Architectures for Elastic Database Services

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

SIMON MANFRED LOESING

M. sc. in Computer Science, ETH Zurich

born on 21.02.1982

citizen of Germany

accepted on the recommendation of
Prof. Dr. Donald Kossmann (ETH Zurich), examiner
Prof. Dr. Gustavo Alonso (ETH Zurich), co-examiner
Prof. Dr. Alfons Kemper (TU Munich), co-examiner

2015
Abstract

The emergence of cloud technologies has caused a profound disruption in the market for data management systems. The access to (virtually) unlimited computing resources has considerably changed the requirements towards data processing and storage systems. Specifically, properties such as scalability, elasticity, cost-efficiency, and ease of deployment are increasingly becoming critical success factors for many businesses. In response to these new demands, traditional relational database systems have been challenged by various new large-scale data services as well as alternative storage systems. This trend has resulted in a jungle of heterogeneous data processing solutions all providing different features and guarantees. This dissertation aims at understanding the limits and the trade-offs in providing traditional database features in the context of cloud computing.

To explore the design space, we assess alternative architectures for distributed transaction processing and highlight key options with regard to scalability, flexibility, and robustness. In particular, we examine what architectural properties are required in order to meet the requirements of dynamic cloud environments. Our analysis is extended by a comprehensive evaluation of existing commercial cloud services that have adopted these architectures. To that end, we have developed a new benchmarking methodology that takes into account the specific characteristics of cloud data services. That is, we validate if the promises of unlimited throughput and reduced cost can be fulfilled. The results are surprising in several ways. Most importantly, it seems that all major vendors have adopted a different architecture for their services. As a result, the cost and performance vary significantly depending on the workload.

The limitations of current cloud data processing solutions with regard to database workloads motivated the development of a new system: Tell. Tell is a distributed relational database designed for scalable and elastic transaction processing. It implements an architecture design based on two principles: We decouple query processing and transaction
management from data storage, and we share data across query processing nodes. The combination of these design choices enables elasticity and is the foundation for cost-efficient database infrastructures. Moreover, the separation of concerns allows for mixed workloads. That is, transactional and analytical workloads can be performed simultaneously on separate processing nodes but accessing a single, shared copy of the data. As a drawback, sharing data among multiple database nodes causes substantial synchronization overhead. To overcome this limitation, we present techniques for scalable transaction processing in shared-data environments. Specifically, we provide mechanisms for efficient data access, concurrency control, and data buffering. The implementation of these techniques is facilitated by new hardware trends and provides scalable OLTP performance.

Architectures that share data have been adopted by several data management systems in the past. A prominent example is Apache Hadoop. Although these systems are widely used, performing database transactions on shared data has not been a main focus as of yet. To bridge this gap, we provide an in-depth evaluation of alternative concurrency control mechanisms for shared-data systems, such as Tell. We adapt established techniques such as snapshot isolation, timestamp ordering, and locking to operate in a distributed environment on top of a shared store. Each technique is designed to minimize synchronization overhead while providing full ACID (Atomicity, Consistency, Isolation, Durability) semantics. Tell is used as a reference system to implement and evaluate the alternative approaches.

This dissertation contributes to our scientific understanding of how to build elastic distributed database services for cloud infrastructures. It provides an essential piece in the puzzle of developing next generation data processing solutions.
Zusammenfassung


Die Grenzen bestehender Cloud-Datenverarbeitungslösungen in Bezug auf Datenbankanfragen hat die Entwicklung eines neuen Systems angeregt: Tell. Tell ist eine verteilte, re-


Acknowledgments

This dissertation would not have been possible without the help of those around me. I first would like to express profound gratitude to my advisor Prof. Dr. Donald Kossmann for his invaluable support and continuous guidance throughout my Ph.D. studies. I feel all the more thankful and honored for his trust and encouragement over the years. I would like to extend my sincere gratitude to Prof. Dr. Gustavo Alonso for his numerous advice and for being a great role model researcher to me. I am also deeply thankful to Prof. Dr. Alfons Kemper for accepting to be part of my examination committee.

The system and the techniques presented in this work are not a one-man project. Besides my advisor, who guided this work, I am sincerely grateful to Markus Pilman and Thomas Etter. Both contributed large amounts of time, code, and many ideas. Working with them was an enriching and unique experience.

I want to extend my gratitude to all the present and past colleagues in the Systems Group who made the last years a memorable and unforgettable experience. Special thanks go to my friends and colleagues Tim, Kyumars, Martin, Philipp, Lukas, Tudor, Louis, Georgios, Lucas, Anja, and Claude for many fruitful discussions and extraordinary moments in and out of the office.

Last but far from least, heartfelt thanks to my family for their endless support. Above all, I am deeply indebted to my patient and loving girlfriend Bianca.
Contents

1 Introduction ................................................................. 1
  1.1 Motivation ............................................................... 1
  1.2 Problem Statement .................................................... 3
  1.3 Contributions .......................................................... 5
  1.4 Structure of the Dissertation ........................................ 7

