational databases. However, this approach requires considerable manual efforts in restructuring applications. This chapter explores a different approach based on inherently-scalable NoSQL data stores. As discussed in Chapter 2, these data stores partition application data to provide incremental scalability, and replicate the partitioned data to tolerate server failures. These good properties have convinced many developers to rely on NoSQL data stores for building their Web applications.

The scalability and high availability properties of NoSQL data stores, however, come at a cost. First, they provide only weak consistency such as eventual data consistency: any data update becomes visible after a finite but undeterministic time. As weak as this consistency property may seem, it does allow to build a wide range of useful applications, as demonstrated by the commercial success of NoSQL data stores such as Amazon SimpleDB [6] and DynamoDB [5]. However, many other applications such as payment and online auction services cannot afford any data inconsistency. It is therefore essential to provide transactional data consistency to support the applications that need it. Second, NoSQL data stores allow data queries only by primary key rather than supporting secondary-key or join queries. This chapter focuses on the first issue and shows how to implement ACID transactions on top of NoSQL data stores while preserving scalability and fault tolerance. The next chapter will address the issue of supporting complex queries such as join queries and secondary-key queries.

A transaction is a set of queries that must be executed atomically on a single consistent view of a database. The main challenge to support transactional guarantees in a NoSQL data store is to provide the ACID properties of Atomicity, Consistency, Isolation and Durability [47] without compromising scalability properties. However, the underlying NoSQL data store may provide only eventual

consistency. We address this discrepancy by creating a temporary secondary copy of the application data in the transaction managers that handle consistency.

Obviously, any centralized transaction manager would face two scalability problems: 1) A single transaction manager must execute all incoming transactions and would eventually become the performance and availability bottleneck; 2) A single transaction manager must maintain a copy of all data accessed by transactions and would eventually run out of storage space. To support scalable transactions, we propose to split the transaction manager into any number of Local Transaction Managers (LTMs) and to partition the application data and the load of transaction processing across LTMs.

CloudTPS exploits three properties typical of Web applications to allow efficient and scalable operations. First, we observe that in Web applications, all transactions are short-lived because each transaction is encapsulated in the processing of a particular request from a user. This rules out long-lived transactions that make scalable transactional systems so difficult to design, even in medium-scale environments [106]. Second, Web applications tend to issue transactions that span a relatively small number of well-identified data items. This means that the commit protocol for any given transaction can be confined to a relatively small number of servers holding the accessed data items. It also implies a low (although not negligible) number of conflicts between multiple transactions concurrently trying to read/write the same data items. Third, many read-only queries of Web applications can produce useful results by accessing an older yet consistent version of data. This allows to execute complex read queries directly in the NoSQL data store, rather than in LTMs.

CloudTPS must maintain the ACID properties even in the case of server failures. For this, we replicate data items and transaction states to multiple LTMs, and periodically checkpoint consistent data snapshots to the NoSQL data store. Consistency correctness relies on the eventual consistency and high availability properties of NoSQL data stores: we need not worry about data loss or unavailability after a data update has been issued to the storage service.

It should be noted that the CAP theorem proves that one must trade off between strong Consistency and high Availability in the presence of network Partitions [44]. Typical NoSQL data stores relax strong consistency for high availability. As discussed in Section 2.5, these systems remain available even in the presence of network partitions, and may generate inconsistent copies of data. After the network is restored, these systems then reconcile inconsistent data updates and effectively implement some form of eventual data consistency [18]. However, these techniques heavily depend on the semantics of specific applications and demand significant efforts from programmers. In this chapter, we make the opposite choice and prefer providing transactional consistency for the applications that cannot afford any data inconsistency. We choose to always guarantee transactional consistency, possibly at the cost of unavailability during network failures. This allows programmers to manage data consistency without having to worry about subtle consistency issues.

To implement CloudTPS efficiently, we must address two additional issues.

Firstly, there exists a wide variety of NoSQL data stores [6, 27, 29, 69]. CloudTPS should be portable across them, and the porting should require only minor adaptations. On the one hand, as current NoSQL data stores use different data models and interfaces, we build CloudTPS upon their common features: our data model is based on key-value pairs. The implementation demands only a simple primary-key-based get/put interface from NoSQL data stores. On the other hand, current NoSQL data stores provide different consistency levels. For instance, Bigtable supports transactions on single data items while SimpleDB provides either eventual consistency or single-item transactions. To ensure the correctness and efficiency of CloudTPS, we implement various mechanisms for different underlying consistency levels.

Secondly, loading a full copy of application data into the system may overflow the memory of LTMs, forcing one to use many LTMs just for their storage capacity. This is, however, not necessary as only the currently accessed data items contribute to maintaining ACID properties. Other unaccessed data items can be evicted from the LTMs if we can fetch their latest stored versions from the NoSQL data store. Web applications exhibit temporal locality where only a portion of application data is accessed at any time [108, 111]. We can therefore design efficient memory management mechanisms to restrict the number of in-memory data items in LTMs while maintaining strong data consistency. Data items being accessed by uncommitted transactions must stay in the LTMs to maintain ACID properties; others depend on a trade-off between memory size and access latency. We use a cost-aware replacement policy to dictate which data items should remain in the LTMs.

We demonstrate the scalability of our transactional database service using a prototype implementation[1]. Following the data models of Bigtable and SimpleDB, transactions are allowed to access any number of data items by primary key at the granularity of the data row. CloudTPS supports both read-write and read-only transactions. In this chapter, we focus on transactional properties with no support for complex queries. The list of primary keys accessed by a transaction must be given explicitly before executing the transaction. In the next chapter, we will return to this issue and show how CloudTPS can support complex queries which require identifying the accessed data items on the fly.

We evaluate our prototype under a workload derived from the TPC-W e-commerce benchmark [67]. We implemented CloudTPS on top of two different scalable data layers: HBase, an open-source clone of Bigtable [52], running in our local cluster; and SimpleDB, running in the Amazon cloud [7]. We show that CloudTPS scales linearly to at least 40 LTMs in our local cluster and 80 LTMs in the Amazon cloud. This means that any increase in workload can be accommodated by provisioning more servers. CloudTPS tolerates server failures, which only cause a few aborted transactions (authorized by the ACID properties) and a temporary drop of throughput during transaction recovery and data reorganization. In

[1] Our prototype is available at `http://www.globule.org/cloudtps`.

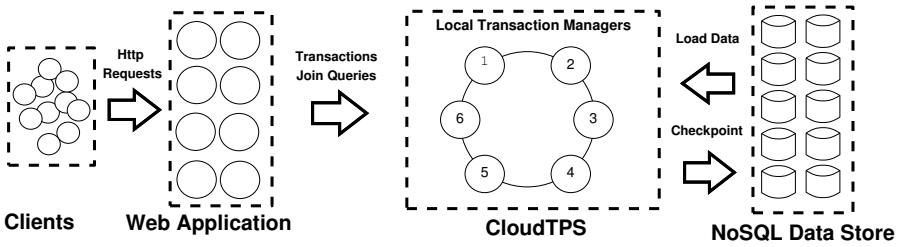


Figure 4.1: CloudTPS system organization.

case of network partitions, CloudTPS may reject incoming transactions to maintain data consistency. It recovers and becomes available again as soon as the network is restored, while still maintaining ACID properties. We finally evaluate our memory management mechanism and show that it can effectively control the buffer sizes of LTMs and only cause minor performance overhead.

This chapter is organized as follows. Sections 4.1 and 4.2 describe the system model and the system design respectively. Section 4.3 discusses implementation details and two optimizations for memory management and read-only transactions. Section 4.4 presents performance evaluations. Finally, Section 4.5 concludes this chapter.

4.1 System Model

Figure 4.1 shows the organization of CloudTPS. Clients issue HTTP requests to a Web application, which in turn issues transactions to a Transaction Processing System (TPS). The TPS is composed of any number of LTMs, each of which is responsible for a subset of all data items. The Web application can submit a transaction to any LTM that is responsible for one of the accessed data items. This LTM then acts as the coordinator of the transaction across all LTMs in charge of the data items accessed by the transaction. The LTMs operate on an in-memory copy of the data items loaded from the NoSQL data store. Data updates resulting from transactions are kept in memory of the LTMs. To prevent data loss due to LTM server failures, the data updates are replicated to multiple LTM servers. LTMs also periodically checkpoint the updates back to the NoSQL data store which is assumed to be highly available and persistent.

We implement transactions using the two-phase commit protocol (2PC). In the first phase, the coordinator requests all involved LTMs and asks them to check that the operation can indeed been executed correctly. If all LTMs vote favorably, then the second phase actually commits the transaction. Otherwise, the transaction is aborted.

CloudTPS transactions are short-lived and access only well-identified data items. CloudTPS allows only server-side transactions implemented as predefined procedures stored at all LTMs. Each transaction contains one or more subtransac-

```
public abstract class SubTransaction {
    //Input by Web Application
    public String className;
    public String tableName;
    public String primaryKey;
    public Hashtable<String,String> parameters;
    //Input by LTMs
    public Hashtable<String,String> dataItem;
    //Output
    public String[][] dataToReturn;
    public String[][] dataToPut;
    public String[] dataToDelete;
    //Data Operations of the SubTransaction
    public VoteResult run();
}
```

Figure 4.2: The parent class of all subtransactions classes.

tions, which operate on a single data item each. The application must provide the primary keys of all accessed data items when it issues a transaction.

Concretely, a transaction is implemented as a Java object containing a list of subtransaction instances. All subtransactions are implemented as subclasses of the `SubTransaction` abstract Java class. As shown in Figure 4.2, each subtransaction contains a unique class name to identify itself, a table name and primary key to identify the accessed data item, and input parameters organized as attribute-value pairs. Each subtransaction implements its own data operations by overriding the `run()` operation. The return value of the `run()` operation specifies whether this subtransaction is able to commit. The execution of `run()` also generates the data updates and the results for read data operations, which are stored in the `Output` attributes.

The bytecode of all subtransactions is deployed at all LTMs beforehand. A Web application issues a transaction by submitting the names of included subtransactions and their parameters. LTMs then construct the corresponding subtransaction instances to execute the transaction. In the first phase of 2PC, LTMs load the data items of each subtransaction and execute the `run()` operation to decide on their votes and generate proposed data updates. If an agreement to commit is reached, LTMs apply the updates.

We assign data items to LTMs using consistent hashing [57]. To achieve a balanced assignment, we first cluster data items into virtual nodes, and then assign virtual nodes to LTMs. As shown in Figure 4.1, multiple virtual nodes can be assigned to the same LTM. To tolerate LTM failures, virtual nodes and transaction states are replicated to one or more LTMs. After an LTM server failure, the latest updates can then be recovered and affected transactions can continue execution while satisfying ACID properties.

4.2 System Design

We now detail the design of the TPS to guarantee the atomicity, consistency, isolation and durability properties. Each of the properties is discussed individually. We then discuss the membership mechanisms to guarantee the ACID properties even in case of LTM failures and network partitions.

4.2.1 Atomicity

The atomicity property requires that either all operations of a transaction complete successfully, or none of them does. To ensure atomicity, for each transaction issued, CloudTPS performs two-phase commit (2PC) across all the LTMs responsible for the data items accessed. As soon as an agreement to commit is reached, the transaction coordinator can simultaneously return the result to the Web application and complete the second phase [54].

To ensure atomicity in the presence of server failures, all transaction states and data items should be replicated to one or more LTMs. LTMs replicate the data items to the backup LTMs during the second phase of a transaction. Thus when the second phase completes successfully, all replicas of the accessed data items are consistent. The transaction state includes the transaction timestamp (discussed in Section 4.2.3), the agreement to commit, and the list of data updates to be committed.

When an LTM fails, the transactions it was coordinating can be in two states. If a transaction has reached an agreement to commit, then it must eventually be committed; otherwise, the transaction can still be aborted. Therefore, we replicate transaction states on two occasions: 1) When an LTM receives a new transaction, it must replicate the transaction state to other LTMs before confirming to the application that the transaction has been successfully submitted; 2) After all participant LTMs reach an agreement to commit at the coordinator, the coordinator updates the transaction state at its backups with the agreement to commit and all the data updates. The participant LTMs piggyback their data updates with their vote messages. This creates in essence in-memory "redo logs" at the backup LTMs. The coordinator must finish this step before carrying out the second phase of the commit protocol. If the coordinator fails after this step, the backup LTMs can then commit the transaction. Otherwise, it can simply abort the transaction without violating the ACID properties.

An LTM server failure also results in the inaccessibility of the data items it was responsible for. It is necessary to re-replicate these data items to maintain N backups. If a second LTM server failure happens during the recovery process of a previous LTM server failure, the system initiates the recovery of the second failure after the first recovery process has completed. The transactions that cannot recover from the first failure because they also accessed the second failed LTM are left untouched until the second recovery process.

As each transaction and data item has $N + 1$ replicas in total, the TPS can thus guarantee the atomicity property under the simultaneous failure of N LTM servers.

4.2.2 Consistency

The consistency property requires that a transaction, which executes on a database that is internally consistent, will leave the database in an internally consistent state. Consistency is typically expressed as a set of declarative integrity constraints. We assume that the consistency rule is applied within the logic of transactions. Therefore, the consistency property is satisfied as long as all transactions are executed correctly.

4.2.3 Isolation

The isolation property requires that the behavior of a transaction is not disturbed by the presence of other transactions that may be accessing the same data items concurrently. The TPS decomposes a transaction into a number of subtransactions, each accessing a single data item. Thus the isolation property requires that if two transactions conflict on any number of data items, all their conflicting subtransactions must be executed sequentially, even though the subtransactions are executed in multiple LTMs.

We apply timestamp ordering for globally ordering conflicting transactions across all LTMs. Each transaction has an globally unique timestamp among all of its conflicting transactions. All LTMs then order transactions as follows: a subtransaction can execute only after all conflicting subtransactions with a lower timestamp have committed. It may happen that a transaction is delayed (e.g., because of network delays) and that a conflicting subtransaction with a younger timestamp has already committed. In this case, the older transaction will abort, obtain a new timestamp and restart the execution of all of its subtransactions.

As each subtransaction accesses only one data item by primary key, the implementation is straightforward. Each LTM maintains a list of subtransactions for each data item it handles. The list is ordered by timestamp so LTMs can execute the subtransactions sequentially in timestamp order. The exception discussed before happens when an LTM inserts a subtransaction into the list but finds its timestamp smaller than the one currently being executed. It then reports the exception to the coordinator LTM of this transaction so that the whole transaction can be restarted. We extended the 2PC with an optional restart phase, which is triggered if any of the subtransactions reports an ordering exception. After a transaction reaches an agreement and enters the second phase of 2PC, it cannot be restarted any more.

We are well aware that assigning timestamps to transactions using a single global timestamp manager can create a potential bottleneck and a single point of failure in the system. A simpler, fully decentralized solution consists of letting each LTM generate timestamps using its own local clock, coupled with the LTM's ID to enforce a total order between timestamps. Note that this solution does not

require perfectly synchronized clocks for ensuring correctness. Transactions do not necessarily need to be timestamped according to their real-time submission order, but there must simply be a total order between any pair of transactions. However, if one LTM clock significantly lags behind the others, then the transactions submitted at this LTM will have a higher chance of being restarted than others. The DAS-3 cluster does not use NTP synchronization between its nodes, which is the reason why we used a centralized timestamp manager in our experiments.

4.2.4 Durability

The durability property requires that the effects of committed transactions cannot be undone and would survive server failures. In our case, it means that all the data updates of committed transactions must be successfully written back to the backend NoSQL data store.

The main issue here is to support LTM failures without losing data. For performance reasons, the commit of a transaction does not directly update data in the NoSQL data store but only updates the in-memory copy of data items in the LTMs. Instead, each LTM issues periodic updates to the NoSQL data store. During the time between a transaction commit and the next checkpoint, durability is ensured by the replication of data items across several LTMs. After checkpoint, we can rely on the high availability and eventual consistency properties of the NoSQL data store for durability.

When an LTM server fails, all the data items stored in its memory that were not checkpointed yet are lost. However, as discussed in Section 4.2.1, all data items of the failed LTM can be recovered from the backup LTMs. The difficulty here is that the backups do not know which data items have already been checkpointed. One solution would be to checkpoint all recovered data items. However, this can cause a lot of unnecessary writes. One optimization is to record the latest checkpointed transaction timestamp of each data item and replicate these timestamps to the backup LTMs. We further cluster transactions into groups, then replicate timestamps only after a whole group of transactions has completed.

Another issue related to checkpointing is to avoid degrading the system performance at the time of a checkpoint. The checkpoint process must iterate through the latest updates of committed transactions and select the data items to be checkpointed. A naive implementation that would lock the whole buffer during checkpointing would also block the concurrent execution of transactions. We address this problem by maintaining an extra buffer in memory with the list of data items to be checkpointed. Transactions write to this buffer by sending updates to an unbounded non-blocking concurrent queue [68]. This data structure has the property of allowing multiple threads to write concurrently to the queue without blocking each other. Moreover, it orders elements in FIFO order, so old updates will not override younger ones.

4.2.5 Membership

To correctly execute transactions, all LTMs must share the same view of system membership to determine the assignment of data items consistently. The system membership changes when LTMs join, leave, fail or recover from failures. These events may happen at any time, including during the execution of transactions. To ensure the ACID properties, changes in system membership must not take place during the 2PC execution of any transaction. When an LTM fails, other LTMs must therefore first complete the recovery of all ongoing transactions before updating the system membership.

In addition to LTM failures, the system may also encounter network failures, which can temporarily split the LTMs into multiple disconnected partitions. In such a case, we choose to always guarantee consistency at the possible cost of a loss of availability. In the case of system partitioning, transactions may still proceed provided that: (i) one of the partitions is able to elect itself as the majority partition; and (ii) its LTMs can recover the consistent states of all data items. In all other cases the system will reject incoming transactions until it fulfills the condition again.

This section presents our mechanism to recover the system consistently from network partitions.

Membership Updates

To ensure consistent membership, all membership changes are realized through a 2PC across all available LTMs. All LTMs block incoming transactions until the new system membership has been committed consistently. In the first phase of a membership change, each LTM waits for all of its coordinated ongoing transactions to terminate, and then votes to commit. After reaching an agreement to commit, the second phase updates the system membership and applies the new data assignment through data item replication/relocation.

Each membership change creates a new membership version attached to a monotonically increasing timestamp. Each LTM attaches the timestamp of its current membership to all of its messages. If an LTM receives a message with a higher timestamp than its own, this means that the other LTMs consider it as having failed. The concerned LTM discards its entire state and rejoins.

After each membership change, the new timestamp is stored in a special membership table in the NoSQL data store. By scanning through this membership table, any new LTM or any Web application instance can locate the currently available LTMs. One issue is that the NoSQL data store may return a stale membership. However, one can contact the TPS and obtain the latest membership as long as the stale membership contains at least one LTM currently in the TPS.

Any LTM may initiate a membership update if it wants to join the system or it detects the unavailability of other LTMs. This means that multiple membership updates may be issued simultaneously. To guarantee the isolation of such updates,

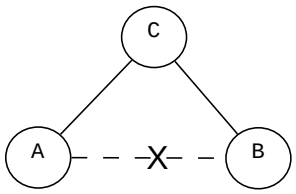


Figure 4.3: An example of unclean network partition.

we use a simple optimistic concurrency control mechanism so that only one membership update can take place at a time [89]. If an LTM receives a new request for a membership update before a previous one has finished, then this LTM will vote to abort this new request. To avoid continuous conflicts and aborts, LTMs may insert a random time delay before re-initiating the aborted membership update.

Dealing with Network Partitions

In case of a network partition, multiple system subsets may consider that the other unreachable LTMs have failed, recover from their "failures" and carry on with processing the application workload independently from each other. However, this would violate the ACID properties and must therefore be avoided.

For simplicity, we assume that no network partition occurs during the recovery of another partition. Supporting this latter case requires additional algorithms that we consider out of the scope of the thesis.

We use the Accessible Copies algorithm [39] to recover the system consistently during network partitions. This algorithm ensures that only one partition may access a given data item by allowing access to a given data item only within a partition that contains a majority of replicas. Instead of using a majority partition for each data item, we adapt the majority rule such that only the partition that contains more than half of the previous membership can access all data items. Minority partitions are forbidden access to any data item. It may happen that the majority partition lacks more than N LTM servers from the previous membership[2], and thus cannot recover all data items; in this case it rejects all incoming transactions until it can recover all data items.

Once a majority partition is established, it can recover all the ongoing transactions and accept new incoming transactions. After recovery, all LTMs in the majority partition have the new system membership with an increased timestamp. The other ones, which still have the previous membership timestamp, can detect after network partition recovery that they have been excluded from the membership, and rejoin as new members.

When an LTM discovers that other LTMs are unreachable because of LTM crashes and/or network partitions, it identifies its new partition membership through

[2]Assuming that each transaction and data item has $N + 1$ replicas.

a 2PC across all LTMs. In the first phase, it sends an invitation to all LTMs; any responding LTM which votes commit belongs to its partition membership. After all LTMs either respond or time-out, the second phase updates the partition membership of all LTMs in the partition of the coordinator. One optimization is to exclude the discovered unreachable LTMs from the first 2PC of building partition membership. This optimization is effective for the scenario of LTM failures, avoiding a possible delay of time-out in waiting for responses from these failed LTMs. In case of network partitions, the first 2PC may fail to establish a majority partition to recover the system. LTMs should then include these excluded LTMs back into the following periodic 2PCs building partition membership.

The above mechanism can organize the TPS into a number of disjoint partitions, provided that the network is "cleanly" partitioned: any two LTMs in the same partition can communicate, and any two LTMs in different partitions cannot. However, a network may also be "uncleanly" partitioned due to the lag of reconstructing routing tables. Figure 4.3 shows an example of an unclean partition where each LTM has a different view of reachable LTMs: view(A) = {A,C}, view(B) = {B,C} and view(C) = {A,B,C}. In this case, LTM C may join two different partitions: either {A,C} or {B,C}, which both turn out to be majority partitions. To ensure that an LTM can belong to only one partition at a time, we define that if an LTM has already joined a partition, it will vote abort to any invitation of joining a different partition.

Minority partitions periodically try to rejoin the system by checking if previously unavailable nodes become reachable again. Receiving an abort vote for an invitation indicates that partitions are reconnected. In this case, the two partitions can be merged through a 2PC across all LTMs in the two partitions. The first phase is to push the memberships of two partitions to all the participant LTMs. A participant LTM votes to commit if the received membership matches its current latest partition membership. Otherwise, it votes to abort. If an agreement to commit is reached, the second phase updates the partition membership of all participant LTMs into the combined membership of two partitions. If any participant LTM votes to abort or fails to respond, the 2PC is aborted.

4.3 System Implementation

This section discusses implementation details of CloudTPS, in particular how to support various backend NoSQL data store. We also present two optional optimizations: memory management to prevent memory overflow in the LTMs, and handling of read-only transactions containing complex read queries.

4.3.1 Portability

CloudTPS relies on a NoSQL data store to ensure transaction durability. However, current NoSQL data stores support different data models, consistency guar-

Table 4.1: Key differences between cloud data services

	SimpleDB	**Bigtable**	**PNUTS**
Data Item	Multi-value attribute	Multi-version with timestamp	Multi-version with timestamp
Schema	No schema	Column families	Explicitly claimed attributes
Operation	Range queries on arbitrary attributes of a table	Single-table scan with various filtering conditions	Single-table scan with predicates
Consistency	Eventual consistency	Single-row transaction	Single-row transaction

antees, operation semantics and interfaces. Adapting CloudTPS to all of them is a challenge. We compare three prominent and typical NoSQL data stores: Amazon SimpleDB, Google Bigtable and Yahoo PNUTS. Our implementation is compatible with SimpleDB and Bigtable. Porting CloudTPS to other NoSQL data stores requires only minor adaptations.

SimpleDB, Bigtable and PNUTS have a number of similarities in their data models. They all organize application data into tables. A table is structured as a collection of data items with unique primary keys. The data items are described by attribute-value pairs. All attribute values are typed as strings. Data items in the same table can have different attributes. Data items are accessed with get/put by primary key. Operations across tables, such as join queries, are not supported.

On the other hand, as shown in Table 4.1, the three NoSQL data stores also have some key differences:

First, SimpleDB supports multiple values per attribute of a data item, while Bigtable and PNUTS only allow one. To be compatible with all of them, our data model allows only one value per attribute.

Second, SimpleDB does not impose a predefined schema for its tables. PNUTS requires explicit claims of all attributes in a table, but it is still compatible with SimpleDB, as it does not require all records to have values for all claimed attributes and new attributes can be added at any time without halting query or update activity. On the other hand, Bigtable groups attributes into predefined column families. To access an attribute, one must include its column family name as its prefix. We address this difference by always prepending attribute names with the column family name for Bigtable.

Third, all three NoSQL data stores support sophisticated data-access operations within a table, but via different APIs. SimpleDB supports range queries inside a table with its specific language; Bigtable and PNUTS provide similar functionality with table scanning using various filtering conditions or predicates. This difference

is irrelevant to the system design described before, as it accesses data items only by primary key. However, the optimization of read-only transactions, described in Section 4.3.3, allows Web applications to access consistent data snapshots in NoSQL data stores directly via their APIs. Therefore, the implementation of this optimization depends on the interface of the underlying NoSQL data stores.

Finally, SimpleDB provides eventual consistency by default so that applications may read stale data. In contrast, Bigtable and PNUTS support single-row transactions, so they can guarantee returning the latest updates. We assume that when CloudTPS starts and loads a data item from the NoSQL data store for the first time, all the replicas of this data item are consistent. So CloudTPS can obtain the latest updates in this case, regardless of the consistency level of underlying NoSQL data store. However, this is not true for reloading a data item that has been recently updated. Different data consistency models of NoSQL data stores require additional adaptations to implement our performance optimizations, as discussed in the following sections.

4.3.2 Memory Management

For efficiency reasons we keep all data in the main memory of the LTMs. However, maintaining a full copy of all application data may overflow the memory space, if the size of the data is large. One would thus have to allocate unnecessary LTM servers just for their memory space, rather than for their contributions to performance improvement. On the other hand, we notice that Web applications exhibit temporal data locality so that only a small portion of application data is accessed at any time [111, 108]. Keeping unused data in the LTMs is not necessary for maintaining ACID properties, so LTMs can evict these data items in case of memory shortage, and reload them from the NoSQL data store when necessary.

The key issue is that the eviction of any data items from LTMs must not violate the ACID properties of transactions. Obviously, the data items that are currently accessed by ongoing transactions must not be evicted until the transaction completes and the data updates have been checkpointed. After evicting a data item from the LTMs, future transactions may require it again. To guarantee strong consistency for these transactions, LTMs have to guarantee that the latest version of the evicted data items can be obtained from the NoSQL data store in the next read. The solution to this issue, however, depends on the consistency level guaranteed by the underlying NoSQL data store. To ensure that the latest version of a data item is visible, CloudTPS requires that the underlying NoSQL data store supports at least monotonic-reads consistency [104]. If the data service provides the read-your-writes consistency, checkpointing back the latest updates successfully is sufficient to be able to evict a data item. For instance, Bigtable and PNUTS support single-row transactions and thus provide read-your-writes consistency. If the data service provides only eventual consistency, such as in SimpleDB, then LTMs may still obtain stale data even after a get returned the latest version. To address this problem, we store the timestamps of the latest versions of all data items in LTMs,

which can then determine if the newly loaded version of data item is up-to-date. If it is not, LTMs will abort the transactions and maintain ACID properties at the cost of rejecting these transactions.

Storing the latest timestamps of all data items in memory may also overflow the memory if the number of data items is extremely large. Storing them in the NoSQL data store is not an option, since they must maintain strong consistency. A simple solution could be to store them in the local hard drive of the LTM.

Another difficulty is that SimpleDB does not support multi-versions with time stamp, but only multi-values for an attribute. We address this by attaching a timestamp at the end of the value of each attribute and so transform multi-values into multi-versions.

To minimize the performance overhead of memory management, we must maximize the hit rate of transactions in LTMs and thus carefully select which data items should be evicted. Standard cache replacement algorithms, such as LRU, assume that all data items have identical sizes. However, in CloudTPS, data items can have very different sizes, leading to poor performance. Instead we adopt the cost-aware GreedyDual-Size (GDS) algorithm [21], which leverages knowledge of data item sizes to select data items to evict. The GDS algorithm associates a value H to each data item p: $H(p) = L + cost/size$, where L is the H value of the latest evicted data item. We set the $cost$ parameter to 1 for all data items as this optimizes hit rate. The parameter $size$ refers to the size of data item p. Each time an LTM needs to replace a data item, it selects the data item with the lowest H value and updates its L value to the H value of this evicted data item. When a data item is accessed, the H value of this data item is recalculated with the updated parameters: the latest L value and its possibly changed $size$.

4.3.3 Read-Only Transactions

CloudTPS supports read-write and read-only transactions indifferently. The only difference is that in read-only transactions no data item is updated during the second phase of 2PC. Read-only transactions have the same strong data consistency property as read-write transactions, but also the same constraint: accessing well-identified data items by primary key only. However, CloudTPS provides an additional feature to support complex read-only transactions containing for example range queries.

We exploit the fact that many read queries can produce useful results by accessing a consistent but possibly stale data snapshot. For example, in e-commerce Web applications, a promotion service may identify the best seller items by aggregating recent orders information. However, it may not be necessary to compute the result based on the absolute most recent orders. We therefore introduce the concept of Weakly-Consistent Read-only Transaction (WCRT): A WCRT contains any number of read operations offered by the NoSQL data store, such as table scans for Bigtable. Web applications issue WCRTs directly to the NoSQL data store, by-

passing the LTMs. All read operations of a WCRT executes on the same internally consistent but possibly slightly outdated snapshot of the database.

To implement WCRTs, we introduce a snapshot mechanism in the checkpoint process of LTMs, which marks each data update with a specific snapshot ID that is monotonically increasing. This ID is used as the version number of the newly created version when it is written to the NoSQL data store. A WCRT can thus access a specific snapshot by only reading the latest version of any data item of which the timestamp is not larger than the snapshot ID.

We group transactions in sets of M consecutive transactions. Each set constitutes a new snapshot. Assuming that the transaction timestamp is implemented as a simple counter, the first snapshot reflects all the updates of committed transactions $[0, M)$. The next snapshot reflects updates from transactions $[0, 2M)$, and so on. At the finest granularity, with $M = 1$, each read-write transaction creates a new snapshot.

The key issue in this snapshot mechanism is to determine whether a consistent snapshot is fully available in the NoSQL data store such that WCRTs can execute on it. A consistent snapshot contains all the updates of the transactions which it reflects. It is fully available only after all these updates have been checkpointed back. The main difficulty is that a transaction may update data items across multiple LTMs, where each LTM performs checkpoints for its own data items independently from the others. Therefore, CloudTPS must collect checkpoint progress information from multiple LTMs. To address this issue, we use the NoSQL data service as a shared medium for collecting checkpoint progress information. The system creates an extra table named `checkpoint`, where each LTM writes its latest completed snapshot ID into a separate data item using its membership ID as the primary key value. So the minimal snapshot ID stored in the `checkpoint` table represents the latest snapshot of which the updates are all checkpointed.

Even though all the updates of a snapshot have been checkpointed successfully, the availability of this snapshot still depends on the consistency level provided by the NoSQL data store. The data services must provide at least monotonic-reads consistency, so that LTMs can verify the visibility of the updates before claiming the snapshot is available. Bigtable and PNUTS support single-row transactions and thus provide the read-your-writes consistency. Therefore, the snapshot is immediately available after writing all checkpoints back. Lastly, if the NoSQL data store supports only eventual consistency, it is impossible to guarantee the visibility of certain writes in the next read so this feature is not supported.

4.4 Evaluation

We demonstrate the scalability of CloudTPS by presenting the performance evaluation of a prototype implementation on top of two different families of scalable data layers: HBase running in our local DAS-3 cluster [52] and SimpleDB running in the Amazon cloud. We also show that CloudTPS can recover from LTM failures

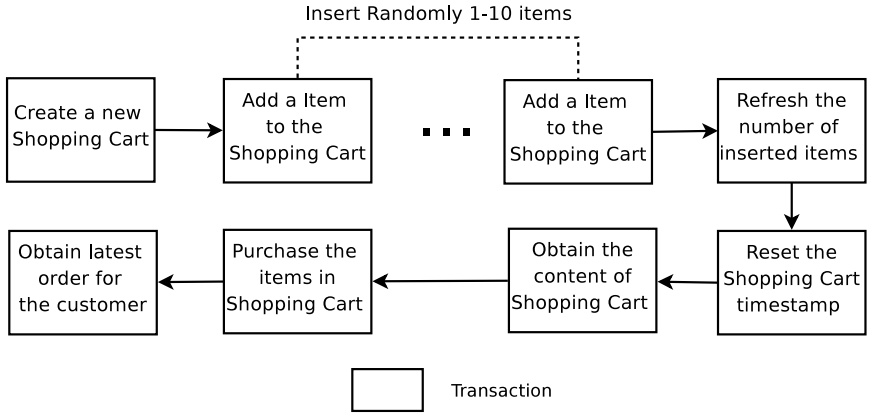


Figure 4.4: Transactions of TPC-W.

and network partitions efficiently by presenting the throughput of CloudTPS under these failures. Lastly, we demonstrate the effectiveness of the memory management mechanism and discuss the trade-off between system performance and buffer sizes.

We evaluate CloudTPS under a workload derived from TPC-W [67], an industry standard e-commerce benchmark that models an online bookstore similar to Amazon.com.

4.4.1 Migration of TPC-W to the cloud

TPC-W was originally designed as a Web application using a SQL-based relational database as backend. However, in this chapter, we focus on transactional guarantees for NoSQL data stores with no specific support for complex queries. We therefore need to adapt the original relational data model of TPC-W into the data models of Bigtable and SimpleDB, such that complex queries can be transformed into primary-key queries. We return to this topic in the next chapter where we will see how to host TPC-W on CloudTPS with no need for schema adaptation.

As described in Section 4.3.1, we can easily adapt the Bigtable data model into SimpleDB data model by using the exact same attribute names, which are prepended with the column family names. Therefore, we first adapt the relational data model of TPC-W into the Bigtable data model.

Using similar data denormalization techniques as in Chapter 3, we designed a Bigtable data model for TPC-W that contains the data accessed by the transactions in Figure 4.4. The relational data model of TPC-W comprises six tables that are accessed by these transactions. To adapt this data model to Bigtable, we first combine five tables (`Orders`, `Order_Line`, `Shopping_Cart`, `Shopping_Cart_Entry`, `CC_XACTS`) into one bigtable named `Shopping`. Each of the original tables is stored as a column family. The new bigtable `Shopping` has the same primary key as table `Shopping_Cart`. For table `Order_Line`, multiple rows

are related to one row in table `Order`, they are combined into one row and stored in the new bigtable by defining different column names for the values of same data column but different rows. Second, for the remaining table `Item`, only the column `i_stock` is accessed. We can thus have a bigtable named `Item_Stock` which contains only this column and has the same primary key. Finally, for the last transaction in Figure 4.4, which retrieves the latest order information for a specific customer, we create an extra index bigtable `Latest_Order` which uses customer IDs as its primary key and contains one column storing the latest order ID of the customer.

For both HBase and SimpleDB, we populate 144,000 customer records in the `Latest_Order` bigtable and 10,000 item records in the `Item_Stock` bigtable. We then populate the `Shopping` bigtable according to the benchmark requirements. As shown in Figure 4.4, the workload continuously creates new shopping carts. Thus, the size of the `Shopping` bigtable increases continuously during the evaluation, while the other two bigtables remain constant in size. In the memory management evaluation, we also measure the performance of 1 million records in the `Item_Stock` bigtable.

In the performance evaluation based on HBase, we observed a load balancing problem. TPC-W assigns new shopping cart IDs sequentially. However, each HBase node is responsible for a set of contiguous ranges of ID values, so at any moment of time, most newly created shopping carts would be handled by the same HBase node. To address this problem, we horizontally partitioned the bigtables into 50 subtables and allocated data items to subtables in round-robin fashion.

To port TPC-W to SimpleDB, we organize application data into a number of domains (i.e., tables), but each domain can only sustain a limited update workload. So we also have to horizontally partition a table in round-robin fashion and place each partition into a domain. Different from HBase, we can use at most 100 domains for the whole application. We therefore partition the three tables into a different number of subtables according to our estimated data-access loads. We horizontally partition the `Shopping` bigtable into 80 domains and the other two bigtables into 5 domains each. This way SimpleDB can provide sufficient capacity for both writes and reads, while CloudTPS remains the performance bottleneck for performance evaluation.

4.4.2 Experiment Setup

We perform evaluations on top of two scalable data layers: 1) HBase v0.2.1 [52] running in the DAS-3 cluster [34]; and 2) SimpleDB in the Amazon cloud [7]. We use Tomcat v5.5.20 as application server. The LTMs and load generators are deployed in separate application servers.

The evaluations from this chapter were realized using CloudTPS v0.1. Note that the next chapter uses CloudTPS v0.2, which features major performance improvement compared to version 0.1. The details of the differences between these two versions will be discussed in Section 5.3.

DAS-3 is an 85-node Linux-based server cluster. Each node has a dual-CPU / dual-core 2.4 GHz AMD Opteron DP 280, 4 GB of memory and a 250 GB IDE hard drive. Nodes are interconnected through a Gigabit LAN.

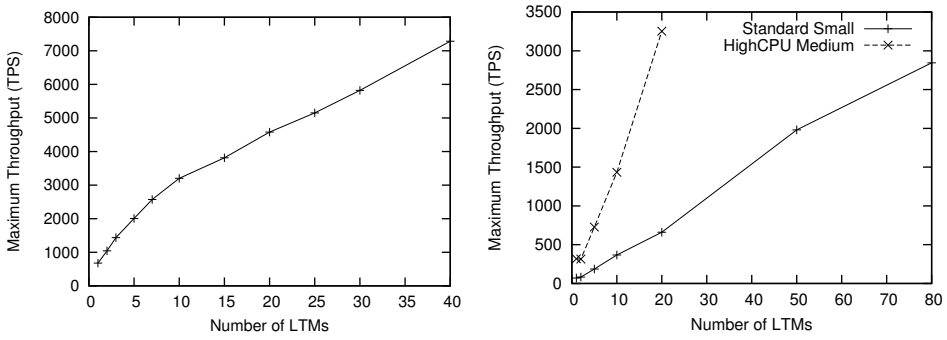
Amazon EC2 offers various types of virtual machine instances. We perform our evaluations with Small Instances in the Standard family (with 1.7 GB memory, 1 virtual core with 1 EC2 Compute Unit, and 160 GB storage) as well as Medium Instances in the High-CPU family (with 1.7 GB of memory, 2 virtual cores with 2.5 EC2 Compute Units each, and 350 GB of storage). At the time of our experiment, Standard Small instances cost $0.10 per instance-hour while High-CPU Medium instances cost $0.20 per instance-hour. One EC2 Compute Unit provides the CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor.

TPC-W workload is generated by a configurable number of Emulated Browsers (EBs) which issue requests from one simulated user. Our evaluations assume that the application load remains roughly constant. The workload that an Emulated Browser issues to the TPS mainly consists of read-write transactions that require strong data consistency. Figure 4.4 shows the workflow of transactions issued by an Emulated Browser, which simulates a typical customer shopping process. Each EB waits for 500 milliseconds on average between receiving a response and issuing the next transaction.