2 State-Of-The-Art ........................................................... 9
  2.1 Technology Trends ...................................................... 10
    2.1.1 Warehouse-Scale Computing ..................................... 10
    2.1.2 Scale-Out Data Processing Systems ............................ 12
  2.2 Distributed Database Architectures .................................. 14
    2.2.1 Classic .............................................................. 14
    2.2.2 Partitioning ........................................................ 15
    2.2.3 Shared-Data ......................................................... 18
    2.2.4 Design Options ..................................................... 20
      2.2.4.1 Replication ....................................................... 20
      2.2.4.2 Caching ........................................................ 21
  2.3 Cloud Data Services .................................................... 22
    2.3.1 Amazon (AWS) ....................................................... 24
      2.3.1.1 AWS MySQL .................................................... 24
2.3.1.2 AWS MySQL/R ........................................ 24
2.3.1.3 AWS RDS ........................................... 25
2.3.1.4 AWS SimpleDB ..................................... 26
2.3.1.5 AWS S3 ............................................. 27
2.3.2 Google ................................................. 27
2.3.3 Microsoft ............................................... 28
2.4 Concluding Remarks ....................................... 29

3 Evaluation of Cloud Data Services .......................... 31
3.1 Methodology for a Cloud Benchmark ...................... 31
  3.1.1 Requirements ......................................... 32
  3.1.2 Limitations of the TPC-W ............................ 33
  3.1.3 A Benchmark for the Cloud ......................... 35
     3.1.3.1 Configurations .................................. 35
     3.1.3.2 Metrics and Experiments ......................... 36
  3.1.4 Related Work ......................................... 40
3.2 Experiments and Results .................................. 41
  3.2.1 Implementation and Environment ..................... 41
  3.2.2 Big Picture .......................................... 44
  3.2.3 Scale-Out ............................................ 46
  3.2.4 Cost .................................................. 49
     3.2.4.1 Cost/WIPS ...................................... 49
     3.2.4.2 Cost per Day .................................... 50
     3.2.4.3 Cost Analysis .................................... 52
  3.2.5 Scale-Up ............................................. 56
  3.2.6 Bulk-Loading ........................................ 56
3.3 Concluding Remarks ....................................... 57
4 Tell: A Distributed Shared-Data Database 59

4.1 Design Principles ........................................... 60
    4.1.1 Shared Data ........................................ 60
    4.1.2 Decoupling Query Processing and Storage .......... 60
    4.1.3 In-Memory Storage ................................... 61
    4.1.4 ACID Transactions .................................... 62
    4.1.5 Complex Queries ..................................... 62

4.2 Technical Challenges ...................................... 62
    4.2.1 Data Access ......................................... 63
    4.2.2 Concurrency Control .................................. 64
    4.2.3 Limitations ......................................... 64

4.3 Related Work .............................................. 65

4.4 Transaction Processing ................................... 67
    4.4.1 Distributed Snapshot Isolation ...................... 68
    4.4.2 The Commit Manager ................................ 70
    4.4.3 Life-Cycle of a Transaction ......................... 72
    4.4.4 Recovery and Fail-Over ............................... 72
        4.4.4.1 Failure of Processing Node ................. 72
        4.4.4.2 Failure of a Storage Node ............... 73
        4.4.4.3 Failure of a Commit Manager ............ 74
    4.4.5 Discussion of Alternatives ......................... 74

4.5 Data Access and Storage ................................ 75
    4.5.1 Data Mapping ....................................... 75
    4.5.2 Mixed Workloads ................................... 77
    4.5.3 Latch-Free Index Structures ......................... 77
        4.5.3.1 B+Tree Index ............................... 78
        4.5.3.2 Design Considerations .................... 78
## Contents

4.5.4 Garbage Collection ........................................... 79
4.6 Buffering Strategies ........................................... 80
  4.6.1 Transaction Buffer .......................................... 80
  4.6.2 Shared Record Buffer ....................................... 80
  4.6.3 Shared Buffer with Version Set Synchronization .......... 82
4.7 Experimental Evaluation ....................................... 83
  4.7.1 Implementation and Environment ......................... 83
  4.7.2 Benchmarks ................................................ 84
  4.7.3 Scale-Out .................................................. 85
    4.7.3.1 Processing Nodes ...................................... 85
    4.7.3.2 Storage Nodes ......................................... 87
    4.7.3.3 Commit Managers ...................................... 87
  4.7.4 Comparison to Partitioned Databases ..................... 88
  4.7.5 Comparison to Shared-Data Databases .................... 92
  4.7.6 Network .................................................... 92
  4.7.7 Buffering Strategies ...................................... 93
4.8 Concluding Remarks ........................................... 94

5 Concurrency Control Techniques .................................. 95

  5.1 Snapshot Isolation ............................................ 97
  5.2 Timestamp Ordering .......................................... 97
    5.2.1 Life-Cycle of a Transaction ............................ 99
    5.2.2 Recovery and Fail-Over .................................. 99
    5.2.3 Discussion of Alternatives .............................. 100
  5.3 Centralized Locking .......................................... 100
    5.3.1 Centralized Locking Service ............................ 101
    5.3.2 Life-Cycle of a Transaction ............................ 102
    5.3.3 Recovery and Fail-Over .................................. 102
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4</td>
<td>Data-Coupled Locking</td>
<td>103</td>
</tr>
<tr>
<td></td>
<td>5.4.1 Life-Cycle of a Transaction</td>
<td>104</td>
</tr>
<tr>
<td></td>
<td>5.4.2 Recovery and Fail-Over</td>
<td>104</td>
</tr>
<tr>
<td>5.5</td>
<td>Experimental Evaluation</td>
<td>105</td>
</tr>
<tr>
<td></td>
<td>5.5.1 Benchmark, Environment, and Implementation</td>
<td>105</td>
</tr>
<tr>
<td></td>
<td>5.5.2 Write-Intensive Workloads</td>
<td>106</td>
</tr>
<tr>
<td></td>
<td>5.5.3 Read-Intensive Workloads</td>
<td>108</td>
</tr>
<tr>
<td></td>
<td>5.5.4 Resource Utilization</td>
<td>109</td>
</tr>
<tr>
<td></td>
<td>5.5.5 Replication</td>
<td>110</td>
</tr>
<tr>
<td></td>
<td>5.5.6 Network</td>
<td>112</td>
</tr>
<tr>
<td>5.6</td>
<td>Related Work</td>
<td>112</td>
</tr>
<tr>
<td>5.7</td>
<td>Concluding Remarks</td>
<td>115</td>
</tr>
<tr>
<td>6</td>
<td>Conclusions</td>
<td>117</td>
</tr>
<tr>
<td></td>
<td>6.1 Summary</td>
<td>118</td>
</tr>
<tr>
<td></td>
<td>6.2 Directions for Future Work</td>
<td>119</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Motivation