4.4.3 Scalability Evaluation

We study the scalability of CloudTPS in terms of maximum sustainable throughput under a response time constraint. In DAS-3, we assign one physical machine for each LTM, and have low contention on other resources such as network. Therefore, for the evaluations in DAS-3, we define a demanding response time constraint that imposes that 99% of the transactions must return within 100 ms. On the other hand, in the public Amazon cloud, our LTMs have to share a physical machine with other instances, and we have less control over the resources such as CPU, memory, network, etc. Furthermore, even multiple instances of the exact same type may exhibit different performance behavior [36]. Therefore, to prevent these interferences from disturbing our evaluation results, we relax the response time constraint for the evaluations in the Amazon cloud: 90% of the transactions must return within 100 ms.

We perform the scalability evaluation by measuring the maximum sustainable throughput of the system consisting of a given number of LTMs before the constraint gets violated. In DAS-3, we start with one LTM and 5 HBase servers, then add more LTM and HBase servers. We carry out each round of the experiment for 30 minutes to measure the performance of the system under a certain number of EBs. In all cases, we deliberately over-allocate the number of HBase servers and client machines to make sure that CloudTPS remains the performance bottleneck. We perform similar steps in the Amazon cloud. CloudTPS remains the performance bottleneck, as SimpleDB can provide sufficient capacity for both writes and



(a) HBase + DAS-3. (b) SimpleDB + Amazon EC2.

Figure 4.5: Scalability of CloudTPS under a response time constraint.

reads. We configure the system so that each transaction and data item has one backup in total, and set the checkpoint interval to 1 second.

Figure 4.5(a) shows that CloudTPS scales nearly linearly in DAS-3. When using 40 LTM servers it reaches a maximum throughput of 7286 transactions per second generated by 3825 emulated browsers. In this last configuration, we use 40 LTM servers, 36 HBase servers, 3 clients to generate load, and 1 global timestamp server. This configuration uses the entire DAS-3 cluster so we could not extend the experiment further. The maximum throughput of the system at that point is approximately 10 times that of a single LTM server.

Figure 4.5(b) shows the scalability evaluation in the Amazon cloud. Here as well, CloudTPS scales nearly linearly with both types of EC2 virtual instances. When using 80 Standard Small instances, CloudTPS reaches a maximum throughput of 2844 transactions per second generated by 1600 emulated browsers. The maximum throughput of the system at that point is approximately 40 times that of a single LTM server. When using 20 High-CPU Medium instances, CloudTPS reaches a maximum throughput of 3251 transactions per second generated by 1800 emulated browsers. This is a 10-fold improvement compared to one LTM.

Furthermore, we explore the cost-effectiveness of the two EC2 instance types for CloudTPS. The High-CPU medium instances cost 2 times more than Standard Small instances. As show in Figure 4.5(b), 20 High-CPU medium instances, which together cost $4 per hour, can sustain a higher throughput than 80 Standard Small instances, which together cost $8 per hour. For this application, using High-CPU medium instances are more cost-effective than Standard Small ones.

The linear scalability of CloudTPS relies on the property that transactions issued by Web applications access only a small number of data items, and thus span only a small number of LTMs. We illustrate this property by measuring the number of LTMs that participate in the transactions with the configuration of 40 LTMs servers. As shown in Figure 4.6, 91% of transactions access only two LTMs, i.e., one LTM and its backup. We expect this behavior to be typical of Web applications. The purchase transaction in Figure 4.4 is the only transaction that accesses

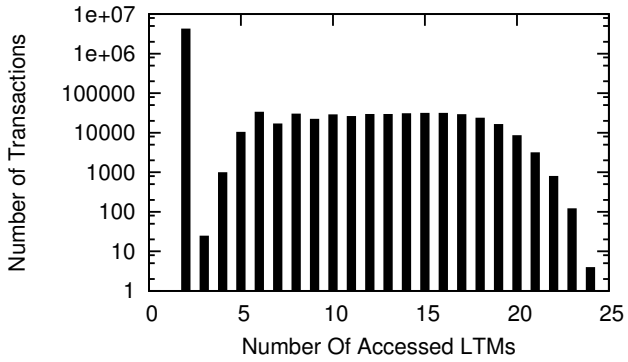


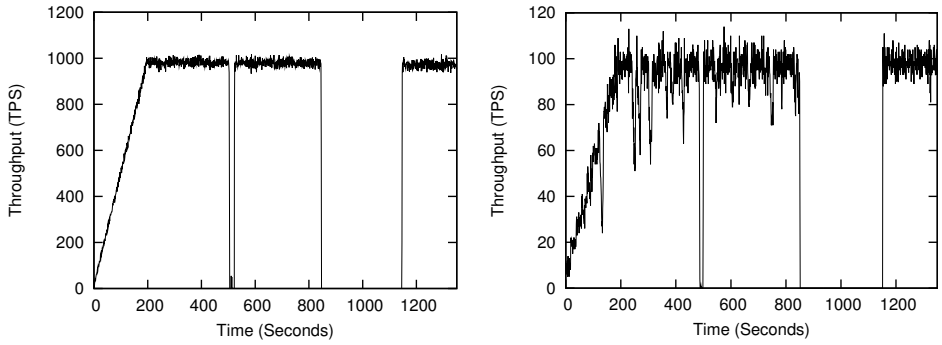
Figure 4.6: Number of LTMs accessed by the transactions of TPC-W with a total system size of 40 LTMs.

more than one data item. It first creates an order and clears the shopping cart inside the data item of the `Shopping` bigtable, then updates the stocks of all purchased items in the `Item_Stock` bigtable, and lastly updates the latest order ID of the customer in the `Latest_Order` bigtable. As the number of items contained in a shopping cart is uniformly distributed between 1 and 10, the number of data items accessed by a purchase transactions also has a uniform distribution between 3 and 12. Counting in the backup LTMs, the maximum number of accessed LTMs is 24. Figure 4.6 shows that larger number of purchase transactions access 5 to 19 LTMs. It is because the accessed data items may be located within the same LTM, so the number of accessed LTMs may be smaller than the number of accessed data items.

In our evaluations, we observe that CloudTPS is mostly latency-bound. For example, LTMs that are stressed to the point of almost violating the response time constraint never exhibit a CPU load above 50%, and their network bandwidth usage consistently remains very low. The main factors influencing performance are the network round-trip times and the queueing delays inside LTMs. CloudTPS is therefore best suited for deployments within a single data center. Some NoSQL data stores, such as PNUTS, replicate data across data centers to ensure low latency for geographically distributed user-base and tolerate failures of a complete data center. Using CloudTPS in such scenarios would increase network latencies between LTMs, thereby increasing response time of transactions and decreasing the throughput. Exploiting a multi-data center environment efficiently would require to revisit the policy which assigns data items to LTMs so that data items are placed close to the users which access them most. We however consider such extension as out of the scope of this thesis.

4.4.4 Tolerance to Failures and Partitions

We now study the system performance in the presence of LTM server failures and network partitions. We perform the evaluation in both DAS-3 and the Amazon



(a) DAS3+HBase. (b) EC2+SimpleDB.

Figure 4.7: Effect of LTM failures and network partitions.

cloud. We configure CloudTPS with 5 LTM servers, and each transaction and data item has one backup. We generate a workload using 500 EBs in DAS-3 and 50 EBs in the Amazon cloud, such that the system would not overload even after an LTM server failure. After the system throughput is stabilized, we first kill one LTM server. Afterwards, we simulate a 5-minute network partition where each partition contains one LTM server.

After detecting the failures, all live LTMs continuously attempt to contact other LTMs. The time delay between two attempts of contact follows a uniform distribution between 200 and 1200 milliseconds.

Figure 4.7(a) illustrates the evaluation in DAS-3. We first warm up the system by adding 25 EBs every 10 seconds. The full load is reached after 200 seconds. After running the system normally for a while, one LTM server is shutdown to simulate a failure at time 504 seconds. After the LTM failure it takes 18.6 seconds for the system to return to the previous level of transaction throughput. This duration is composed of:

- 0.5 second to rebuild a new membership, including a delay of 382 ms before the 2PC to avoid conflicts and 113 ms to build the new membership;
- 12.2 seconds to recover the blocked transactions which were accessing the failed LTM;
- 5.9 seconds to reorganize the data placement of LTMs to match the new system membership.

Afterward, at time 846 seconds, we simulate a network partition lasting for 5 minutes. When we restore the network partition, the system recovers and returns to the previous level of transaction throughput in 135 milliseconds. The reason why the system recovers so fast is because there is no LTM failure along with the network partition, so all blocked transactions can resume execution without recovery, and no data redistribution is necessary.

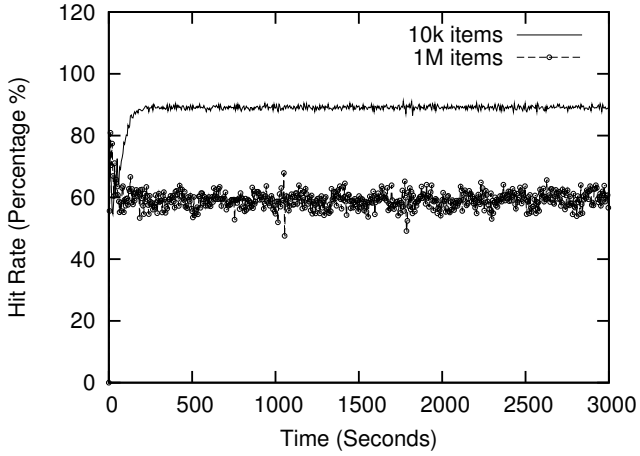


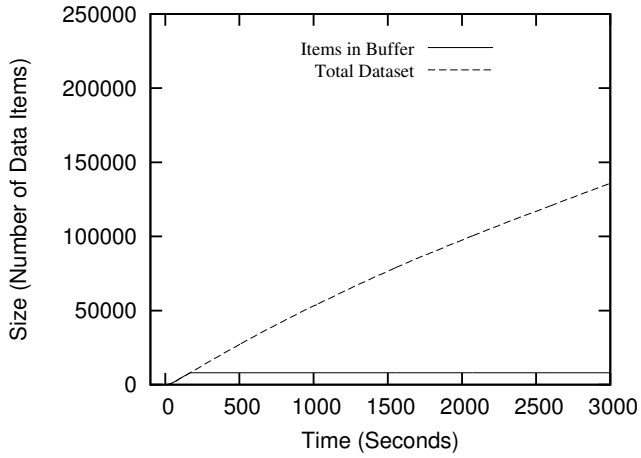
Figure 4.8: Hit rate of LTM #1.

Figure 4.7(b) depicts the same evaluation in Amazon EC2. The LTM server fails at time 486 seconds. After detecting the LTM failure, the system spends 13 seconds to recover and the transaction throughput returns to the previous level at time 499 seconds. During the failure recovery, the remaining 4 LTMs first merge into one partition in about 1 second. Then the system recovers transactions in 4 seconds and reorganizes data placement in 8 seconds. Later, the system encounters a 5-minutes network partition. After the network partition is restored, the system recovers in 207 milliseconds and returns to the previous level of transaction throughput at time 1152 seconds. The system throughput in the Amazon cloud fluctuates more than in DAS-3 because we have less control of virtualized resources in the Amazon cloud.

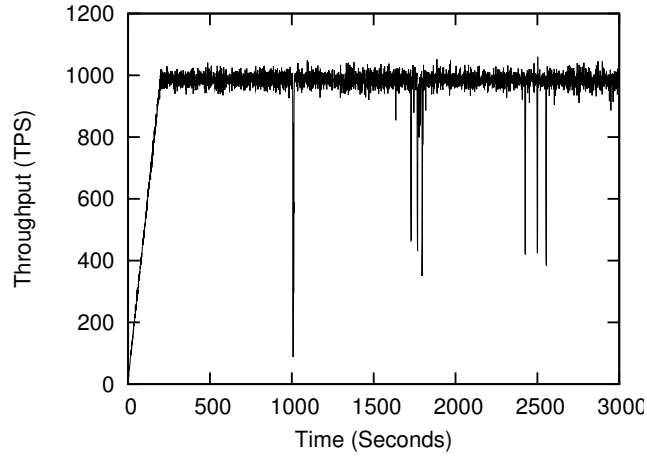
4.4.5 Memory Management

Lastly, we demonstrate that our memory management mechanism can effectively prevent LTMs from memory overflow, and study the performance of CloudTPS with different buffer sizes and data sizes. We carry out the evaluation in DAS-3 on top of HBase, which provides read-your-writes consistency. We configure the system such that, before evicting a data item, LTMs fetch the data item from HBase and verify that the obtained value reflects the latest in-memory updates. Therefore, this performance evaluation represents the system implementation for the NoSQL data stores supporting monotonic-reads consistency level.

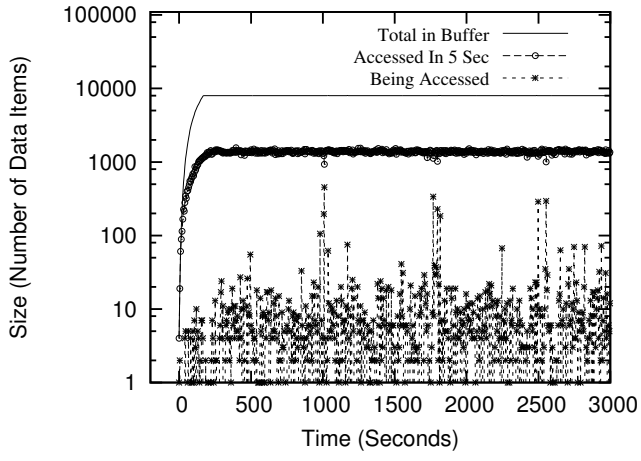
We first deploy a system with 3 LTMs and impose a constant workload for one hour. We configure the system so that each LTM can maintain at most 8000 data items in its buffer. We then evaluate the system under two different scales of data set sizes: either 10,000 or 1,000,000 records in the `Item_Stock` table. For the



(a) Buffer size of LTM #1.

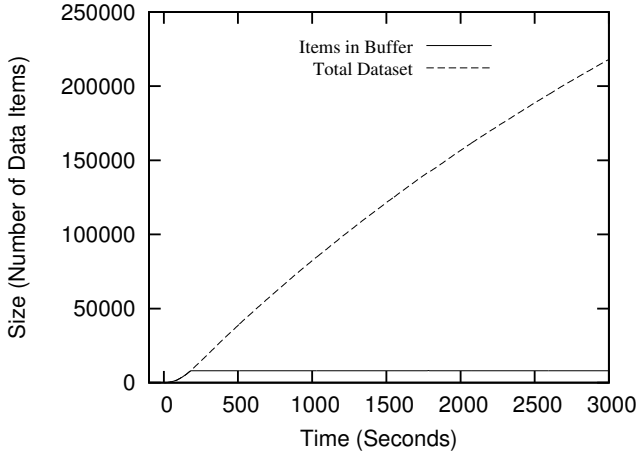


(b) Total system throughput.

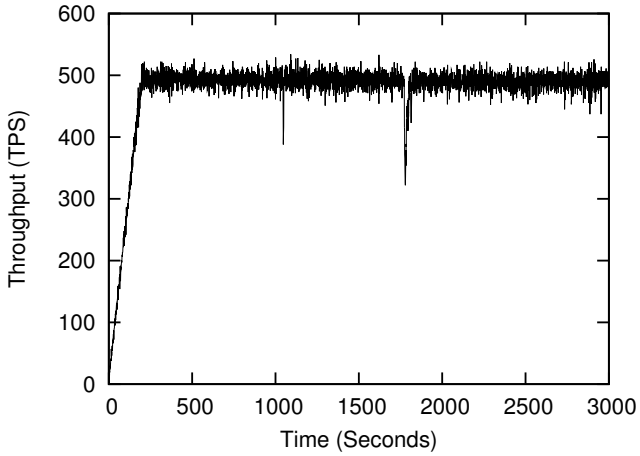


(c) Data locality of LTM #1

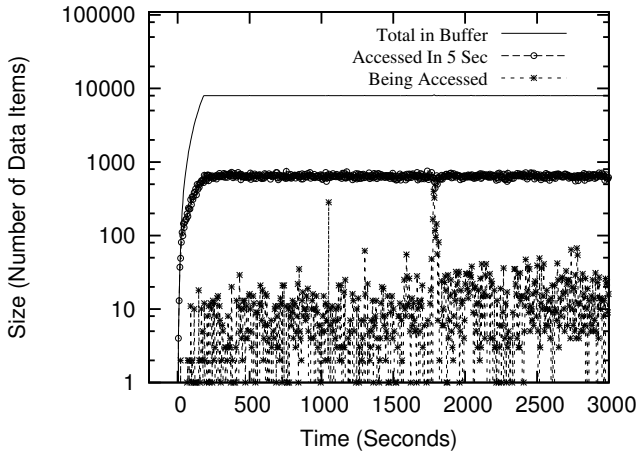
Figure 4.9: Memory management evaluation with 10,000 items.



(a) Buffer size of LTM #1.



(b) Total system throughput.



(c) Data locality of LTM #1.

Figure 4.10: Memory management evaluation with 1,000,000 items.

data size of 10,000 items, we impose a workload of 500 EBs. For the data size of 1 million items, we impose 250 EBs.

Figure 4.9(a) shows that under both data set sizes, our mechanism effectively maintains the buffer size of LTM #1 within the limit of 8000 data items. Using 10,000 `Item_Stock` items, without memory management, after an hour, this LTM would have to maintain almost 140,000 data items in memory. As for the data size of 1,000,000 `Item_Stock` items, Figure 4.10(a) shows that, after an hour, the total data set increases to an even larger number of more than 200,000 data items. In both cases, without memory management, the size of the total accessed data set increases almost linearly, which would eventually cause a memory overflow.

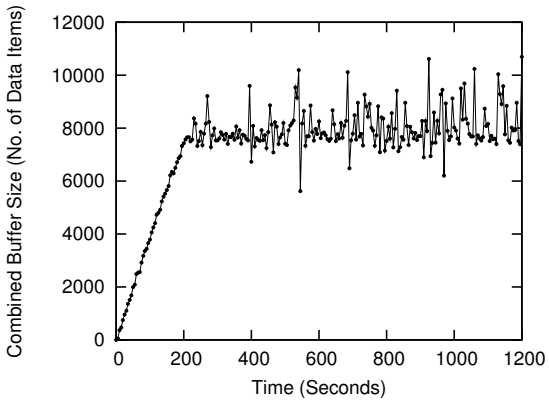
We then compare the performance of the system under different data set sizes. Figure 4.8 shows that the hit rate of LTM #1 stabilizes around 90% for 10,000 items, and about 60% for 1 million items. The other LTMs in the system behave similarly. Figures 4.9(b) and 4.10(b) show the total transaction throughput during the 1-hour evaluation. The drops of throughput at some points are due to the JVM garbage collection, which temporarily block the LTMs. With 10,000 items, the system sustains a transaction throughput of about 1000 TPS and 99.4% of transactions complete within 100 ms. For 1 million items, the system sustains about 500 TPS, but only 97% of transactions satisfy the performance constraint.

The efficiency of our memory management mechanism depends on the data locality of the Web application. Figures 4.9(c) and 4.10(c) show that only very few data items are being accessed at a time in the two different scenarios. Note that Figures 4.9(c) and 4.10(c) are in log scale. Comparing to the total accessed data items shown in Figure 4.9(a), this application shows strong data locality which implies that our mechanism can only introduce minor performance overhead[3].

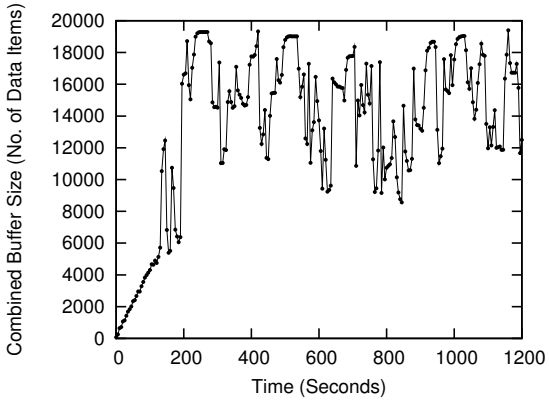
Finally, we study performance with different buffer sizes of LTMs in terms of 99th percentile of response times of the system. We configure the system with 20 LTMs and impose a workload of 2400 EBs, which issues about 4800 transactions per second. We start with the minimum buffer size required by LTMs to maintain the ACID properties, where only the absolutely necessary data items remain in the buffer. To achieve this, we evict any evictable data items as soon as possible. We then increase the buffer size until no data item is evicted at all. Similar to the previous evaluation, we evaluate the system performance with 10,000 and 1 million records in the `Item_Stock` table.

Figure 4.11 shows the combined buffer size of all LTMs when applying the Evict-Now algorithm. For the data size of 10,000 items, the average buffer size is 7957 data items, which means 397 data items per LTM. For the data size of 1 million items, the average buffer size is 14,837, so 741 data items per LTM. Figure 4.12 shows the performance of our system under different buffer sizes. The initial value of each line in Figure 4.12 indicates the 99th percentile of response

[3]Note that TPC-W randomly selects books to add into a shopping cart with uniform distribution. Several works consider that this behavior is not representative of real applications and create extra locality artificially [8, 76, 97]. We can thus consider unmodified TPC-W as a somewhat worst-case scenario.



(a) With 10,000 items.



(b) With 1,000,000 items.

Figure 4.11: Minimum total buffer size for 20 LTMs under a load of 2400 EBs.

times of the system using Evict-Now algorithm. Therefore, we adopt 397 and 741 as the initial values for the X-axis in Figure 4.12. Note that we plot this figure in log scale.

We first study the 99th percentile of response times with 10,000 items. When we increase the buffer size from the minimum size of 397 to 1000 data items per LTM, the 99th percentile response time decreases dramatically from 799 ms to 62 ms. When we continue increasing the buffer size to 100,000 data items where no data item is evicted at all, the 99th percentile response time only improves to 46 ms. In other words, increasing the buffer size from 397 to 1000 data items, the response time of the system decreases by an order of magnitude. Increasing the buffer size even further by two orders of magnitude to 100,000 data items can only achieve 25% further reduction of response time. At the point of 1000 data items per LTM, the overall buffer size of the system reaches 20,000 data items, which is large enough to contain almost all 10,000 `item_stock` data items and other currently accessed data items from other two tables. Increasing the buffer size even further

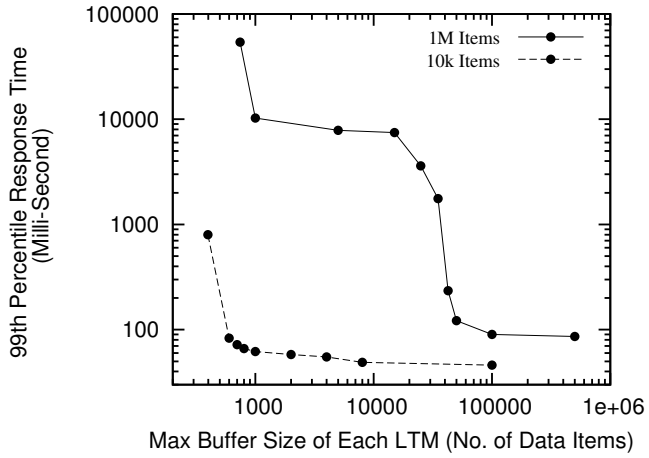


Figure 4.12: Response time with different buffer sizes.

can only allow to store seldomly accessed data items, and thus cannot effectively improve the hit rate of the system.

With 1 million items, the 99th percentile of response times decreases dramatically from 54 seconds to 10 seconds, when the buffer size increases from the minimum size of 741 to 1000 data items per LTM. However, increasing the buffer space from 1000 to 15,000 does not bring much performance improvement, because the total data size is so large that the hit rate remains roughly the same. If we continue increasing the available storage from 15,000 to 100,000, the 99th percentile of response times decreases dramatically again from 7486 ms to 90 ms. After the point of 100,000 data items, continue increasing the buffer size further does not bring significant performance improvement.

Comparing the two lines in Figure 4.12, we notice that a good buffer size for 10,000 items could be 1000 data items. For the line of "1M items," we can find a similar point of 100,000 data items. In both cases, this represents about 10% of the total data set size.

4.5 Conclusion

Many Web applications need strong data consistency for their correct execution. However, although NoSQL data stores provide high scalability and availability properties, they provide only relatively weak consistency properties. This chapter shows how one can support strict ACID transactions on NoSQL data stores without compromising their scalability property.

This work relies on few simple ideas. First, we load data from the NoSQL data store into the transactional layer. Second, we split the data across any number of LTMs, and replicate them only for fault tolerance. Web applications typically access only a few partitions in any of their transactions, which gives CloudTPS linear

scalability. CloudTPS supports full ACID properties even in the presence of server failures and network partitions. Recovering from a failure causes only a temporary drop in throughput and a few aborted transactions. Recovering from a network partition, however, may possibly cause temporary unavailability of CloudTPS, as we explicitly choose to maintain strong consistency over high availability. Our memory management mechanism can prevent LTM memory overflow. We expect typical Web applications to exhibit strong data locality so this mechanism allows maintaining only a small part of application data in the memory of LTMs. This prevents using many LTMs just for their memory capacity. Besides, our evaluation shows that the mechanism introduces only minor performance overhead. Data partitioning also implies that transactions can only access data by primary key. Read-only transactions that require more complex data access can still be executed, but on a possibly outdated though internally consistent snapshot of the database.

This chapter demonstrates that the weak consistency properties of NoSQL databases are not a fatality, and that one can enforce strong transactional consistency without losing their good scalability and fault-tolerance properties. CloudTPS allows Web applications with strong data consistency demands to make use of NoSQL data stores. Compared with the approach from Chapter 3, CloudTPS partitions data automatically and provides properties of incremental scalability and fault tolerance. However, CloudTPS requires its transactions to access only well-identified data items, which rules out complex queries such as join queries. The next chapter shows how to extend the current design of CloudTPS to support join queries while still maintaining transactional properties.

Chapter 5

Consistent Join Queries in NoSQL Data Stores

NoSQL data stores provide good properties of scalability and high availability. However, they usually relax data consistency and support only very simple types of queries that select data records from a single table by their primary keys. In Chapter 4, we have seen how to implement ACID transactions on top of NoSQL data stores without compromising scalability. This chapter now addresses the remaining issue of supporting complex queries such as join queries. Importantly, we implement join queries without compromising the transactional or scalability properties.

Join queries are an essential feature for any database system, as they allow to query related information from multiple tables in a single atomic operation. These queries are often the result of data normalization techniques, which have been used for decades to help guaranteeing semantic data integrity in large systems.

The relational data model, typically implemented via the SQL language, provides great flexibility in accessing data, including support for sophisticated join queries. However, the features of flexible data querying and strong data consistency prevent one from partitioning data automatically. In Chapter 3, we have seen that scaling Web applications with relational databases requires significant manual efforts in denormalizing data and restructuring applications.

NoSQL data stores partition data automatically, but do not support join queries. The lack of join queries in NoSQL data stores can often be mitigated using techniques such as rewriting a join query into a sequence of simple primary-key queries. However, such translation is not a trivial task at all. First, one must design data schemas carefully to allow such query rewrites. Second, and more importantly, programmers need sufficient understanding of subtle concurrency issues to realize and handle the fact that a sequence of simple queries is equivalent to the original join query only in the case where no update of the same data items is issued at the same time. Although skilled programmers can effectively develop good applications using this data model, we consider that program correctness should not be an

optional feature left under the sole responsibility of the programmers. Correctness should as much as possible be provided out of the box, similar to the foolproof strong consistency properties of relational databases.

This chapter discusses the design and implementation of join queries that are strongly consistent by design, relieving programmers from the burden of adapting their programs to the peculiarities of NoSQL data stores. At the same time the system retains the scalability properties of the NoSQL data stores. We implement join queries in CloudTPS, of which the transactional functionalities have been presented in Chapter 4. This chapter focuses on CloudTPS's support for consistent join queries, while retaining the original scalability and fault-tolerance properties of the underlying NoSQL data store.

CloudTPS supports a specific type of join queries known as foreign-key equi-joins: the matching relationship between two records is expressed as an equality between a foreign key and a primary key. This is by far the most common type of joins in Web applications. For example, every join query issued by Wikipedia to its database belongs to this category[1] [115]. Support for this family of join queries also allows us to implement secondary-key queries: CloudTPS only needs to maintain a separate index table that maps secondary-key values back to their corresponding primary keys. Secondary-key queries are then translated into equivalent join queries. When the main table is updated, its associated index table is updated atomically as well.

The scalability properties of CloudTPS originate from the fact that most queries issued by Web applications (including transactions and join queries) actually access a small number of data items compared with the overall size of the database. This property is verified in all real-world Web applications that we studied: because database queries are embedded in the processing of an end-user HTTP request, programmers tend to naturally avoid complex and expensive queries which would for example scan the entire database.

We demonstrate the performance and scalability of CloudTPS using a realistic workload composed of primary-key queries, join queries, and transactions issued by the TPC-W Web hosting benchmark. This benchmark was originally developed for relational databases and therefore contains a mix of simple and complex queries similar to queries Web applications would use if their NoSQL data store supported them. We show that, with no change of the initial relational data schema nor the queries addressed to it, CloudTPS achieves linear scalability while enforcing data correctness automatically. In large-scale configurations, CloudTPS outperforms replicated PostgreSQL up to three times.

This chapter is organized as follows. Sections 5.1 describes the CloudTPS database model which includes the data and query model. Section 5.2 presents the system design and Section 5.3 discusses implementation details. Section 5.4 presents performance evaluations. Section 5.5 concludes this chapter.

[1] Wikipedia's query workload also contains aggregate queries, which are out of the scope of this thesis.

5.1 Database Model

We now detail the CloudTPS data model, the type of join queries it supports, and the way they are expressed by programmers in our system.

5.1.1 Data Model

Different NoSQL data stores employ similar yet different data models to define how the data are organized. However, CloudTPS aims to support join queries for a wide range of underlying data stores. For example, Bigtable and SimpleDB use similar data models with tables, rows and columns; however, Bigtable requires defining column families as a prefix for column names, while SimpleDB does not impose any schema; SimpleDB supports multiple values for a column, while Bigtable supports only a single value for one column but with multiple versions. Similar to AppScale [19] (which also aims to unify access to many different NoSQL databases), CloudTPS must define a single logical data model to be mapped over different physical data store models. However, while AppScale defines its unified data model as simple key-value pairs with queries spanning only a single table, CloudTPS needs a more structured data model to support complex join operations across multiple tables.

CloudTPS defines its data model as a collection of tables. Each table contains a set of records. A record has a unique Primary Key (PK) and an arbitrary number of attribute-value pairs. An attribute is defined as a Foreign Key (FK) if it refers to a PK in the same or another table. Applications may use other non-PK attributes to look up and retrieve records. These attributes are defined as Secondary Keys (SK) and are supported in CloudTPS by creating a separate index table which maps each SK to the list of PKs where this value of the SK is found. A secondary-key query can thus be transformed into a join query between the index table and the original table. CloudTPS expects applications to define the table schema in advance, with the table names, the PK, all the SKs and FKs together with their referenced attributes. Other non-key attributes can be left undefined in the table schema and different records of the same table need not to share the same set of attributes.

Figure 5.1 shows an example data model which defines four data tables and one index table. The table `book` defines `book_id` as its primary key. The FK `author_id` of table `book` refers to the PK of table `author`. To support secondary-key queries which select books by their titles, CloudTPS automatically creates an index table `indexOf_bookTitle`. Each record of table `book` matches the record of table `indexOf_bookTitle` of which the PK value equals its SK `title`. Therefore, the SK `title` is also a FK referring to the PK of the index table `indexOf_bookTitle`.

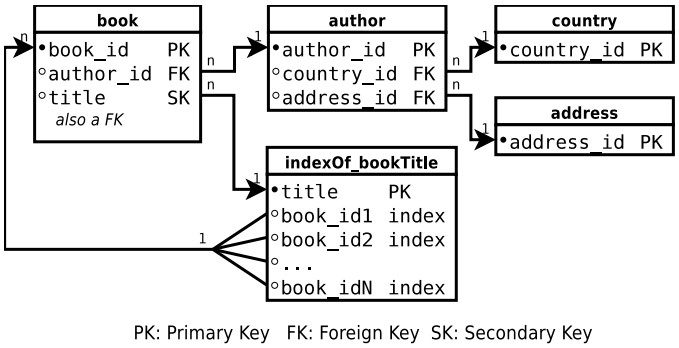


Figure 5.1: An example data model for CloudTPS

5.1.2 Join Query Types

A join query combines records from two or more tables in a database following the relationship constraints defined in the query. CloudTPS restricts this very general definition to support a specific class of join queries known as foreign-key equi-join, where the relationship between two records is expressed as an equality between a FK and a PK (in the same or another table). For example, one such constraint can be that the author name found in the `book` record matches the author name found in the matching `author` record. The query for this example is "`SELECT * FROM book, author WHERE book.book_id = 10 AND book.author_id = author.author_id`."

Equi-joins are by far the most common join queries, compared to relationships such as less than or greater than. These queries often result from database normalization methodologies. For example, we observed that all join queries in Wikipedia follow this structure.

In CloudTPS, join queries must give an explicit list of primary keys referring to initial records found in one table. These records and the table where they are stored are referred to as the root records and root table of this join query, respectively. This restriction excludes in particular full table scans to join two tables completely.

This thesis considers only *inner-joins* which return all records that have at least one matching record, while the final combined record contains merged records from the concerned tables. Other types of join, such as outer-join (which may return records with no matching record), and semi-joins (which only return records from one table), are out of the scope of this thesis.

5.1.3 API

Web applications access CloudTPS using a Java client-side library, which offers mainly two interfaces to submit respectively join queries and transactions.

Join queries are expressed as a collection of `JoinTable` and `JoinEdge` Java objects. A `JoinTable` object identifies one table where records must be found. It contains the table name, the projection setting (a list of attributes to be returned)

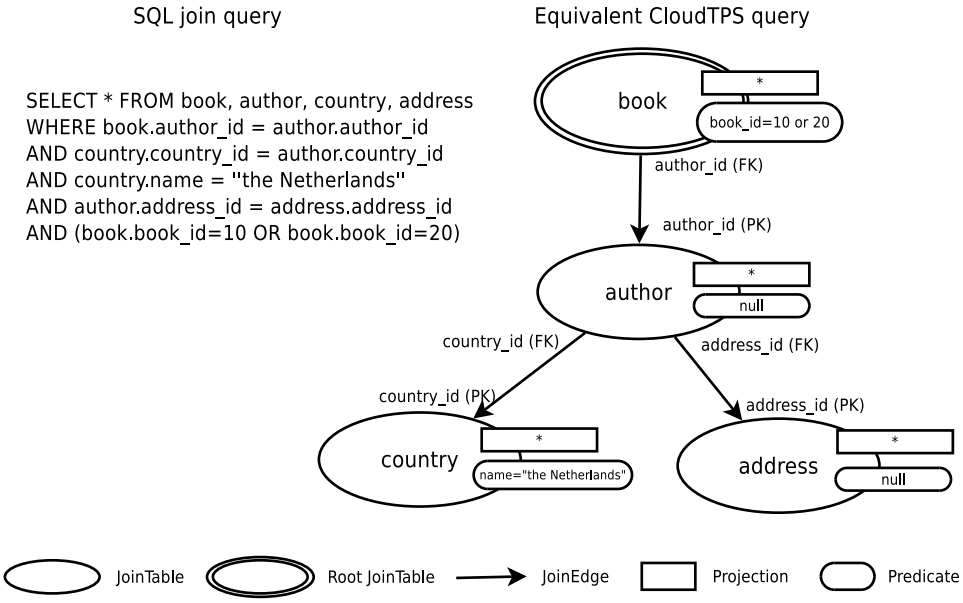


Figure 5.2: CloudTPS's representation of a join query

and possibly a predicate (a condition that a record must satisfy to be returned). A `JoinEdge` object represents a join operation between two tables. It contains references to the two `JoinTable` objects, the names and properties (i.e., PK or FK) of join attributes. A join query must designate one `JoinTable` object as the root table, which contains the list of primary keys of the root records. Multiple `JoinTable` objects are joined together using `JoinEdge` objects of which each matches the FK from one `JoinTable` to the PK of another. Self-join queries are supported by creating two `JoinTable` objects with the same table name.

Figure 5.2 shows the SQL and CloudTPS representations of a join query which retrieves information about two books and their authors. The `book` object is the root table, and the primary keys of root records are `10` and `20`. A `JoinEdge` starts from `JoinTable book` to `author` indicating the FK `author_id` in table `book` refers to the PK of `JoinTable author`. The predicate of `JoinTable country` selects only books with an author from the Netherlands.

CloudTPS also handles read-only and read-write transactions defined as a `Transaction` Java object containing a list of `SubTransaction` objects. Each subtransaction represents an atomic list of operations on one single record. All subtransactions are implemented as subclasses of the `SubTransaction` abstract Java class. As shown in Figure 4.2, each subtransaction contains a unique `className` to identify itself, a table name and primary key to identify the accessed data item, and input parameters organized as attribute-value pairs.

5.2 System Design

CloudTPS considers join queries as a specific kind of multi-row transactions. It therefore enforces full transactional consistency to the data they access, even in the case of machine failures or network partitions. Note that the underlying NoSQL data stores do not need to guarantee strong consistency across multiple data items.

CloudTPS consists of a number of Local Transaction Managers (LTMs). To ensure strong consistency, CloudTPS maintains an in-memory copy of the accessed application data. Each LTM is responsible for a subset of all data items. We assign data items to LTMs using consistent hashing [57] on the item's primary key. This means that any LTM can efficiently compute the identity of the LTM in charge of any data item, given its primary key. Transactions and join queries operate on this in-memory data copy, while the underlying NoSQL data store is transparent to them.

Figure 4.1 shows the organization of CloudTPS. Clients issue HTTP requests to a Web application, which in turn issues queries and transactions to CloudTPS. A transaction or join query can be addressed to any LTM, which then acts as the coordinator across all LTMs in charge of the data items accessed by this query. If an accessed data item is not present in the LTM's memory, the appropriate LTM will load it from the NoSQL data store. Data updates resulting from transactions are kept in memory of the LTMs and later checkpointed back to the NoSQL data store. LTMs employ a replacement policy so that unused data items can be evicted from their memory (the caching policy is discussed in details in Chapter 4). CloudTPS expects LTMs to be connected by a low-latency network as in a data center.

5.2.1 Join Algorithm

Join Queries

Intuitively, processing a join query spanning multiple tables requires to recursively identify matching records, starting from the root records (known by their primary keys) and following `JoinEdge` relationships. Take the join query in Figure 5.2 for example, LTMs first access the root records in table `book` and identify the matched records in table `author`, then access the identified `author` records and lastly join them with the leaf table `country` and `address`. All the matched records are returned to the coordinator who combines them into the final result for the client.