In recent years, cloud computing has evolved from a hyped trend to a multi-billion business. A multitude of cloud services have emerged and changed the way we perceive and operate with IT resources [SPN+14]. The range of available services varies from general purpose infrastructure services, such as hosted servers or storage space, to specific software solutions for communication, personal data, and others. The growing success of the cloud is based on several key benefits. First, cloud computing provides (virtually) unlimited amounts of resources. It enables access to large-scale infrastructures that span data-centers worldwide and thus facilitates highly scalable solutions. Second, cloud computing enables software systems to dynamically react to changing operational conditions and optimize resource utilization. Resources can be acquired on-demand within short amounts of time and released once they are not required anymore. As a result, it is no longer necessary to maintain an over-provisioned infrastructure to handle peak load scenarios. Instead, a cluster can be scaled out (i.e., add and integrate new nodes) automatically the moment utilization reaches critical levels. Third, cloud computing promises significant cost savings. The pay-as-you-go business model, adopted by most vendors, ensures that customers are only charged for the amount and the time resources are utilized. As a consequence, the potential to reduce cost in comparison to buying and maintaining a private infrastructure is considerable.
Chapter 1. Introduction

Cloud technologies increase flexibility with regard to resource utilization. However, software systems, in particular databases, need to be adequately designed in order to fully benefit from cloud infrastructures [Rys11]. In the following, we describe key properties introduced by the cloud computing era that have become fundamental design requirements for modern data management systems:

- **Scalability**: Cloud computing enables to process any load at any given time without being limited by hardware resources. A cloud-enabled system should be able to scale out to dozens or hundreds of nodes distributed across multiple data-centers.

- **Elasticity**: Elasticity describes the ability to dynamically adjust the size of a cluster if needed. That is, resources (nodes) can be added or removed on-demand with minimal effort and without interrupting operation. Moreover, elastic systems facilitate resource management and allow for operating on many low-cost commodity servers instead of expensive high-end machines.

- **Fault-Tolerance**: State-of-the-art services are expected to be highly available. Thus, systems should be able to tolerate node and network failures seamlessly. Ideally, every request succeeds, and the users do not even notice failures.

- **Operational agility**: Operational agility characterizes all properties that facilitate the setup and the adaptability of a system. In order to enable large-scale deployments, systems should minimize the need for manual user intervention. Features such as intuitive configuration, simple APIs, self-management, and automatic failure handling are essential.

Relational database management systems (RDBMS) have dominated the landscape for data processing systems in the last decades. RDBMS support complex queries (SQL) and are optimized towards performing concurrent transactions while ensuring data integrity. Consequently, they provide powerful, established, and widely accepted mechanisms for data processing [HS05]. However, RDBMS are also known to be complex monolithic entities that excel on dedicated and powerful hardware, but they are not designed with regard to elasticity and agility [SMA+07, Sel08]. It is the rigidity towards the new requirements of cloud computing that has caused the emergence of an alternative breed of storage systems, referred to as NoSQL (“Not Only SQL”) [Moh13].
NoSQL systems [DHJ+07, CDG+08, CRS+08] are designed to operate in cloud infrastructures. They simplify or sacrifice features from relational databases such as ACID transactions and rich query support in order to improve scalability, availability, and elasticity. The development of NoSQL has been substantially influenced by the CAP theorem [GL02]. The CAP theorem states that it is impossible for a distributed service to provide the following three guarantees: consistency, availability, and partition-tolerance. Only two out of three guarantees are possible in a given system, and many NoSQL stores lower consistency guarantees in favor of availability and partition-tolerance [Cat11]. Although NoSQL systems meet the requirements with regard to cloud environments, they introduce new limitations. By stripping away RDBMS features, many complex tasks, previously performed by a database, must now be handled in the application layer. For instance, in an eventually consistent store, the application developer must ensure that data updates are not lost and data integrity is preserved [Vog09]. Consequently, NoSQL systems complicate the implementation of many use-cases and are not the silver bullet for large-scale data processing.

Although traditional databases and NoSQL stores have limitations, recent efforts in the database community are geared towards building systems that combine the best of both worlds [SVS+13, Vol14]. This trend is often referred to as NewSQL [Sto11]. The vision is to build data processing systems that match NoSQL stores with regard to scalability and elasticity, while at the same time provide the powerful features of relational databases, namely the relational data model, a SQL interface, and ACID transactions. Going beyond existing work, this dissertation discusses and addresses the challenges of NewSQL.