The method to identify the matched records, however, differs according to the role of the given records. If an already known record contains a FK which references the PK of a new record, then the new record can be efficiently located by its PK. We call this type of join queries forward join queries. On the other hand, if the PK of an already known record is *being referenced* by the FK of a new record to be found, then in principle it is necessary to scan the full table and search for all records whose FK is equal to the PK of the known record. We name such join

Forward query	SELECT * FROM author, book WHERE book.book_id = 10 AND book.author_id = author.author_id
Backward query	SELECT * FROM author, book WHERE author.author_id = 100 AND book.author_id = author.author_id

Table `author`

author_id (PK)	(CloudTPS index entry)
100	Ref::book::author_id::10 = 10 Ref::book::author_id::30 = 30
101	Ref::book::author_id::20 = 20

Table `book`

book_id (PK)	author_id	title
10	100	title1
20	101	title2
30	100	title3

Figure 5.3: Index data layout example, with two types of join queries

queries backward join queries. To avoid a prohibitively expensive full table scan for each backward join query, we use a technique similar to join indices in centralized databases [78, 110]. We complement the referenced table with direct links to the PKs of matching records. This allows to translate such queries into forward join queries. On the other hand, we now need to maintain these indexes every time the tables are updated. If a data update changes the reference relationships among records, the update query must be dynamically translated into a transaction in which the indexes are updated as well.

Figure 5.3 shows an example index data layout to support a forward and a backward join query, for the same data schema as in Figure 5.1. The table `book` contains a FK `author_id` referring to the PK value of table `author`. The forward join query can be processed directly without any indexes, as the FK `author_id` of its root `book` record identifies that the PK of its matched `author` is `100`. The backward join query, in contrast, starts by accessing its root record in table `author` and requires additional indexes to identify the matching record. The indexes are stored as arbitrary number of index attributes in each record of the referenced table. Doing this does not require to change the data schema as all NoSQL data stores support the dynamic addition of supplementary fields onto any data item. Each index attribute represents one matched referring record with the corresponding FK. CloudTPS creates these indexes upon the declaration of the data schema, then maintains their consistency automatically. In Figure 5.3, the `author` record of PK(100) contains two index attributes showing that this record is referenced by two `book` records with PK `10` and `30`. Using these indexes, the backward join query in Figure 5.3 can then efficiently identify the two matching `book` records.

Secondary-Key Queries

We use a similar solution by building explicit indexes. However, unlike joins, there exists no table where we can add index information. Instead, we create a separate index table for each SK. Each record of the index table has its PK equal to the SK of one or more records, of which the PKs are stored as its index attributes. A secondary-key query can then translate into a forward join query between the

Table 5.1: Translating a secondary-key query into a join query

Original	SELECT * FROM book WHERE book.title="bookTitle"
Translated query	SELECT * FROM book WHERE indexOf_bookTitle.title= "bookTitle" AND book.title=indexOf_bookTitle.title

index table and the original table. Table 5.1 shows an example secondary-key query which searches records by the SK `title` of table `book`. The translated query first locates the root record in the index table by using the given SK value `bookTitle` as the PK value. It then retrieves the PKs of the matched `book` records.

5.2.2 Consistency Enforcement

To ensure strong consistency, CloudTPS implements join queries as multi-item read-only transactions. Our initial implementation of CloudTPS, as shown in Chapter 4, already supported multi-item transactions. However, it required the primary keys of all accessed data items to be specified at the time a transaction is submitted. This restriction excludes join queries, which need to identify matching data items during the execution of the transaction. Besides, it also prohibits transparent index management as programmers would be required to provide the primary keys of the records containing the affected index attributes. To address this issue, we propose two extended transaction commit protocols: (i) for read-only transactions to support join queries, and (ii) for read-write transactions to support transparent index management.

To implement join queries consistently, we use the same mechanisms as described in Chapter 4 to implement the isolation, consistency and durability properties:

Consistency means that a transaction executing on a database that is internally consistent, will leave the database in an internally consistent state. We assume that consistency rules are applied within the logic of transactions, so consistency is ensured as long as all transactions are executed correctly.

Isolation means that the behavior of a transaction is not impacted by the presence of other concurrent transactions. In CloudTPS, each transaction is assigned a globally unique timestamp. LTMs are required to execute conflicting transactions in the order of their timestamps. Transactions which access disjoint sets of data items can execute concurrently.

Durability means that the effects of committed transactions will not be undone, even in the case of server failures. CloudTPS checkpoints the updates of committed transactions back to the NoSQL data store. During the time between a transaction commit and the next checkpoint, durability is ensured by replicating the data items and transaction states across several LTMs.

We now turn to **Atomicity**: either all operations of a transaction succeed successfully, or none of them does.

In CloudTPS, a transaction is composed of any number of subtransactions, where each subtransaction accesses a single data item atomically. To enforce atomicity, transactions issue a two-phase commit (2PC) across all LTMs responsible for the accessed data items. As shown in Figure 5.4(a), in the first phase, the coordinator submits all the subtransactions to the involved LTMs and asks them to check that the operation can indeed be executed correctly. If all LTMs vote favorably, the second phase actually commits the transaction. Otherwise, the transaction is aborted. To implement join queries, we however need to extend 2PC into two different protocols respectively: one for join queries as read-only transactions, and one for transparent index management in read-write transactions.

Read-Only Transactions for Join Queries

The 2PC protocol requires that the identity of all accessed data items is known at the beginning of the first phase. However, join queries can identify matching records only after accessing the root records. We therefore extend the 2PC protocol. During the first phase, when the involved LTMs complete the execution of their subtransactions, besides the normal commit and abort messages, they can also vote conditional commit which requires more subtransactions to be added to the transaction. The LTM submits the new subtransaction to both the responsible LTM and the coordinator. The responsible LTM executes this new subtransaction, while the coordinator adds it to the transaction and waits for its vote. The coordinator can commit the transaction only after no subtransaction requests to add new subtransactions.

As read-only transactions do not commit any updates, LTMs can terminate these subtransactions immediately after all concerned LTMs return their votes. The coordinator therefore does not need to send the commit messages to the involved LTMs. Transactional consistency is enforced by the timestamp ordering protocol: concurrent read-only transactions which access non-disjoint sets of data items are executed in the same order at all LTMs.

This extension allows join queries to access root records first, then add matching records to the transaction during the query execution. For example, in Figure 5.4(b) the coordinator is initially aware of only two root records held by LTM 15 and LTM 66. After executing its subtransaction, LTM 15 identifies a matching record hosted by LTM 34. LTM 15 submits the new subtransaction to LTM 34 directly, and also returns the new subtransaction along with its conditional commit vote to the coordinator. On the other hand, LTM 66 identifies no matching record so it simply returns commit. Finally, LTM 34 executes the new subtransaction and also returns commit with no more new subtransactions. The coordinator can then commit the transaction by combining the records together and returning the result to the client.

The implementation of this extended transaction commit protocol requires adaptation of both the transaction coordinator and the subtransaction engine for read–only transactions. Figure 5.5 shows the algorithm's pesudocode for read-only

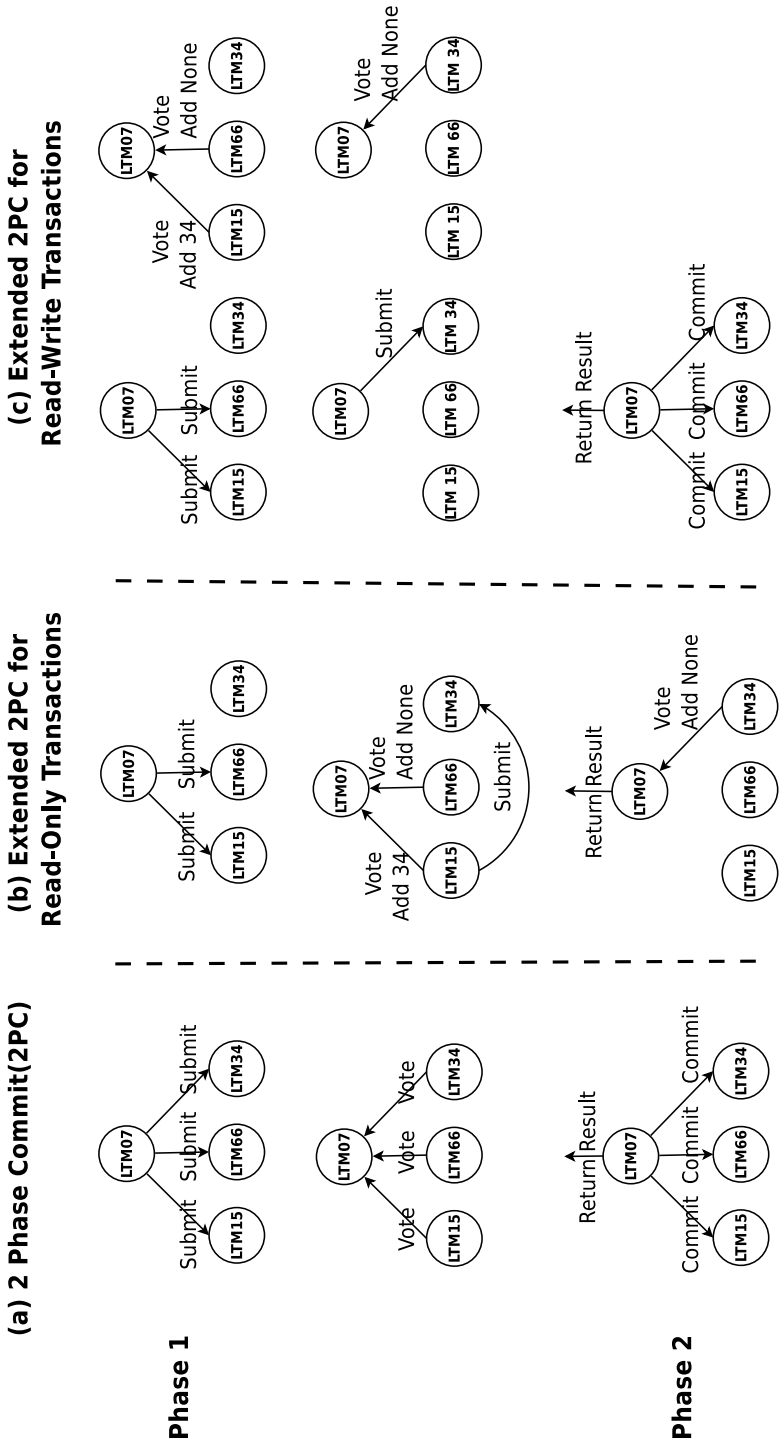


Figure 5.4: Two-phase commit vs. the extended transaction commit protocols

```
1  public class ROTransactionCoordinator {
2    JoinQuery query;
3    List<ROSubTransaction> subtrans;
4
5    public void start(){
6      getTimestamp();
7      buildROSubTrans(query.rootTable, subtrans);
8      submit(subtrans); //access root records
9    }
10
11   public handleVote(ROVote msg){ //Collecting results
12     query.update(msg.table, msg.PK, msg.isValid, msg.data);
13     query.addRecord(msg.matchingRecords); //adding to the transaction
14     if(query.isAllRecordsAccessed()){
15       returnResultToClient(); //transaction terminated
16     }
17   }
18 }//End of class ROTransactionCoordinator
19
20 public class ROSubTransaction extends SubTransaction {
21   //Input
22   JoinQuery query;
23   JoinTable table;
24
25   //Output
26   boolean isValid;
27   Attribute[] projectedRecord; //the set of to−be−returned attributes
28   String[] matchingRecords; //the primary keys of matched Records
29   List<ROSubTransaction> derivedSubtrans;
30
31   //Operation
32   public VoteResult run(){
33     isValid=verify(table.predicate, super.dataItem); //check predicate
34     if(isValid) { //continue only if the predicate is satisfied
35       projectedRecord=project(table.projection, super.dataItem);
36       matchingRecords=identify(table.edges, super.dataItem);
37       if(matchingRecords.size>0) //exist matched records?
38         derivedSubtrans=generateSubTrans(matchingRecords);
39         submit(derivedSubtrans);
40         return CONDITIONAL_COMMIT;
41     }
42     return COMMIT;
43   }
44 } //End of class ROSubTransaction
```

Figure 5.5: The pseudocode of the extended read-only transaction implementation

transactions. The transaction coordinator starts by translating the input join query into subtransactions to access the root records. After submitting these subtransactions, the transaction coordinator then acts as the result collector receiving the content of the accessed records and registering the primary keys of the newly identified matching records. When all registered records have been accessed, the coordinator combines the retrieved records and returns them to the client.

The new subtransaction class `ROSubTransaction` extends the abstract Java class `SubTransaction` as defined in Figure 4.2. Its input variables indicate the record it accesses, the predicate it must satisfy and the projection setting to select the attribute-values to be returned. It overrides the `run()` operation to implement the join algorithm. It first verifies if the accessed record satisfies the conditions as defined as the `predicate` of its `JoinTable`. Only the records that satisfy these conditions can be identified as valid matching records. An example of such condition is shown in Figure 5.2 where a valid matching `country` record must have its attribute `name` equal to "the Netherlands." If satisfied, the member variable `isValid` is set to true and the subtransaction continues to look for matching records for this currently accessed record. The primary keys of the identified matching records are stored in `matchedRecords`. For each item in `matchedRecords`, the LTM generates one subtransaction and submits it to the responsible LTM. If the `JoinTable` is a leaf table or the record is not valid, the generated `matchedRecords` is empty so that this subtransaction terminates without creating any more new subtransactions.

A join query may require to return only a subset of attributes, which is defined in the `projection` of the `JoinTable`. A subtransaction stores the specified attributes in the `projectedRecord`. After the operation `run()` has executed, the LTM returns the output member variables back to the coordinator along with the vote.

In case of machine failures or network partitions, LTMs can simply abort all read-only transactions without violating the ACID properties.

Read-Write Transactions for Index Management

CloudTPS transparently creates indexes on all FKs and SKs. To ensure strong data consistency, when a read-write transaction updates any data items, the affected index attributes must also be updated atomically. As each index attribute stands for a referring record matching to its belonging record, when the FK of this referring record is inserted/updated/deleted, the corresponding index attribute must also be adjusted. To enforce strong data consistency, these affected index attributes must be updated within the same read-write transaction. Considering the example in Figure 5.3, a read-write transaction could insert a `book` record which matches an existing record in table `author`. When this read-write transaction commits, the primary key of this new `book` record must already be stored as an index attribute into the corresponding `author` record.

CloudTPS creates indexes automatically, so index maintenance must also be

transparent to the programmers. Here as well, this means that transactions must be able to identify data items to be updated during the execution of the transaction. For example, a query which would increment a record's secondary key needs to first read the current value of the secondary key before it can identify the records it needs to update in the associated index table.

To implement transparent index management, we extend the commit protocol for read-write transactions to dynamically add subtransactions. Similar to the extension for read-only transactions, during the first phase, LTMs can generate and add more subtransactions to access new data items. However, unlike in read-only transactions, LTMs should not submit new subtransactions to the responsible LTMs directly. In read-write transactions, if any subtransaction votes abort, the coordinator sends abort messages to all current subtransactions immediately, in order to minimize the blocking time of other conflicting transactions. Allowing LTMs to submit new subtransactions directly to each other opens the door to ordering problems where the coordinator received the information that new subtransactions have been added after it has aborted the transaction. Therefore, in read-write transactions, the involved LTMs submit new subtransactions to the coordinator only. The coordinator waits until all current subtransactions return before issuing any additional subtransactions. The coordinator can commit the transaction when all subtransactions vote commit and do not add any new subtransactions. If any subtransaction in any phase votes abort, then the coordinator aborts all the subtransactions.

We can implement transparent index management with this protocol. Whenever a subtransaction is executed, the LTM in charge of this data item automatically examines the updates to identify the affected index attributes. If any FKs or SKs are modified, the LTM then generates new subtransactions to update the affected index attributes.

Figure 5.4(c) shows an example of the extended read-write transaction. Initially, the coordinator LTM 07 submits subtransactions to update data items hosted in LTM 15 and LTM 66. LTM 15 identifies an affected index attribute hosted by LTM 34, while LTM 66 identifies none. LTM 15 thus generates a new subtransaction for updating this index attribute and returns it back to the coordinator along with its vote of commit. After both LTM 15 and LTM 66 vote commit, the coordinator starts a new phase and submits the new subtransaction to LTM 34. After LTM 34 also votes commit, the coordinator finally commits the transaction.

Figure 5.6 shows the pseudocode of extended read-write transactions. At the beginning, the coordinator submits the subtransactions given by the client. The coordinator then waits for these subtransactions to vote and possibly return new subtransactions. At line 11, after all current subtransactions vote commit, if there are new added subtransactions in this phase, the coordinator then submits these new subtransactions and waits for their votes. If there are no more new subtransactions, the coordinator finally enters the final phase to commit the transaction.

The new subtransaction class `RWSubTransaction` extends the abstract Java class `SubTransaction` as defined in Figure 4.2. It adds a new operation `post-`

```
1  public class RWTransactionCoordinator {
2    List<SubTransaction> subtrans, derivedSubtrans;
3
4    public void start(){
5      getTimestamp();
6      submit(subtrans);
7    }
8
9    public handleVote(RWVote msg){
10     if(msg.vote == COMMIT){
11       derivedSubtrans.addAll(msg.derivedSubtrans);
12       if(allVoteCommit()) //is all subtransactions vote commit?
13         if(derivedSubtrans.size()!=0) { // is any new subtransactions generated?
14           submit(derivedSubtrans); //submit these new subtransactions
15           subtrans.addAll(derivedSubtrans); // add them to the transaction
16           derivedSubtrans.clear();
17         } else {
18           replicateTransactionState(); //for fault tolerance
19           commit(subtrans); //commit the transaction
20           returnResultToClient();
21         }
22     }
23     else if(msg.vote== ABORT)
24     { this.abort();} //abort the transaction
25   }
26 } //End of class RWTransactionCoordinator
27
28 public abstract class RWSubTransaction extends SubTransaction {
29   List<RWSubTransaction> derivedSubtrans;
30
31   public VoteResult run(){
32     return generateUpdates(); //update output member variables
33   }
34
35   VoteResult postrun(){ //identify affected index attributes
36     derivedSubtrans=generateIndexUpdates(super.dataToPut, super.dataToDelete);
37   }
38 }// End of class RWSubTransaction
```

Figure 5.6: The pseudocode of the extended read-write transaction implementation

`run()` to automatically generate updates on affected index attributes. These updates are stored as a set of `SubTransaction` instances into the variable `derivedSubtrans`, which will be returned to the coordinator if the vote is commit.

Fault Tolerance

CloudTPS must maintain strong data consistency even in the case of machine failures and network partitions. CloudTPS uses the same fault-tolerance mechanism as in our previous chapter. We therefore briefly mention the main concepts here again, and refer the reader to Chapter 4 for full details.

To execute transactions correctly all LTMs must agree on a consistent membership, as this is key to assigning data items to LTMs. Any membership change is therefore realized by a transaction across all LTMs.

During a network partition, LTMs are divided into multiple disjoint groups. In this case, we choose to guarantee ACID properties at the possible cost of a loss of availability. A partition may proceed accepting transactions provided that: (i) this partition is able to elect itself as the majority partition, of which the LTMs represent more than half of the previous membership; and (ii) its LTMs can recover the consistent states of all data items. In all other cases the system will reject incoming transactions until the partition is resolved and it fulfills the condition again. If a majority partition exists and manages to complete the recovery, the other LTMs will discard their entire states and rejoin when the network partition is resolved.

Recovering from an LTM failure implies that some surviving LTM fulfills the promises that the failed LTM made before failing. Such promises belong to two cases. In the first case, a coordinator initiated a transaction but failed before committing or aborting it. To recover such transactions, each LTM replicates its transaction states to one or more backup LTMs (chosen by consistent hashing through the system membership). If the coordinator fails, its backups have enough information to finish coordinating the ongoing transactions.

In the second case, a participant LTM voted commit for some read-write transactions but failed before it could checkpoint the update to the NoSQL data store. Here as well, each LTM replicates the state of its data items to one or more backup LTMs so that the backups can carry on the transactions and checkpoint all updates to the data store. Assuming that each transaction and data item has N backups in total, CloudTPS can guarantee the ACID properties under the simultaneous failure of up to N LTM servers.

An LTM server failure also results in the inaccessibility of the data items it was responsible for. Upon any change in membership it is therefore necessary to re-replicate data items to maintain the correct number of replicas. Following an LTM failure, CloudTPS can return to its normal mode of operation after all ongoing transactions have recovered, a new system membership has been created, and the relevant data items have been re-replicated.

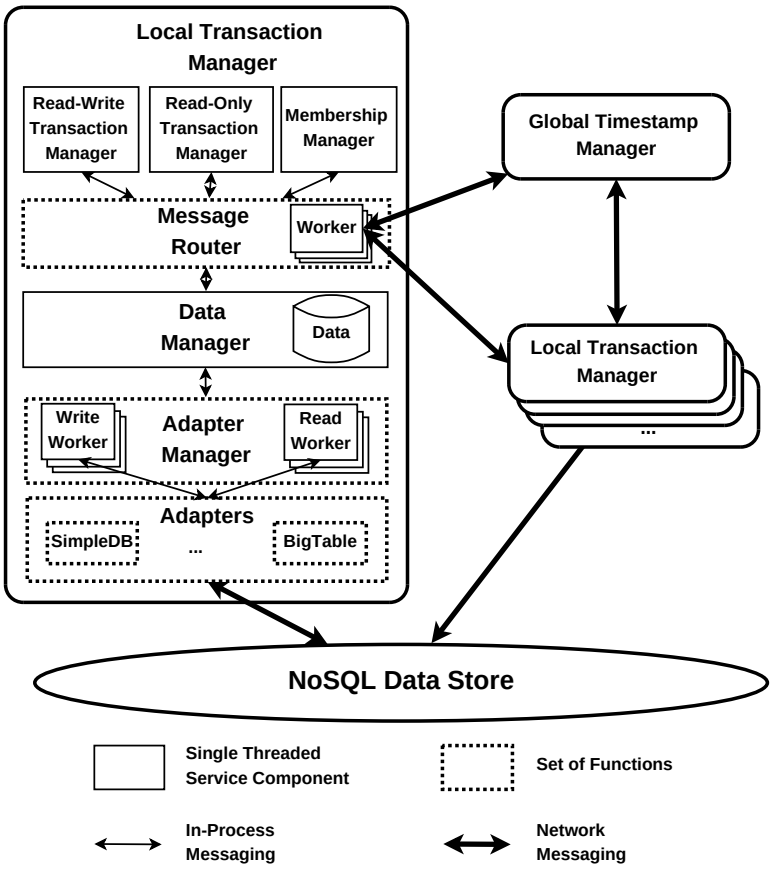


Figure 5.7: The internal architecture of CloudTPS

5.3 Implementation

5.3.1 CloudTPS Architecture

CloudTPS is composed of any number of identical LTMs. The internal architecture of an LTM, shown in Figure 5.7, consists of four core components running inside a Tomcat application server. This potentially allows to run CloudTPS on the same application servers as the Web application itself for improved communication performance.

Each LTM has a read-write and a read-only transaction manager to carry out the logics of transaction coordinators. The Read-Write transaction manager also maintains coordination with other LTMs in charge of backup replicas. To locate data items across CloudTPS, the membership manager maintains a consistent view of system membership. The data manager manages the in-memory copy of data and executes subtransactions in sequential order according to transaction timestamps.

The previous version-0.1 implementation of CloudTPS, as evaluated in Chapter 4, uses many levels of locks to manage concurrent access to various data struc-

tures such as tables, records and transaction states. Such design brings considerable performance overheads. This chapter is based on a re-implementation of CloudTPS, named as version 0.2, to remove the locks and improve LTM performance.

The new implementation of LTMs is inspired by the SEDA architecture [114]. Similarly to SEDA, CloudTPS is designed to handle large number of concurrent transactions with strong response time constraints. We implement the four core components as single-threaded self-contained service components. Each service component maintains a FIFO message queue for accepting messages. The service component continuously listens to its message queue, and handles incoming messages sequentially. Each service component is single-threaded to avoid the need to lock private data structures.

Executing a transaction consists of sending a sequence of messages between service components in the local or remote LTMs. Service components send messages via the message router, which is not an active service component but simply a passive set of Java classes. If the destination service component resides in the local LTM, the message router simply forwards the message to the corresponding message queue. Otherwise, the message router dispatches the message to one of its worker threads to perform network messaging. Each worker is implemented as a single-threaded service component which continuously waits for network messages to send. At the other end, the message receiver is implemented as a regular java servlet deployed in the application server.

Besides interacting with other components, the data manager also needs to access the underlying NoSQL data store to load and checkpoint data items. The data manager performs these operations by invoking the Adapter manager, which dispatches each operation to one of its own worker service components. The data items loading operations have higher priority than checkpointing. Loading a data item is in the critical path of a transaction which needs this item to progress, while checkpointing can be delayed with no impact on transaction latencies. The adapter manager therefore maintains two separate pools of workers for loading and checkpointing data items, respectively. Loading different data items can be done completely independently, so the adapter manager selects workers to load a data item in round-robin fashion. However, for checkpointing updates, the updates of multiple conflicting transactions on the same data item must be checkpointed in the order of transaction timestamps. To guarantee sequential order, the adapter manager dispatches all updates of the same data item to the same worker.

The workers of the adapter manager access the underlying NoSQL data store via adapters, which transform the logical data model of CloudTPS into the physical data model used by the NoSQL data store. The Web application does not need any adjustment depending on the type of underlying NoSQL data store. The current implementation supports SimpleDB and HBase [52] (an open source clone of Bigtable). Migrating CloudTPS between these two NoSQL data stores requires only to change the adapter configuration. One can also easily implement new adapters for other NoSQL data stores as we discuss in the next section.

```
1 public interface Adapter {
2     public void createTables(List<DataTable> schemas);
3     public void deleteTables(List<DataTable> schemas);
4     public int loadData(DataTable tableSchema, String strRowKey, HashMap<String,
          String> row);
5     public void checkpointOneItem(DataTable tableSchema, String rowKey, String[][]
          dataToPut, String[] dataToDelete);
6     public void commitAll();
7     public void deleteOneRow(DataTable tableSchema, String row);
8 }
```

Figure 5.8: The pseudocode of the Adapter interface

5.3.2 Adapters

CloudTPS aims to provide a uniform data-access overlay that allows Web applications to transparently access different NoSQL data stores. To be compatible with a broad range of NoSQL data stores, CloudTPS requires that table names include only letters and digits. The primary key and all attributes are of type string. All table names and attribute names are case-insensitive. CloudTPS automatically transforms all input parameters into lower-case.

Different NoSQL data stores have different rules for the definition of attribute names. For example, Bigtable requires the column-family name to be a prefix to the attribute name, while SimpleDB does not. For Bigtable, we store the application data in the `Data:` column family while the index attributes are in `Ref:`. For SimpleDB, the application data have the same attribute names as in queries. The index attributes are prepended with the prefix `Ref:` so as to differentiate them from the application data.

Some NoSQL data stores, such as SimpleDB, require to horizontally partition tables into multiple subtables for higher throughput. Even for NoSQL data stores that partition tables automatically, such manual partitioning is often still necessary for better load balancing. For example, HBase automatically partitions tables according to the size of the data rather than the load they receive. CloudTPS transparently splits tables horizontally into a configurable number of partitions. The physical table name for the NoSQL data store is then appended with the partition index.

Supporting a new data store in CloudTPS requires developing only a new Java class that implements the `Adapter` interface, as defined in Figure 5.8. The methods `createTables` and `deleteTables` implement transparent table partitioning. The `loadData` method receives the table schema and the primary key as input. It locates the physical table and retrieves the record by its primary key. It then transforms the loaded attribute names into logical names used in the queries of Web applications. The transformed record are returned in the parameter `row`. The `checkpointOneItem` method writes the updates of a committed transac-

tion back to the underlying NoSQL data store. When a read-write transaction requires deleting a record completely, the `deleteOneRow` method is used. These two methods can accumulate multiple updates and write back in batch to improve performance. LTMs periodically invoke method `commitAll` to write all accumulated updates immediately. This is simple, straightforward code. The two adapters currently implemented in CloudTPS are about 400 lines long each.

5.4 Evaluation

We now evaluate the performance and scalability of CloudTPS in three scenarios: micro- and macro-benchmarks, and a scenario with node failures and network partitions.

5.4.1 Experiment Setup

System Configuration

We execute CloudTPS on top of the same two different families of scalable data layers as the evaluation in Chapter 4: SimpleDB running in the Amazon cloud, and HBase running in our local DAS-3 cluster [52]. However, different from the evaluation in Chapter 4, we upgrade the software by using Tomcat v6.0.26 as application servers, HBase v0.20.4 as backend data store and CloudTPS v0.2 to handle join queries and read-write transactions in both platforms. The LTMs of CloudTPS and load generators are deployed on separate machines.

Throughput Measurement

Given a specific workload and number of LTMs, we measure the maximum sustainable throughput of CloudTPS under a constraint of response time. For the evaluations in DAS-3, we define a demanding response time constraint which imposes that 99% of transactions must return within 100 ms. DAS-3 assigns a physical machine for each LTM, and has low contention on other resources such as the network. On the other hand, in the public Amazon cloud, LTMs have to share a physical machine with other instances, and we have less control of the resources such as CPU, memory, network, etc. Furthermore, even multiple virtual instances of the exact same type may exhibit different performance behavior [36]. To prevent these interferences from disturbing our evaluation results, we relax the response time constraint for the evaluations in the Amazon cloud: 90% of transactions must return within 100 ms.

To determine the maximum sustainable throughput of CloudTPS, we perform several rounds of experiments with different request rates. The workload is generated by a configurable number of Emulated Browsers (EBs), each of which issues requests from one simulated user. Each EB waits for 1000 milliseconds on average between receiving a response and issuing the next transaction. Our evaluations

assume that the application load remains roughly constant. In each round, we configure different numbers of EBs and measure the throughput and response time of CloudTPS. We start with a small number of EBs, and increase the number of EBs until the response time of CloudTPS violates the response time constraint. Each round lasts 30 minutes.

Throughout the evaluation, we provision sufficient resources for clients and underlying NoSQL data stores, to ensure that CloudTPS remains the bottleneck of the system.

5.4.2 Microbenchmarks

We first study the performance of join queries and read-write transactions in CloudTPS using microbenchmarks.

Workload

Two criteria influence the performance of join queries in CloudTPS: the number of data items that they access, and the length of the critical execution path (i.e., the height of the query's tree-based representation). For example, a join query joining two tables has a critical execution path of one. We first evaluate the performance of CloudTPS under workloads consisting purely of join queries or read-write transactions with specific number of accessed records and length of critical execution path.

The microbenchmark uses only one table, where each record has a FK referring to another record in this table. We can therefore generate a join query with arbitrary length of its critical execution path by accessing the referenced record recursively. Given the length of the critical execution path, we can control the number of accessed records by defining the number of root records. We generate 10,000 records in this table.

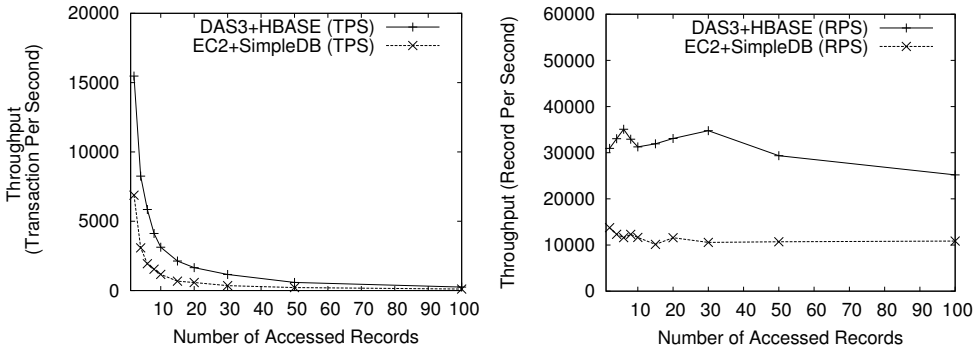
CloudTPS applies a cache replacement strategy to prevent LTMs from memory overflow when loading application data. In our evaluation with microbenchmarks, we configure the system such that the hit rate is 100%.

In this set of experiments, we deploy CloudTPS with 10 LTMs.

Join Queries

Here we study the performance of CloudTPS with join queries only. We first evaluate CloudTPS with join queries all having the same length of critical execution path of one, but access different numbers of data items. As shown in Figure 5.9(a), in both DAS-3 and EC2 platforms, when the number of accessed data items increase, the throughput in terms of transaction per second (TPS [2]) decreases dramatically. This is to be expected, since the complexity of join queries largely depends on the

[2] *Note that the TPS in the name of CloudTPS stands for: Transaction Processing System.*



(a) Expressed in transactions/second (b) Expressed in record accesses/second

Figure 5.9: Throughput of join queries with different number of accessed data items

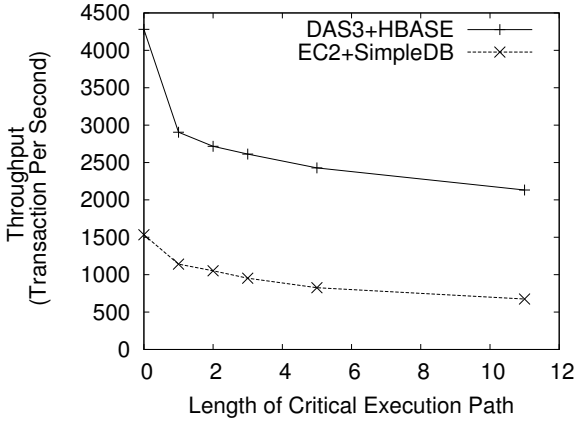
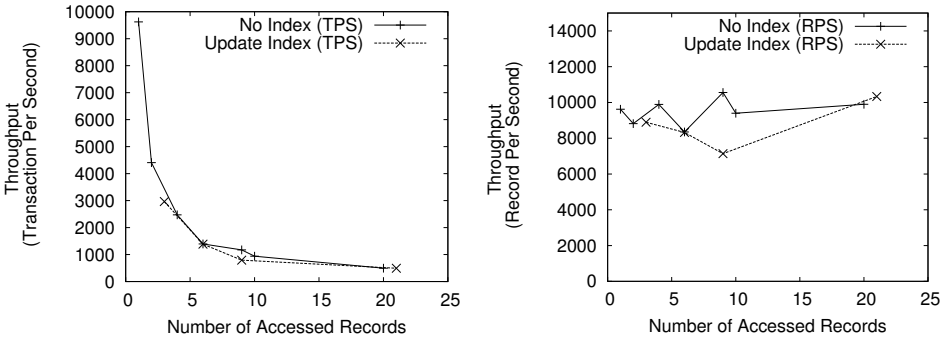


Figure 5.10: Throughput of join queries with different length of execution path (all queries access 12 data items)

number of data items they access. Figure 5.9(b) shows the same throughput expressed in numbers of accessed records per second. The result remains close to the ideal case, where the lines stay perfectly horizontal. We can also see that instances in DAS-3 perform approximately three times faster than medium High-CPU instances in EC2.

We then evaluate the system with join queries that access the same number of data items (12 items), but with different length of critical execution path. Figure 5.10 shows that as the length of the critical execution path increases, the maximum sustainable throughput decreases slightly. This is expected as longer execution paths increase the critical path of messages between LTMs, and therefore imply higher transaction latencies. To maintain the strict response time constraint, the system must reduce throughput.



(a) Transaction Per Second (b) Record Per Second

Figure 5.11: Throughput of read-write transactions with different number of accessed data items

Read-Write Transactions

We now study the performance of CloudTPS with a workload composed of read-write transactions (including read-write transactions that update index records). The updated index records are included in the count of accessed records of a transaction. We perform this evaluation on the DAS-3 platform.

Similar to join queries, as shown in Figure 5.11(a), the throughput in terms of TPS decreases dramatically when the number of accessed records increases. However, Figure 5.11(b) shows that the throughput in record accesses per second remains roughly constant. This shows that the performance bottleneck is the update operation of individual data items rather than the cost of the transaction itself.

We also note that in Figure 5.11, the line for transactions which only update data records and the line for transactions which also update indexes are very close to each other. This means the extra phase of updating index records does not degrade the system performance significantly. One only needs to pay the price of updating the extra index records.

5.4.3 Scalability Evaluation

TPC-W Web Application

We now evaluate the scalability of CloudTPS under a demanding workload derived from the TPC-W Web application [67]. TPC-W is an industry standard e-commerce benchmark that models an online bookstore similar to Amazon.com. It is important to note that TPC-W was originally designed and developed for relational databases. Therefore, it contains the same mix of join queries and read-write transactions as cloud-based applications would if their data store supported join queries. TPC-W contains 10 database tables.

Deploying TPC-W in CloudTPS requires no adaption to the database schema except converting all data types to string. This contrasts to the initial implemen-

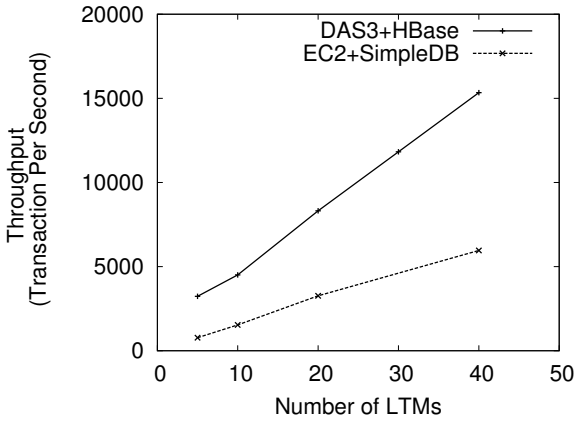


Figure 5.12: Scalability of CloudTPS under TPC-W workload

tation of CloudTPS, as described in Chapter 4, which requires denormalizing application data to transform join queries into primary-key queries. As in this chapter, CloudTPS supports join queries consistently on top of NoSQL databases, we kept all simple and complex queries unchanged, and merely translated them to CloudTPS's tree-based representation as discussed in Section 5.1.3.

TPC-W contains a secondary-key query which selects a customer record by its user name. CloudTPS therefore automatically creates an index table `indexOf_customerC_uname` referring to the SK `c_uname` of data table `customer`. This query is then rewritten into a join query across the two tables. The index table is the root table and the input user name is the primary key of the root record.

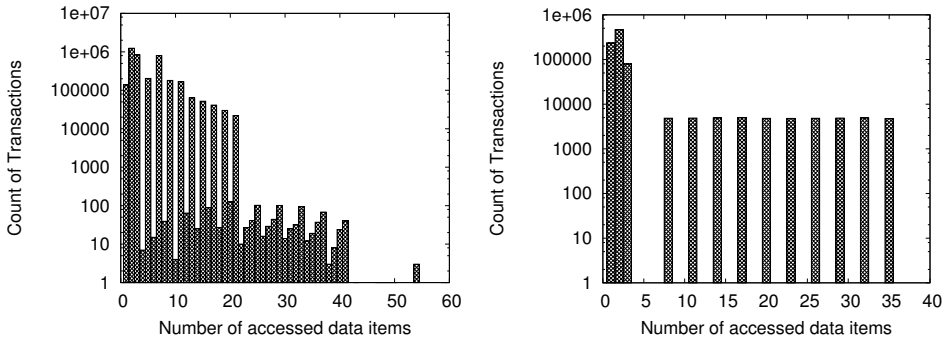
We derive a workload from TPC-W containing only join queries and read-write transactions. This workload excludes all simple primary-key read queries, which are the most common query type for Web applications. This creates a worst-case scenario for CloudTPS's performance and scalability.

We populate the TPC-W database with 144,000 customer records in table `Order` and 10,000 item records in table `Item`. We then populate the other tables according to the TPC-W benchmark requirements.

TPC-W continuously creates new shopping carts and orders. Each insert triggers one cache miss. On the other hand, as the size of affected data tables keeps increasing, this eventually results in more record evictions from the LTMs, which in turn potentially triggers more cache misses. During our scalability evaluation, we observe a hit rate around 80%.