### 1.2 Problem Statement

Several systems [Vol14, Nuo14, Chu14] have attempted to orchestrate traditional relational databases to run in the cloud. These systems use a common approach to scale out databases. That is, they horizontally partition data among several database instances (also called Sharding) and forward queries to the corresponding instance(s) that owns the required data to answer the queries. We refer to this type of architecture as partitioned database. Partitioned databases are very scalable as long as transactions only access a single instance. The moment a transaction spreads across multiple instances, a consensus protocol, such as Two-Phase Commit (2PC), is required to reach agreement on commit or abort. Distributed transactions involve overhead and limit system scal-
ability. Although many solutions have been proposed to improve partitioning, optimize transactional execution, and minimize the number of distributed transaction in partitioned databases [KKN+08, JAM10, TDW+12], the architecture itself has proven to be inflexible and difficult to manage in large-scale deployments [LFWZ09, Col11]. A major problem in partitioned databases is the coupling of processing resources to data partitions. As a result, the maximum query load on a particular partition is limited by the hardware resources of the node where the partition is located. Furthermore, the coupling complicates scale-out as transaction execution and simultaneous data repartitioning (i.e., move data to another node) require close coordination [SI09]. These problems are an inherent obstacle to agility and elasticity.

This dissertation addresses these limitations and supports the hypothesis that it is possible to build a database system that combines the benefits of traditional relational databases and the advantages of NoSQL stores without sacrificing performance. As a solution, we propose an alternative architecture design for distributed transaction processing that enables elasticity and operational agility. Our architecture is based on two principles: The first principle is data sharing across all database instances. In contrast to partitioned databases, database instances do not exclusively own a partition but can access the whole data and execute any query. Hence, transactions can be entirely performed on a single instance and consensus protocols are not necessary. The second principle is decoupling query processing and transaction management from data storage. While in traditional RDBMS engines the two are tightly coupled, we logically separate them into two architectural layers. This break-up simplifies deployment and enables elasticity as both layers can be scaled out independently. Applying both principles results in a two-tier architecture in which database instances operate on top of a shared record store. Hence, the name shared-data architecture.

The shared-data architecture has recently shown to scale in the context of online analytical processing (OLAP). For instance, it is the foundation behind the success of Apache Hadoop [Apa14a]. We use the same elemental principles to allow for scalable online transaction processing (OLTP) and implement a distributed relational database system called Tell. In a shared-data system, such as Tell, it is even possible to run an OLTP workload and perform analytical queries on separate instances but accessing the same shared single-copy of the data. This mixed workload scenario enables scalable analytics on live production data and more generally enables use-cases not possible in partitioned databases. Although the focus of this dissertation is on OLTP workloads, we discuss the implications
and present initial ideas for performing mixed (OLTP and OLAP) workloads in shared-data architectures.

Sharing data, in particular with regard to OLTP workloads, raises several technical challenges [SC11]: First, shared data access requires synchronization. Records can be updated by multiple query processing nodes simultaneously, and therefore distributed concurrency control is necessary. Second, caching data in local buffers is only possible to a limited extent as updates made by one query processing node have to be visible on the other nodes instantly. As a result, most requests access the latest record version remotely from the shared store. To address these challenges, we introduce techniques for scalable transaction processing in shared-data environments. Specifically, we describe mechanisms for efficient data access, concurrency control, and data buffering. In combination with new hardware trends such as in-memory data storage and low-latency networking technology (i.e., Infini-Band [Inf14]), the techniques enable performance characteristics that top state-of-the-art partitioned databases.

1.3 Contributions

In this dissertation, we study the design space for distributed relational databases with regard to the requirements of cloud computing. After evaluating the state-of-the-art, we propose a shared-data architecture for elastic database services. We address the technical challenges of sharing data and highlight the architectural benefits. Our analysis is completed by an evaluation of distributed concurrency control techniques for shared-data systems. In summary, this dissertation makes the following contributions:

An Evaluation of Architectures for Transaction Processing in the Cloud. We compare alternative architectures for distributed transaction processing with regard to scalability, elasticity, and fault-tolerance. Furthermore, we report on the results of a comprehensive evaluation of existing commercial cloud services that have adopted these architectures. As part of the evaluation process, we argue that traditional benchmarks (like the TPC benchmarks) are not sufficient for cloud services and prevent comparable results. To that end, we present a new benchmarking methodology that considers the dynamic properties of cloud computing. We propose a benchmarking framework based on the TPC-W benchmark [Tra02] that allows users to compare cloud data services and
enables providers to gradually improve their solutions. Moreover, we assess the cloud storage offerings of Amazon, Google, and Microsoft. In particular, we evaluate how well the different services scale, how their cost/performance ratios compare and how predictable the cost is with regard to changes in the workload. Although the reported results are just a snapshot of the current state-of-the-art, they allow a look behind the scenes and enable to draw conclusions on the performance and cost characteristics of the different architectural variants.

Tell, a Distributed Shared-Data Database. We propose an architecture design for distributed relational databases that enables to combine the benefits of traditional RDBMS with the advantages of NoSQL storage systems. The architecture is based on two principles: We decouple query processing and transaction management from data storage, and we share data across query processing nodes. The combination of these design choices is the key to elasticity and agility while providing scalable ACID transactions. We implement the architecture in Tell and operate database instances on top of a shared in-memory record store. In order to minimize the overhead of synchronization and bad data locality, we present techniques for concurrency control, data access, and data buffering. We use multi-version concurrency control (MVCC) [BG81] and latch-free distributed B-tree indexes to ensure that transactions are never prevented from making progress. The synchronization overhead is kept minimal by relying on atomic read-modify-write (RMW) primitives [SVdWB08] as a key operation for conflict detection. Finally, we evaluate the effectiveness of our design and compare Tell to traditional partitioned databases.

Concurrency Control Techniques for Shared-Data Databases. As a critical scalability feature, the right concurrency control mechanism for transactional execution in shared-data architectures is of utmost relevance. Consequently, we evaluate alternative concurrency control approaches. We adapt established techniques such as snapshot isolation [BBG+95], timestamp ordering [BG81], and locking [BHG87] to operate in a distributed environment on top of a shared store. All techniques are implemented in Tell in order to provide a common ground for evaluation. As every concurrency control approach has specific characteristics and unique properties, an experimental study evaluates the performance and behavior of each approach using a variety of workloads. In different experiments, we adjust the read/write ratio and analyze the different approaches under
1.4 Structure of the Dissertation

The dissertation is structured as follows:

**Chapter 2** provides a review of the state-of-the-art in distributed databases and cloud data services. We present important technology trends and analyze several architecture variants for distributed transaction processing. We discuss how these architectures have been adopted in public data services by three of the big players in the cloud market, namely Amazon, Google, and Microsoft.

**Chapter 3** presents a new methodology for benchmarking cloud data services. We illustrate the limits of the TPC-W benchmark and propose modifications in the workload and the metrics in order to assess the specific characteristics of cloud services (i.e., scalability, pay-as-you-go, and fault-tolerance). In a second part, we conduct an evaluation of the performance, cost, and cost predictability of the cloud services presented in Chapter 2.
Chapter 1. Introduction

Chapter 4 describes the design principles and the implementation of Tell, a distributed database system based on the shared-data architecture. We address the challenges related to shared data. Specifically, we propose techniques for concurrency control, indexing, storage, and buffering. Finally, Tell is evaluated and compared to partitioned databases such as VoltDB [SW13] and MySQL Cluster [MyS14].

Chapter 5 explores alternative concurrency control mechanisms for shared-data architectures. We describe the implementation, illustrate the life-cycle of transactions, and explain the fail-over mechanism for several concurrency control techniques. The chapter is completed by a detailed evaluation of the different techniques on a variety of workloads.

Chapter 6 reviews the contributions and results obtained in this dissertation. The findings are summarized and conclusions are listed. Finally, we hint at open problems and present directions for future work.
This chapter presents the state-of-the-art in data management in the cloud. We analyze the advent of cloud computing and highlight alternative approaches to distributed transaction processing. The chapter is structured as follows:

First, we detail recent technology trends. We elaborate on the technical origins of cloud computing, discuss the implications, and illustrate the consequences for data processing systems. In particular, we highlight how cloud computing has enabled infrastructures at massive (warehouse) scale. Furthermore, current developments in database systems are outlined, and the trend to distribute and scale-out processing is emphasized.

In a second part, we revisit architectures for distributed databases. The classical three-tier architecture is compared to alternative variants based on partitioning as well as shared data. The architectures are examined with regard to the demands of cloud computing. That is, we study to what extent properties such as scalability, elasticity, and fault-tolerance can be provided. Furthermore, agility, an important property that facilitates operation in large deployments, is examined. Each architecture can be operated in combination with additional features, such as replication or caching, that affect system configuration and the amount of resources required.

Finally, we show how these architectures can be applied to cloud environments. Several alternative approaches to implement the architectures on existing cloud infrastructure services are listed. Additionally, commercial database-as-a-service (DBaaS) solutions that have adopted the architectures are described.
2.1 Technology Trends

This section reports on recent technology trends. The emergence of cloud computing has enabled access to vast amounts of resources and allowed for computing at warehouse-scale. Furthermore, several factors such as the increasing amounts of data, heterogeneous workloads, and modern hardware have introduced new challenges for data processing and consequently intensified the trend towards scale-out.

2.1.1 Warehouse-Scale Computing

With the advent of cloud computing \[AFG^{+10}\], the access to computation and storage resources has been considerably simplified. Cloud providers typically operate multiple data-centers distributed across the world and enable public access to their resources. Many influential companies, such as Dropbox, Netflix, or Airbnb, run their applications in cloud infrastructures.

The cornerstone of cloud environments is virtualization \[BDF^{+03}\]. Virtualization enables to share the resources of a physical server among several guest environments. A guest environment, also called virtual machine (VM), is an operating system running in isolation on a defined set of resources (CPU cores, main memory, and disk space). Resource isolation decouples operation from the hardware it runs on and thereby simplifies deployment. In particular, virtual machines can be started and stopped autonomously without interfering with other VMs. Virtualization is well-suited for environments in which utilization varies and where resources are continually re-assigned. It is therefore an elemental technique for cloud computing.

The performance overhead of virtualization has been subject to controversy. Although hardware support for virtualization has improved in recent years (i.e., Intel’s VT-x and AMD’s AMD-V), the authors of the Relational Cloud project \[CJP^{+11}\] argue that multiple databases sharing a server cause inefficiencies and require up to three times more machines. In contrast, some recent studies \[BMST09, Sal14b\] come to the conclusion that virtualized and native (“bare metal”) setups perform similarly for non I/O-intensive workloads. In practice, both types of operation are encountered. Some DBaaS systems operate on virtual machines (e.g., Amazon Relational Database Service), while others run on bare metal (e.g., Microsoft Azure SQL Database). Relational Database Service (RDS) and Azure SQL Database are described in more detail in Section 2.3.
Based on virtualized infrastructures, cloud providers offer basic computing services such as virtual machines, attached disk storage, and networking components. These services are typically referred to as *Infrastructure-as-a-Service* (IaaS). Users can acquire the resources to store data and operate applications in the data-centers of the cloud providers. For example, a major IaaS vendor is Amazon Web Services (AWS) [MVM10]. AWS operates a service called Elastic Compute Cloud (EC2) to provide compute capacity in the form of virtual machines. Currently, EC2 offers two dozen different types of VMs with varying amounts of virtual CPU cores, memory capacity, and storage technology. Amazon charges for the size of the VM and the duration of usage. Typically, users pay for every hour in which their VMs are active (pay-as-you-go).