Scalability Results

Figure 5.12 depicts the results of the scalability experiments in DAS-3 and the Amazon cloud. We can see that the overall system throughput grows linearly with



(a) Read-Only Transactions (b) Read-Write Transactions

Figure 5.13: Number of data items accessed by transactions (y axis is in log scale).

the number of LTMs. This means that CloudTPS can accommodate any increase of workload with a proportional number of compute resources.

In DAS-3, with 40 LTMs, CloudTPS achieves a maximum sustainable throughput of 15,340 transactions/second. For this experiment, we also use 30 machines to host HBase, 1 machine as timestamp manager and 8 clients. This configuration uses the complete DAS-3 Cluster, so we cannot push the experiment further. Figure 5.13 shows the distribution of the number of data items accessed per transactions under this configuration of 40 LTMs (note that the y axis is in log scale). On average, a read-only transaction accesses 4.92 data items and a read-write transaction accesses 2.96 data items. Within all input transactions, 82% transactions are join queries, and 18% are read-write transactions. As for the length of critical execution path, 33.3% of the read-only transactions have a length of one, while the other 66.7% have two. For read-write transactions, 84.2% of them need to update indexes, while the other 15.8% do not.

In EC2, with 40 LTMs, CloudTPS achieves a maximum sustainable throughput of 5,960 TPS. CloudTPS achieves three times better throughput in DAS-3 than in EC2 with High-CPU medium instances.

This evaluation shows that CloudTPS scales linearly under a demanding workload typical of a Web application designed with no restriction regarding join queries. We expect CloudTPS to continue scaling linearly with even larger numbers of LTMs.

Comparison with a Relational Database

We compare CloudTPS with PostgreSQL v.9.0 on DAS-3. The PostgreSQL setup contains one master and N slaves, using the binary-replication mechanism, which streams data changes, in the form of write-ahead logs, over the network to the slaves almost immediately on completion on the master [83]. We issue all read-write transactions on the master, and balance read-only queries across the slaves. When running CloudTPS, we count both CloudTPS and HBase nodes as database

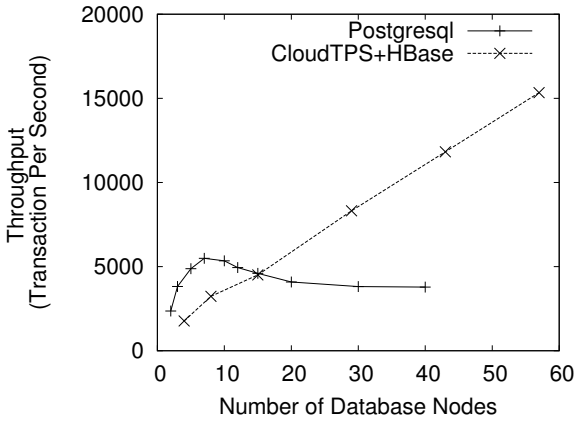


Figure 5.14: Scalability of CloudTPS vs. PostgreSQL

nodes. Running the same experiment in EC2 is not possible as we cannot measure the number of machines used by SimpleDB.

Figure 5.14 illustrates the differences between CloudTPS and a replicated relational database. In small systems, PostgreSQL significantly outperforms CloudTPS because each slave can execute read-only queries locally. PostgreSQL reaches a maximum throughput of 5493 TPS using one master and six slaves. However, at this point the master server becomes the bottleneck as it needs to process all update operations and send the binary operations to its slaves. The throughput eventually decreases because of the growing number of slaves to which it must send updates. On the other hand, CloudTPS starts with a modest throughput of 1770 TPS in its smallest configuration of 4 machines (two machines for CloudTPS and two machines for HBase). However, its throughput grows linearly with the number of database nodes, reaching a throughput of 15,340 TPS using 57 nodes (40 machines for CloudTPS and 17 machines for HBase). This clearly illustrates the scalability benefits of CloudTPS compared to a replicated relational database.

5.4.4 Tolerating Network Partitions and Machine Failures

Finally, we illustrate CloudTPS's behavior in the presence of machine failures and network partitions. In Chapter 4, we have shown that CloudTPS v0.1 recovers automatically from one network partition and one machine failure. Here we will show that CloudTPS v0.2, which adds support of consistent join queries, tolerates multiple network partitions and machine failures. We configure CloudTPS with 10 LTMs and then alternately create three network partitions and two machine failures. Each network partition lasts 1 minute. We run this experiment in DAS-3.

As shown in Figure 5.15, CloudTPS recovers automatically from all the three network partitions and two machine failures. The system shows similar recovery performance to the Chapter 4's evaluation of fault tolerance. In case of single-

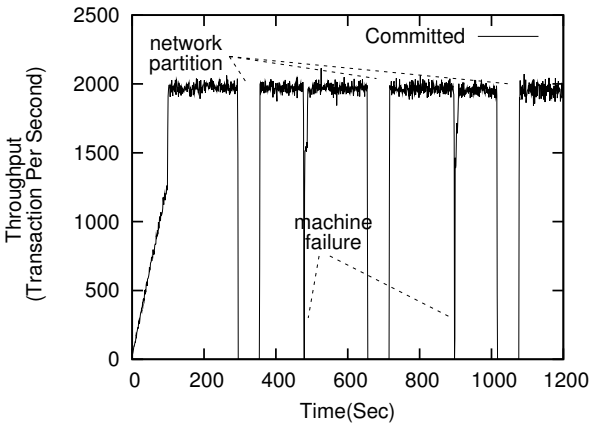


Figure 5.15: CloudTPS tolerates 3 network partitions and 2 machine failures

machine failures, CloudTPS recovers within about 14 seconds before failed transactions are recovered and the responsible data items of the failed LTM are re-replicated to new backup LTMs. On the other hand, for network partitions, as no data re-replication is necessary, CloudTPS recovers almost instantly after the network partition terminates. In all cases the transactional ACID properties are respected despite the failures.

5.5 Conclusion

NoSQL data stores are often praised for their good scalability and fault-tolerance properties. However, they are also criticized for the very restrictive set of query types they support. As a result, Web application programmers are obliged to design their applications according to the technical limitation of their data store, rather than according to good software engineering practice. This creates complex and error-prone code, especially when it comes to subtle issues such as data consistency under concurrent read/write queries.

This chapter proves that scalability, strong consistency and complex join queries do not necessarily contradict each other. CloudTPS exploits the fact that Web applications mostly use join queries that access a small fraction of the total available data set. By carefully designing algorithms such that only a small subset of the LTMs is involved in the processing of any particular transactions, we can implement strongly consistent join queries without compromising the original scalability properties of the NoSQL data store. We designed transactional protocols to address the needs of read-only join queries as well as read-write transactions which transparently update index values at runtime. The system scales linearly in our local cluster as well as in the Amazon cloud.

Chapter 6

Conclusions and Open Issues

Data management is essential for most Web applications. The requirements these applications put on their data store are two-fold: first, the data store should provide advanced functionalities such as strong data consistency and complex-query support to improve programmer efficiency in application development; second, the data store should provide properties such as scalability, elasticity and high availability to maintain reasonable performance under arbitrary workload while keeping operational costs under control. This thesis addressed the question: *is it possible to build a data store that accommodates all these requirements simultaneously*?

Web application developers have access to two main families of data stores: relational databases and NoSQL data stores. However, none of them meet all the aforementioned requirements. Relational databases support ACID transactions and complex queries, but implementing scalable and elastic applications using them requires considerable efforts from application programmers. In contrast, NoSQL data stores provide good properties of scalability and high availability, but lack support for strong data consistency and complex queries. Based on this observation, this thesis explored two different approaches to build a data store which provides all these properties.

In Chapter 3, we showed how one may scale Web applications using relational databases, while retaining their transactional properties. We presented a systematic approach to denormalize data into a number of independent data services. As each data service has a simplified data-access pattern, classical scaling techniques such as database replication and horizontal data partitioning can work effectively. The evaluation shows that the restructured application achieves linear scalability with increasing number of database instances. However, this approach requires significant manual efforts in restructuring the application, and in implementing data partitioning for scaling update-intensive data services. This formed the motivation for us to turn to NoSQL data stores which are inherently scalable and partition data automatically.

Chapter 4 and 5 discussed how one may extend NoSQL data stores to support ACID transactions and complex queries. We presented CloudTPS, a middle-

	Master-Slave SQL	NoSQL	Data Denormalization	CloudTPS
Supporting complex queries	Yes	No	Partially	Partially
Strong data consistency	Yes	No	Yes	Yes
Scalability	No	Yes	Yes	Yes
Elasticity	No	Yes	Yes	Yes
Fault tolerance	Yes	Yes	Yes	Yes
Supporting large data sizes	Yes	Yes	Yes	Yes
Work for the application developer	Low	Medium	High	Medium

Table 6.1: Supported properties of two main families of data stores and our approaches for Web applications.

ware system which stands between NoSQL data stores and their Web applications, and which handles transactions and complex queries such as join queries. The updates of committed transactions are later checkpointed back to the underlying NoSQL data store to achieve durability. CloudTPS achieves linear scalability as it partitions data automatically across any number of LTMs. Data are also replicated across multiple LTMs to tolerate machine failures and network partitions. In Chapter 4, we describe the implementation of transactional functionalities as well as system optimizations such as memory management to support large volumes of data. Chapter 5 extends the transaction commit protocol to allow the system to support complex queries such as join and secondary-key queries. To our knowledge, CloudTPS is currently the only NoSQL data store capable of supporting such queries without compromising scalability or strong data consistency.

6.1 Discussion

In Chapter 2, we described the desirable properties of an ideal data store for Web applications: performance, scalability, elasticity, supporting complex queries, strong data consistency, high availability and supporting large data sizes. One may therefore wonder: did we reach this goal? Table 6.1 shows the main properties of relational databases, NoSQL data stores as well as our two approaches in the context of Web applications.

Master-slave SQL databases can replicate an entire database to multiple slave databases. These systems support the same SQL language with rich semantics of complex queries and ACID transactions as centralized relational databases. Application developers do not need to modify any query or transaction of applications

that are based on relational databases. This approach can effectively scale Web applications with read-only or read-mostly workload: the system can address higher workload by balancing read-only queries to more slave databases. With multiple replicas of the complete database, this approach provides good fault tolerance properties. However, its main drawback is that it cannot scale under update-intensive workloads as all updates must be carried out on all replicas. Besides, this approach lacks elasticity, as adding a new node to the system requires replicating the full database. Such replication may cause significant costs, especially in the case of large data sets.

Relational databases also support partial replication and multi-master setups, but their improved performance comes at the cost of additional complexity for application developers.

NoSQL data stores replicate and partition data automatically. These data stores are elastic and provide good scalability for update workloads. Adding or removing a node causes only minor overheads. NoSQL data stores support large data sets as they partition the increasing amount of data across the available nodes. The partitioned data are also replicated to tolerate machine failures and network partitions. NoSQL data stores provide high availability even in the case of network partitions, but at the possible cost of compromising strong data consistency. This property might be acceptable for Web applications which prefer high availability over data consistency. However, programming correct applications using a weakly consistent data store is not an easy task. Furthermore, NoSQL data stores typically support only simple queries which either read or write a single data item. A few advanced systems support secondary-key queries, but only within a single table or table partition. Complex queries across multiple tables such as join queries are not supported. The lack of support for complex queries creates additional work to implement the same functionality using simple queries only.

Our approach of data denormalization, as presented in Chapter 3, provides good scalability for update-intensive Web applications, while maintaining the transactional properties of Web applications. Each data service has a simple data access pattern, which allows scaling techniques to work effectively. For example, one can scale read-only or read-mostly data services with master-slave database replication. Update-intensive data services are harder to build as they require implementing data partitioning. One might consider using CloudTPS to implement them though. Compared to using CloudTPS to scale Web applications directly, data denormalization has the potential to achieve better performance as one can scale data services with simplified workload more effectively compared to scaling the complete Web application. However, this approach imposes significant amount of work on application developers. The restructuring of an application requires one to re-implement lots of code as well as many queries of the application.

CloudTPS demonstrates the feasibility of supporting strong data consistency and complex queries in NoSQL data stores while preserving their good properties of scalability and elasticity. CloudTPS uses a non-standard format for representing queries, but building a SQL parser over it is quite easy [45]. This can greatly im-

prove programmer efficiency compared to regular NoSQL data stores. CloudTPS partitions data across LTMs in a similar way as NoSQL data stores to preserve their good properties of scalability and elasticity. It also replicates the partitioned data to tolerate machine failures and network partitions. CloudTPS supports large data sizes not only by partitioning data across LTMs, but also by applying memory management mechanisms such that only a small part of application data need to be maintained in LTMs.

As we can see from Table 6.1, among the four approaches, CloudTPS constitutes a step forward towards the initial goal. However, it still misses some important properties. First, CloudTPS lacks support of aggregate queries, which may access large number of data items. Implementing aggregate queries in CloudTPS as ACID transactions like join queries might compromise scalability of the system. Trading off data consistency for performance would be an effective approach to implement scalable aggregate queries. However, CloudTPS focuses on strong data consistency and does not allow relaxing data consistency. Second, the CAP theorem has shown that one must trade off between the two properties of data consistency and high availability in the case of network partitions. CloudTPS explicitly chooses to maintain strong data consistency without trading off for high availability. As discussed in Section 2.5, with this design, CloudTPS can provide strong data consistency properties, so that programmers can manage data consistency efficiently. However, it might be useful for data stores to allow programmers to make different choices for parts of their applications, possibly using application-specific knowledge about the semantics of data and queries.

Although CloudTPS does not achieve all the properties of the ideal data store, we believe it shows one possible direction for hosting Web application data.

6.2 Open Issues

This thesis presented approaches to enable scalable data management for Web applications. This is a very wide area, which cannot be fully covered in a single dissertation. We discuss here a number of important new research topics that this work has highlighted.

This thesis focuses on achieving linear scalability rather than improving absolute performance. However, it is still an interesting topic to investigate how to optimize the absolute performance of CloudTPS. Transactions and complex queries may access multiple data items located at different nodes. Such distributed operations are significantly less efficient than local transactions that access one node only. A workload-aware data placement may largely reduce the average number of nodes involved in transactions and thereby greatly improve performance. A few approaches have been proposed to analyze data access pattern and generate optimized data placements across distributed machines [31, 41]. However, these approaches are done offline in a centralized and unscalable manner. To cope with widely fluctuating and dynamic workloads, similar techniques have been applied for online

analysis and dynamic re-partitioning in the context of CloudTPS [103]. However, dynamic re-partitioning introduces overheads in locating data items and performing data re-organization, which may compromise the performance gain from executing more transactions locally. We therefore need sufficiently lightweight data localization mechanisms so that the gains of placing related data together exceed the costs.

Web applications may contain aggregate queries, which access large amount of data. Maintaining materialized views is an effective technique to support aggregate queries in centralized databases. It would be interesting to investigate how one may implement materialized views scalably in CloudTPS. To scale aggregate queries under update-intensive workload, one must partition each materialized view so that the load of read as well as writes on materialized views can be balanced across the LTMs. It is essential that updating a materialized view does not cause too much cost. There have been research efforts of supporting scalable aggregate queries in the context of CloudTPS [74]. This work extends Gupta's work on incremental view maintenance [49] and implements materialized views as CloudTPS tables, which are partitioned automatically. Although this work shows linear scalability, it supports only a subset of all possible aggregate queries. For example, this work cannot scale aggregation functions such as `MIN` and `MAX` as these two functions require inspecting all records of a table upon record deletion. Extending this work to all types of aggregation remains future research.

Bibliography

[1] 10gen. Mongodb, 2012. http://www.mongodb.org/.

[2] Bruno Abrahao, Virgilio Almeida, Jussara Almeida, Alex Zhang, Dirk Beyer, and Fereydoon Safai. Self-adaptive SLA-driven capacity management for internet services. In *Proceedings of the IEEE/IFIP Network Operations and Management Symposium*, April 2006.

[3] Marcos K. Aguilera, Wojciech Golab, and Mehul A. Shah. A practical scalable distributed B-tree. *Proceedings of the International Conference on Very Large Data Bases*, 1:598–609, August 2008.

[4] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 159–174, 2007.

[5] Amazon.com. Amazon DynamoDB., 2012. http://aws.amazon.com/dynamodb.

[6] Amazon.com. Amazon SimpleDB., 2012. http://aws.amazon.com/simpledb.

[7] Amazon.com. EC2 elastic compute cloud, 2012. http://aws.amazon.com/ec2.

[8] Khalil Amiri, Sanghyun Park, and Renu Tewari. DBProxy: a dynamic data cache for web applications. In *Proceedings of the International Conference on Data Engineering*, pages 821–831, 2003.

[9] Cristiana Amza, Emmanuel Cecchet, Anupam Chanda, Alan Cox, Sameh Elnikety, Romer Gil, Julie Marguerite, Karthick Rajamani, and Willy Zwaenepoel. Specification and implementation of dynamic web site benchmarks. In *Proceedings of the IEEE International Workshop on Workload Characterization*, November 2002.

[10] Tony Bain. Is the relational database doomed? *ReadWrite Enterprise*, 12, Feb 2009. http://www.readwriteweb.com/enterprise/2009/02/is-the-relational-database-doomed.php.

[11] Jason Baker, Chris Bond, James C. Corbett, JJ 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 *Proceeding of the Biennial Conference on Innovative Data Systems Research*, 2011.

[12] Philip A. Bernstein and Nathan Goodman. Timestamp-based algorithms for concurrency control in distributed database systems. In *Proceedings of the 6th international conference on Very Large Data Bases*, pages 285–300, 1980.

[13] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, 1981.

[14] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., 1987.

[15] Robert L. Bocchino, Vikram S. Adve, and Bradford L. Chamberlain. Software transactional memory for large scale clusters. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 247–258, 2008.

[16] Christof Bornhövd, Mehmet Altinel, C. Mohan, Hamid Pirahesh, and Berthold Reinwald. Adaptive database caching with DBCache. *Data Engineering*, 27(2):11–18, June 2004.

[17] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. Building a database on S3. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 251–264, 2008.

[18] Eric Brewer. CAP twelve years later: How the "rules" have changed. *IEEE Computer*, 45:23–29, 2012.

[19] Chris Bunch, Navraj Chohan, Chandra Krintz, Jovan Chohan, Jonathan Kupferman, Puneet Lakhina, Yiming Li, and Yoshihide Nomura. An evaluation of distributed datastores using the AppScale cloud platform. In *Proceedings of the IEEE International Conference on Cloud Computing*, pages 305–312, 2010.

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

[21] Pei Cao and Sandy Irani. Cost-aware www proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 1997.

[22] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. The information visualizer, an information workspace. In *Proceedings of the SIGCHI conference on Human factors in computing systems: Reaching through technology*, pages 181–186, 1991.

[23] Apache Cassandra. The Apache Cassandra project. `http://cassandra.apache.org/`.

[24] Rick Cattell. Scalable SQL and NoSQL data stores. *ACM SIGMOD Record*, 39:12–27, May 2011.

[25] Emmanuel Cecchet. C-JDBC: a middleware framework for database clustering. *Data Engineering*, 27(2):19–26, June 2004.

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

[27] 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 *Proceeding of the USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[28] Forrester Consulting. eCommerce Web site performance today: an updated look at consumer reaction to a poor online shopping experience, Aug 2009.

[29] 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. In *Proceedings of the international conference on Very Large Data Bases*, 2008.

[30] Italo Cunha, Jussara Almeida, Virgilio Almeida, and Marcos dos Santos. Self-adaptive capacity management for multi-tier virtualized environments. In *Proceedings of the International Symposium on Integrated Network Management*, May 2007.

[31] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: a workload-driven approach to database replication and partitioning. In *Proceedings of the international conference on Very Large Data Bases*, pages 48–57, September 2010.

[32] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Elastras: An elastic transactional data store in the cloud. In *Proceedings of the Workshop on Hot topics in cloud computing*, 2009.