IaaS enables to acquire (or release) resources on-demand and as a result to operate large infrastructures. Today, it is possible to acquire hundreds of virtual machines and scale out any cluster within minutes. However, large deployments raise challenges with regard to administration and maintenance. In large systems, particularly if built with commodity hardware, failures are common rather than exceptional. Seamless recovery from failures is of high relevance because service interruptions are no longer acceptable for most businesses. Consequently, fault-tolerance and high availability are important requirements in cloud-operated systems. In order to facilitate the management of large clusters, many cloud providers have introduced “helper” services such as load-balancing, service monitoring, data caching, and many more. These managed services support users in operating complex systems.

Cloud platforms, or *Platform-as-a-Service* (PaaS), are an alternative to infrastructure services. PaaS systems provide a fully managed solution stack as a service. The runtime environment, the storage system, and other components are operated and maintained by the cloud provider. Users no longer have to worry about configuring, updating, scaling, and backing up their systems. Scale-out of resources and handling of failures occurs automatically without user intervention. Examples of PaaS platforms are Google App Engine or AWS Beanstalk.

In summary, cloud technologies lay the foundation for scalable and elastic infrastructures. However, the interesting aspect is how data processing solutions running in the cloud can benefit from the possibilities. Specifically, modern data management systems must be designed with regard to scale-out.
2.1.2 Scale-Out Data Processing Systems

Data processing systems have undergone significant changes over the last decades. During the 1980s and 1990s, relational databases dominated the landscape for data processing. RDBMS were conceived as generic data management solutions with focus on efficient processing of OLTP workloads [GR92]. OLTP workloads typically consist of many short-running and update-intensive transactions. In the mid 1990s, the increasing amounts of data and new requirements with regard to business intelligence and decision support resulted in a new type of workload. Companies demanded to perform data analysis and run OLAP workloads on their data. OLAP workloads are typically characterized by complex ad-hoc queries resulting in long-running read-mostly transactions. The fundamental difference in workload patterns between OLTP and OLAP resulted in separate database systems optimized for processing either operational or analytical queries [CD97]. This separation is still visible in today’s enterprise architectures: Operational data is collected in traditional OLTP databases and is later transformed and transferred into data warehouses optimized for analytical processing.

The wide adoption of the internet and the information explosion has changed the requirements for data processing. New application areas such as graph processing, data streaming, or complex data analytics have surfaced and diversified the workloads storage systems have to process. In order to address these new demands, new data processing systems with custom techniques have been developed. This specialization has led to numerous alternative data processing approaches able to outperform more general RDBMS in specific areas. Stonebraker et al. argue on several occasions the end of the “one-size-fits-all” paradigm [SMA+07, Sto08]. The trend towards more diversified data processing solutions has been intensified by the hype around Big Data, a term used to indicate huge data sets that are complex to process with traditional processing systems. In the following, we describe two new types of systems that enable processing at large scale.

NoSQL systems are specifically designed with regard to cloud computing, and they are able to scale to a high number of nodes. They aim at providing scalability, elasticity and availability by giving up established features such as ACID transactions, rich schemas, or support for complex queries. For example, BigTable [CDG+08] is a storage system that allows atomic single-row operations on structured data. It is used at Google on several clusters exceeding 500 servers and storing hundreds of Terabytes each. Another example is Dynamo [DHJ+07], a key-value store that provides eventual consistent point operations
on key-value pairs. Dynamo uses consistent hashing [KLL+97] to distribute and replicate data across a high number of nodes. It has been developed to address the availability requirements of Amazon’s e-commerce platform. BigTable and Dynamo started the NoSQL movement and inspired the development of many storage systems [LM10, Cho13, Apa14b].

Recently, the Map-Reduce programming model [DG04] gained popularity. Map-Reduce is a simple model for parallelizing the processing of large data sets on commodity hardware. It consists of a map phase that partitions and distributes data for processing, followed by a reduce phase that aggregates intermediate results. Although initially developed for distributed batch-processing, Map-Reduce is more and more used for complex data analytics and Extract-Transform-Load (ETL) tasks [SAD+10]. Map-Reduce excels at these specific workloads and does not replace database systems. It is often used as a complement to classical data warehouses. Apache Hadoop [Apa14a] is a successful open-source implementation of the Map-Reduce model used in many large deployments. For instance, in 2011, Facebook reported to operate a data-center with a 30 Petabyte Hadoop cluster [Fac14]. Driven by the success of Hadoop, several extensions and alternatives such as Hive [TSJ+09], Pig [ORS+08], or Spark [ZCD+12] have been released.

Modern hardware has contributed to the specialization of data processing systems. In recent years, Dynamic Random Access Memory (DRAM) has become inexpensive, and the main memory capacity of servers has largely increased. As a result, storage systems [OAE+10] and databases [FCP+12, DFI+13, Vol14] that primarily store data in main memory are becoming common. DRAM enables low latency data access, and therefore new techniques and storage schemes [DKO+84, MKB09, PZ11] improve data processing efficiency. An interesting use-case for in-memory databases are mixed (OLTP and OLAP) workloads [KN11]. Another hardware trend that impacts data processing is the increase in the number of processing cores per machine. Traditional database designs have scalability issues with the new degree of parallelism [Sub12]. As a result, several approaches have been proposed to improve CPU utilization and scale with the number of cores [PJHA10, SSGA11, GAK12].