[33] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-Store: a scalable data store for transactional multi key access in the cloud. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 163–174, 2010.

[34] DAS3. The Distributed ASCI Supercomputer 3, 2007. `http://www.cs.vu.nl/das3/`.

[35] 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 *Proceedings of the Symposium on Operating Systems Principles*, pages 205–220, 2007.

[36] Jiang Dejun, Guillaume Pierre, and Chi-Hung Chi. EC2 performance analysis for resource provisioning of service-oriented applications. In *Proceedings of the International conference on Service-oriented computing*, pages 197–207, 2009.

[37] Jiang Dejun, Guillaume Pierre, and Chi-Hung Chi. Autonomous resource provisioning for multi-service web applications. In *Proceedings of the International World-Wide Web Conference*, 2010.

[38] Jiang Dejun, Guillaume Pierre, and Chi-Hung Chi. Resource provisioning of Web applications in heterogeneous clouds. In *Proceedings of the 2nd USENIX Conference on Web Application Development*, 2011.

[39] Amr El Abbadi, Dale Skeen, and Flaviu Cristian. An efficient, fault-tolerant protocol for replicated data management. In *Proceedings of the 4th symposium on Principles of Database Systems*, pages 215–229, 1985.

[40] Jeremy Elson and Jon Howell. Handling flash crowds from your garage. In *Proceedings of the USENIX Annual Technical Conference*, pages 171–184, 2008.

[41] Yin fu Huang and Jyh her Chen. Fragment allocation in distributed database design. *Information Science and Engineering*, 17(3):491–506, May 2001.

[42] Lei Gao, Mike Dahlin, Amol Nayate, Jiandan Zheng, and Arun Iyengar. Application specific data replication for edge services. In *Proceedings of the 12th international World Wide Web conference*, May 2003.

[43] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

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

[45] Enes Göktaş. SQL translation layer for CloudTPS. *Bachelor Thesis, Vrije Universiteit Amsterdam*, 2011.

[46] Jim Gray. A conversation with Werner Vogels. *ACM Queue*, 4(4):14–22, May 2006.

[47] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[48] Tobias Groothuyse, Swaminathan Sivasubramanian, and Guillaume Pierre. GlobeTP: Template-based database replication for scalable web applications. In *Proceedings of the International conference on World Wide Web*, May 2007.

[49] Himanshu Gupta and Inderpal Singh Mumick. Incremental maintenance of aggregate and outerjoin expressions. *Information Systems*, 31(6):435–464, September 2006.

[50] Ramesh Gupta, Jayant Haritsa, and Krithi Ramamritham. Revisiting commit processing in distributed database systems. In *Proceedings of the ACM SIGMOD international conference on Management of Data*, pages 486–497, 1997.

[51] Derrick Harris. Facebook trapped in MySQL 'fate worse than death', Jul 2011. `http://gigaom.com/cloud/facebook-trapped-in-mysql-fate-worse-than-death/`.

[52] HBase. An open-source, distributed, column-oriented store modeled after the Google Bigtable paper, 2006. `http://hadoop.apache.org/hbase/`.

[53] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22th annual symposium on Principles of distributed computing*, pages 92–101, 2003.

[54] Svein-Olaf Hvasshovd, Øystein Torbjørnsen, Svein Erik Bratsberg, and Per Holager. The ClustRa telecom database: High availability, high throughput, and real-time response. In *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 469–477, 1995.

[55] Java TPC-W implementation distribution. `http://www.ece.wisc.edu/~pharm/tpcw.shtml`.

[56] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: a high-performance, distributed main memory transaction processing

system. In *Proceedings of the International conference on Very Large Data Bases*, 2008.

[57] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, 1997.

[58] Ladan Kazerouni and Kamalakar Karlapalem. Stepwise redesign of distributed relational databases. Technical Report HKUST-CS97-12, Hong Kong University of Science and Technology, Department of Computer Science, September 1997.

[59] Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *Proceedings of the international conference on Very Large Data Bases*, 2000.

[60] Christos Kotselidis, Mohammad Ansari, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. DiSTM: A Software Transactional Memory Framework for Clusters. In *Proceedings of the 37th International Conference on Parallel Processing*, pages 51–58, 2008.

[61] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Operating Systems Review*, 44:35–40, April 2010.

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

[63] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16:133–169, May 1998.

[64] Justin J. Levandoski, David Lomet, Mohamed F. Mokbel, and Kevin Keliang Zhao. Deuteronomy: Transaction support for cloud data. In *Proceeding of the 5th Biennial Conference on Innovative Data Systems Research*, 2011.

[65] Mei-Ling Liu, Divyakant AGRAWAL, and Amr El Abbad. The performance of two phase commit protocols in the presence of site failures. *Distributed Parallel Databases*, 6(2):157–182, 1998.

[66] Kaloian Manassiev, Madalin Mihailescu, and Cristiana Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *Proceedings of the 11th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 198–208, 2006.

[67] Daniel A. Menascé. TPC-W: A benchmark for e-commerce. *IEEE Internet Computing*, 6(3), 2002.

[68] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the ACM symposium on Principles of distributed computing*, pages 267–275, 1996.

[69] Microsoft.com. Microsoft SQL Azure Database., 2010. `http://www.microsoft.com/azure/data.mspx`.

[70] Robert B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pages 267–277, 1968.

[71] MySQL. MySQL cluster, 2011. `http://www.mysql.com/`.

[72] Shamkant Navathe, Kamalakar Karlapalem, and Minyoung Ra. A mixed fragmentation methodology for initial distributed database design. *Computer and Software Engineering*, 3(4), 1995.

[73] Shamkant Navathe and Minyoung Ra. Vertical partitioning for database design: a graphical algorithm. *SIGMOD Records*, 18(2):440–450, 1989.

[74] Plamen Nikolov. Aggregate queries in NoSQL cloud data stores. *Master Thesis, Vrije Universiteit Amsterdam*, 2011.

[75] Christopher Olston, Anastassia Ailamaki, Charles Garrod, Bruce M. Maggs, Amit Manjhi, and Todd C. Mowry. A scalability service for dynamic web applications. In *Proceedings of Conference on Innovative Data Systems Research*, January 2005.

[76] Christopher Olston, Amit Manjhi, Charles Garrod, Anastassia Ailamaki, Bruce M. Maggs, and Todd C. Mowry. A scalability service for dynamic web applications. In *Proceedings of the Conference on Innovative Data Systems Research*, pages 56–69, 2005.

[77] Democracy UK on Facebook. A snapshot of facebook in 2010, July 2011. `http://www.facebook.com/notes/democracy-uk-on-facebook/a-snapshot-of-facebook-in-2010/172769082761603`.

[78] Patrick O'Neil and Goetz Graefe. Multi-table joins through bitmapped join indices. In *Proceedings of the ACM SIGMOD international conference on Management of Data*, pages 8–11, September 1995.

[79] M. Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems*. Prentice-Hall, Inc., 2nd edition, February 1999.

[80] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceeding of the USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[81] Stefan Plantikow, Alexander Reinefeld, and Florian Schintke. Transactions for distributed wikis on structured overlays. In *Proceedings of the Distributed systems: operations and management*, 2007.

[82] Christian Plattner and Gustavo Alonso. Ganymed: Scalable replication for transactional web applications. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, October 2004.

[83] PostgreSQL.org. Binary replication tutorial, 2010. `http://wiki.postgresql.org/wiki/Binary_Replication_Tutorial`.

[84] Jun Rao, Eugene J. Shekita, and Sandeep Tata. Using paxos to build a scalable, consistent, and highly available datastore. In *Proceedings of the International conference on Very large data bases*, pages 243–254, 2011.

[85] RUBBoS: Bulletin board system benchmark. `http://jmob.objectweb.org/rubbos.html`.

[86] Michael Rys. Scalable SQL. *Communications of the ACM*, 54(6):48–53, June 2011.

[87] George Lawrence Sanders and Seung Kyoon Shin. Denormalization effects on performance of RDBMS. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, January 2001.

[88] Florian Schintke, Alexander Reinefeld, Seif Haridi, and Thorsten Schütt. Enhanced Paxos commit for transactions on DHTs. In *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 448–454, 2010.

[89] Gunter Schlageter. Optimistic methods for concurrency control in distributed database systems. In *Proceedings of the 7th international conference on Very Large Data Bases*, pages 125–130, 1981.

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

[91] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Scalaris: reliable transactional p2p key/value store. In *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, pages 41–48, 2008.

[92] Seung Kyoon Shin and George Lawrence Sanders. Denormalization strategies for data retrieval from data warehouses. *Decision Support Systems*, 42(1):267–282, October 2006.

[93] Randy Shoup and Dan Pritchett. The eBay architecture: Striking a balance between site stability, feature velocity, performance, and cost, Nov 2006. `http://www.addsimplicity.com/downloads/eBaySDForum2006-11-29.pdf`.

[94] Abraham Silberschatz, Henry Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, Inc., 5 edition, 2006.

[95] Swaminathan Sivasubramanian, Gustavo Alonso, Guillaume Pierre, and Maarten van Steen. GlobeDB: Autonomic data replication for web applications. In *Proceedings of the 15th international conference on World Wide Web*, May 2005.

[96] Swaminathan Sivasubramanian, Guillaume Pierre, Maarten van Steen, and Gustavo Alonso. GlobeCBC: Content-blind result caching for dynamic web applications. Technical Report IR-CS-022, Vrije Universiteit, Amsterdam, The Netherlands, June 2006.

[97] Swaminathan Sivasubramanian, Guillaume Pierre, Maarten van Steen, and Gustavo Alonso. Analysis of caching and replication strategies for web applications. *IEEE Internet Computing*, 11(1):60–66, January-February 2007.

[98] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM, pages 149–160, 2001.

[99] Michael Stonebraker. Concurrency control and consistency of multiple copies of data in distributed ingres. *IEEE Transactions on Software Engineering*, 5(3):188–194, 1979.

[100] Michael Stonebraker. Stonebraker on NoSQL and enterprises. *Communications of the ACM*, 54(8):10–11, August 2011.

[101] Michael Stonebraker and Rick Cattell. Ten rules for scalable performance in 'simple operation' datastores. *Communications of the ACM*, 54:72–80, jun 2011.

[102] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the international conference on Very Large Data Bases*, 2007.

[103] Björn Patrick Swift. Data placement in a scalable transactional data store. *Master Thesis, Vrije Universiteit Amsterdam*, 2012.

[104] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the 3rd international conference on on Parallel and distributed information systems*, pages 140–150, 1994.

[105] TPC-W frequently asked questions, question 2.10: "Why was the concept of atomic set of operations added and what are its requirements?", August 1999.

[106] Transaction Processing Performance Council. TPC benchmark C standard specification, revision 5, December 2006. `http://www.tpc.org/tpcc/`.

[107] Twitter. 200 million tweets per day, June 2011. `http://blog.twitter.com/2011/06/200-million-tweets-per-day.html`.

[108] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. Wikipedia workload analysis for decentralized hosting. *Elsevier Computer Networks*, 53(11):1830–1845, July 2009.

[109] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Transactions on Autonomous and Adaptive Systems*, 3:1:1–1:39, March 2008.

[110] Patrick Valduriez. Join indices. *ACM Transactions on Database Systems*, 12:218–246, June 1987.

[111] Werner Vogels. Data access patterns in the Amazon.com technology platform. In *Proceedings of the 33rd international conference on Very large Data Bases, Keynote Speech*, page 1, 2007.

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

[113] Voldemort. Project voldemort, December 2011. `http://project-voldemort.com/`.

[114] Matt Welsh, David Culler, and Eric Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 2001.

[115] Wikibench – the realistic web hosting benchmark. `http://www.wikibench.eu/`.

Summary

Data management is essential for most Web applications. The requirements these applications put on their data store are two-fold: first, the data store should provide advanced functionalities such as strong data consistency and complex-query support to improve programmer efficiency in application development; second, the data store should provide properties such as scalability, elasticity and high availability to maintain reasonable performance under arbitrary workload while keeping operational costs under control. This thesis addresses the question: *is it possible to build a data store that accommodates all these requirements simultaneously*?

Web application developers have access to two main families of data stores: relational databases and NoSQL data stores. However, none of them meet all the aforementioned requirements. Relational databases support ACID transactions and complex queries, but implementing scalable and elastic applications using them requires considerable efforts from application programmers. In contrast, NoSQL data stores provide good properties of scalability and high availability, but lack support for strong data consistency and complex queries. Based on this observation, this thesis explores two different approaches to build a data store which provides all these properties.

First, we study to which extent Web applications based on relational databases can be made elastic and scalable. Second, we explore how one may extend the existing NoSQL data stores with high-level database functionalities, such that their properties of scalability and elasticity are not compromised.

The Relational-Database Approach

In Chapter 3, we show how one may scale Web applications using relational databases, while retaining their transactional properties. We present a systematic approach to denormalize data into a number of independent data services, each of which having exclusive access to its private data store. This restructuring by itself does not lead to linear scalability directly. However, each of the data services has reduced workload complexity, which allows for a more effective application of the optimization techniques such as database replication, query caching and horizontal data partitioning, thus leading to significantly better scalability. For example, read-only data services can be scaled simply by database replication, while update-intensive data services can be scaled more effectively by horizontal data partition-

ing. Importantly, the restructuring does not imply any loss in terms of transactional or consistency properties.

The evaluation shows that the restructured application achieves linear scalability with increasing number of database instances. However, this approach requires significant manual efforts in restructuring the application, and in implementing data partitioning for scaling update-intensive data services. This formed the motivation for us to turn to NoSQL data stores which are inherently scalable and partition data automatically.

The NoSQL-Data-Store Approach

In Chapter 4 and 5, we explore how one may extend the existing NoSQL data stores with support of ACID transactions and complex queries, such that their properties of scalability and elasticity are not compromised.

We implement these missing features of NoSQL data stores in a middleware layer, called CloudTPS which sits between the application and its data store. Our prototype creates a temporary copy of the application data in the memory of its participant machines. All the added functionalities, such as transactions and join queries, operate directly on this copy of the data. All updates are checkpointed back to the underlying data store in a lazy fashion such that users observe strong ACID properties even in the case of machine failures or network partitions. CloudTPS follows the system model of typical NoSQL data stores, which automatically manages data partitioning across any number of machines. CloudTPS also replicates data items to a specified number of machines. When encountering machine failures or network partitions, CloudTPS can recover automatically without compromising data consistency. In Chapter 4, we describe the implementation of transactional functionalities as well as system optimizations such as memory management to support large volumes of data. Chapter 5 extends the transaction commit protocol to allow the system to support complex queries such as join and secondary-key queries. To our knowledge, CloudTPS is currently the only NoSQL data store capable of supporting such queries without compromising scalability or strong data consistency.

Samenvatting

Schaalbare Datamanagement voor Webapplicaties

Datamanagement is essentieel voor de meeste webapplicaties. De eisen die deze applicaties aan hun dataopslag stellen zijn tweeledig: Ten eerste moet de dataopslag voorzien in geavanceerde functionaliteit zoals sterke dataconsistentie en ondersteuning voor complexe queries, ter verbetering van de efficiëntie van programmeurs gedurende de applicatie ontwikkeling. Ten tweede moet de dataopslag schaalbaar en elastisch zijn en een hoge beschikbaarheid hebben om goede prestaties te kunnen leveren onder uiteenlopende soorten werkdruk terwijl de kosten beheersbaar blijven. Dit proefschrift richt zich op de vraag: *is het mogelijk om een dataopslag te ontwikkeling die aan al deze eisen tegelijkertijd kan voldoen?*

De ontwikkelaars van webapplicaties kunnen kiezen uit twee soorten dataopslag: relationele databases of NoSQL dataopslag. Echter voorziet geen van beide in alle eerdergenoemde vereisten. Relationele databases ondersteunen ACID transacties en complexe queries, maar het ontwikkelen van schaalbare en elastische applicaties die hier gebruik van maken, vereist een aanzienlijke inspanning van de applicatie programmeurs. Daarentegen kan NoSQL dataopslag wel voorzien in schaalbaarheid en hoge beschikbaarheid, maar de ondersteuning voor sterke dataconsistentie en complexe queries ontbreekt. Gebaseerd op deze observatie verkent dit proefschrift twee verschillende aanpakken om een dataopslag te ontwikkelen die wel aan alle eisen kan voldoen.

Eerst bestuderen we hoe webapplicaties die gebaseerd zijn op relationele databases elastisch en schaalbaar gemaakt kunnen worden. Ten tweede, verkennen we hoe bestaande NoSQL dataopslag uitgebreid kan worden met geavanceerde database functionaliteit zodat eigenschappen als schaalbaarheid en elasticiteit niet verloren gaan.

De relationele database aanpak

In hoofdstuk 3 laten we zien hoe webapplicaties die gebruik maken van relationele databases opgeschaald kunnen worden, terwijl de transactionele eigenschappen bewaard blijven. We presenteren een systematische aanpak om data te denormaliseren in aan aantal onafhankelijke dataservices die elk exclusieve toegang

hebben tot hun eigen dataopslag. Deze herstructurering leidt nog niet direct tot lineaire schaalbaarheid. Elke dataservice heeft echter een minder complexe werklast, wat effectievere toepassing van optimalisatie technieken zoals database replicatie, query caching en horizontale data partitionering mogelijk maakt, wat leidt tot een significant betere schaalbaarheid. Read-only dataservices kunnen bijvoorbeeld gemakkelijk opgeschaald worden door middel van database replicatie, terwijl update-intensieve dataservices gemakkelijker opgeschaald kunnen worden met behulp van horizontale data partitionering. Belangrijk is dat door deze herstructurering de transactionele en consistentie eigenschappen niet verloren gaan.

De evaluatie laat zien dat de geherstructureerde applicaties lineaire schaalbaarheid vertonen, met een toenemend aantal database instanties. Deze aanpak vereist echter een significante hoeveelheid handmatig werk in het herstructureren van de applicatie en in het implementeren van de data partitionering voor het opschalen van update-intensieve dataservices. Dit vormde de motivatie om ons te wenden tot NoSQL dataopslag die inherent schaalbaar is en automatisch data partitioneert.

De NoSQL dataopslag aanpak

In hoofdstuk 4 en 5 verkennen we hoe bestaande NoSQL dataopslag met ondersteuning voor ACID transacties en complexe queries uitgebreid kan worden zonder dat de schaalbaarheid of elasticiteit in het gedrag komt.

We implementeren deze ontbrekende eigenschappen van NoSQL dataopslag in een middleware laag genaamd CloudTPS, die zich tussen de applicatie en de dataopslag bevindt. Ons prototype creëert een tijdelijke kopie van de applicatie data in het geheugen van de deelnemende machines. Alle toegevoegde functionaliteit, zoals transacties en join queries, opereren direct op deze kopie van de data. Alle updates worden met behulp van checkpoints naar de onderliggende dataopslag gecommuniceerd op een luie manier zodat de gebruiker sterke ACID eigenschappen ervaart, zelfs wanneer er een machine uitvalt of het netwerk partitioneert. CloudTPS volgt het model van typische NoSQL dataopslag die automatisch data partitionering over een willekeurig aantal machines afhandelt. CloudTPS repliceert ook data elementen naar een gespecificeerd aantal machines. Wanneer machine uitvallen of netwerk partitioneringen aan de orde zijn kan CloudTPS automatisch herstellen zonder dat data consistentie in het gedrang komt. In hoofdstuk 4 beschrijven we de implementatie van transactionele functionaliteit evenals systeem optimalisaties, waaronder geheugen management om grote hoeveelheden data aan te kunnen. Hoofdstuk 5 breidt het transactie commit protocol uit om de ondersteuning voor complexe queries zoals join en secondary-key queries mogelijk te maken. Voor zover ons bekend is CloudTPS op dit moment de enige NoSQL dataopslag die in staat is deze queries te ondersteunen zonder dat de schaalbaarheid of sterke data consistentie verloren gaat.