In summary, the requirements for data processing have evolved, and data management systems are adjusting to the new demands. Specifically, the increasing volume of data and Big Data challenges have pushed storage systems to cross the boundaries of a single machine.
Chapter 2. State-Of-The-Art

2.2 Distributed Database Architectures

This section revisits distributed database architectures as they are used in cloud computing today. As a starting point, the classic multi-tier database application architecture is described. Then, several variations of this architecture are presented. These variations are based on basic principles of distributed databases such as partitioning or replication. The interesting aspect is how these concepts have been packaged and adopted by commercial cloud services (Section 2.3). A paper that particularly inspired our work is the classic paper on client-server database architectures [DFMV90].

2.2.1 Classic

As a starting point, Figure 2.1 shows the classic architecture used for most database applications today (e.g., SAP R/3 [KKM98]). Requests from clients are dispatched by a load balancer (depicted as a carousel in Figure 2.1) to an available machine that runs an application (app) server. The app server handles incoming (HTTP) requests from clients and executes the application logic specified. Typically, applications are written in a high-

![Figure 2.1: Classic database architecture](image-url)
level programming language, such as Java or C#, and use standard APIs to access the
database (e.g., JDBC or ODBC). Database queries are shipped to the database server
that interprets the request, returns a result, and possibly updates the database. To ensure
persistence, the database server stores all data and logs on storage devices. The interface
between the database server and the storage system involves shipping physical blocks of
data (e.g., 64 KB blocks) using get and put requests. Traditional storage systems use disks
that are located in the same server or that are attached via the network (e.g., storage area
network). Figure 2.1 shows the variant in which storage is part of the database server
(depicted as a dark gray box). Instead of hard disks, storage systems can use solid-state
disks (SSD), main memory, or a combination of different storage media.

The classic architecture has a number of important advantages. First, it allows to use
“best-of-breed” components at all layers. As a result, a healthy market with a number
of competing products has emerged at each layer. Second, the classic architecture allows
scalability and elasticity at the storage and the application server layers. For instance,
if the throughput of the application needs to be adjusted due to an increased interest of
clients, then it is easy to add machines at the app server layer in order to handle the
additional requests. Likewise, machines at that layer can be turned off or used for a
different purpose if the workload drops. At the storage layer, machines (or disks) can be
added or removed in order to vary the bandwidth and the capacity of the storage system.

The potential bottleneck of the classic architecture is the database server. If the database
server is overloaded, the only way out is to scale up and use a bigger machine. The
machines used as database servers tend to be quite expensive because they must be provi-
sioned for peak workloads. Therefore, the classic architecture of Figure 2.1 has limitations
in scalability and cost, two important requirements in cloud computing. The remainder
of this section lists architectures that overcome the limitations at the database layer.

2.2.2 Partitioning

Figure 2.2 shows how the classic database architecture can be adapted in order to make
use of partitioning. The idea is simple: Rather than having one database server control
the whole database, the database is logically partitioned, and each partition is controlled
by a separate database instance (on a separate server). In the database literature, many
partitioning schemes have been studied: vertical partitioning vs. horizontal partitioning,
round-robin vs. hashing vs. range partitioning [CP84]. All these approaches are relevant and can be applied to database systems in the cloud.

A prominent partitioning technique is Sharding. The idea is to horizontally partition data according to a specific key and to assign related data (e.g., connected by a foreign key relationship) to the same partition. In Sharding, the partitioning key must be selected so that the number of transactions processed by a single partition is maximized. If all transactions can be processed by a single partition, the database instances do not need to coordinate (i.e., agree on the decision to commit or abort) and throughput increases with the number of machines. Sharding has been adopted in several large-scale database applications (e.g., at Facebook or Twitter [Col11]). Unfortunately, Sharding is not suited for every workload: Some business applications operate on complex schemas with inter-dependent data relations and cannot be partitioned effectively (e.g., SAP Sales and Distribution Benchmark [SAP14]).

In addition to the partitioning scheme, there are several variants of the architecture of Figure 2.2. First, partitioning can be transparent or visible to the application programmer (obviously, transparency is desirable). Second, the storage can be connected to the machines that run the database instances or located in a storage area network. Figure 2.2
2.2. Distributed Database Architectures

depicts the variant in which the access to the distributed database is not transparent and each application server maintains a connection to every database. A common solution to hide distribution is to add a federation layer that forwards queries to the right partition and coordinates distributed transactions. A federation layer is not necessarily a separate component, it can be integrated into the application or the database layer. In practice, other variants can also be found (Section 2.3).

Cloud platforms have adopted the architecture of Figure 2.2 to implement multi-tenant databases. That is, database systems that support multiple application databases (tenants). For instance, the architecture of Force.com [Sal14a], a service of the Salesforce platform to run custom-made applications, is based on partitioning. In Force.com, the partitioning key is the tenant. That is, data is distributed according to the application that generated and owns the data. All requests to the same tenant are handled by the same application and database server. As a result, Force.com is tuned to scale with the number of applications. However, the Force.com architecture does not support the scalability of a single application beyond a single database server.

Partitioning and the architecture of Figure 2.2 are a viable solution towards achieving the promises of cloud computing. In contrast to the classic architecture, the database instances can run on cheap commodity hardware. Furthermore, the database cluster can be composed of many instances that each operates on a fairly small data set in order to sustain high load. Given a limited number of cross-partition transactions, the synchronization overhead among the database instances is kept low and partitioning will scale. However, partitioning has limitations with regard to dealing with a fluctuating workload: Adding or removing machines in order to deal with a higher (or lower) query workload involves repartitioning the data and therefore moving data between machines. There are several approaches to perform online repartitioning without interrupting operation [SI09]. However, this process involves close coordination and usually negatively affects system throughput. A recent research avenue explores workload-aware data placement strategies that capture workload changes and adjust partitioning at runtime in order to minimize the number of distributed transactions [CJZM10, QKD13]. These strategies model the workload as a graph that is then partitioned. Although the approaches enable promising results, they incur overhead for many workloads.

In order to achieve fault-tolerance and improve availability, partitioning can be combined with replication (Section 2.2.4).
2.2.3 Shared-Data

Figure 2.3 shows a shared-data architecture that models the database as a distributed system. At first glance, this architecture looks similar to the partitioning architecture shown in Figure 2.2. The differences are subtle, but they have huge impact on the implementation, performance, and cost of a system. The shared-data architecture can also be characterized as a *shared-disk architecture* [Sto86] with a loose coupling between the components in order to achieve scalability and elasticity. It is the building block of a growing number of research as well as commercial systems [BRD11, BBC+11, DAEA13, SVS+13, GFJK+14].

In the shared-data architecture, the storage system is decoupled from the database instances (i.e., query processing and transaction management). The database instances concurrently and autonomously access the shared data from the storage system (e.g., a distributed record manager). In order to reduce overheads, the database tier can be merged with the application tier. That is, the database access is affected as a library as part of the application server rather than providing separate database processes. Compared to partitioned databases, a database instance no longer operates on a dedicated data partition but can access all the data and execute any query. Moreover, the communication pattern changes. While in a partitioned architecture app servers interact with
all database servers, in the shared-data architecture each app server communicates with a single database instance that in turn accesses data from all storage nodes.

The shared-data architecture has significant advantages with regard to cloud computing: It provides elasticity at all tiers. Each request can be routed to any (app/database) server so that elasticity can be achieved at that level. As database instances operate autonomously, they can be added or removed on-demand with minimal effort. Furthermore, the data can be replicated and partitioned in any way at the storage layer so that elasticity can be achieved at that level too. In contrast to partitioned databases, there is no mapping of processing resources to data. In the partitioned architecture, a data partition is assigned to a database server limiting the number and the type of queries that can be executed on that particular partition. In the shared-data architecture this mapping no longer exists, and therefore mixed workloads become possible. For instance, some database instances could run an OLTP workload, while others execute analytical queries on the same data.

The flexibility of the shared-data architecture, however, comes at a cost: Shared data access requires synchronization and involves network communication. In order to synchronize read and write access to the shared data, distributed protocols that guarantee different levels of consistency can be applied. Again, a large variety of different protocols are conceivable, and the classic textbook that gives an overview of such protocols and consistency levels is [TVS07]. Depending on the required consistency level, buffering data in the database instances is only possible to a limited extent. For instance, to provide ACID transactions, updates made by one database instance must be visible to the others instantly, and as a result the latest record version must be retrieved from the storage layer. This is a significant limitation because data access latencies can quickly become a dominant factor in query execution time.

Using existing cloud services, we study a variant of the shared-data architecture (Amazon S3, Section 2.3.1.5) that achieves eventual consistency. In accordance with the CAP theorem, we sacrifice consistency to achieve availability and partition tolerance. As a result, this variant provides durability and atomicity but does not comply with the isolation requirements of database transactions (i.e., serializability). In Chapter 4 we present a new database system that is based on the principles of the shared-data architecture but provides ACID-compliant transactions. Specifically, we propose techniques to overcome the mentioned limitations while keeping the advantages with regard to flexibility and elasticity.
2.2.4 Design Options

This section covers design options that are supplemental to each of the previously described database architectures. That is, they can be combined with the classic, partitioning, and shared-data architecture if needed. First, replication is described as an established approach to achieve fault-tolerance and improve scalability. Then, caching is presented as a technique to reduce the load on the database layer.

2.2.4.1 Replication

Figure 2.4 shows how replication can be used in a database architecture. Again, the idea is simple and has been studied extensively in the past. As with partitioning, there are several database servers. Each database server controls a copy of the whole database (or a partition of the database, if combined with partitioning). Furthermore, there are many variants conceivable. The most important design aspect of replication is the mechanism to keep the replicas consistent. A prominent protocol is ROWA (read-one-write-all) based on a master copy [PA04]. Figure 2.4 shows a variant in which the replication is transparent.
and the storage is associated with the database servers. If replication is transparent, then requests are routed automatically to the master or a replica. If replication is not transparent, applications direct all update requests to the database server that controls the master copy, and the master server propagates all updates to the replicas. Applications can issue requests of read-only transactions to any database server (master or replica).

Commodity hardware can be used in order to run the database servers. In particular, the replicas can run on cheap “off-the-shelf” machines as long as they can keep up with the update load from the master copy. Furthermore, the architecture of Figure 2.4 can nicely scale out and scale down if the workload is read-mostly. At any point in time, a replica server can be dropped in order to deal with a decreasing workload. Adding a replica server for an increasing query workload involves copying the database from the master (or a replica) to the new server. For update-intensive workloads, the master can become the bottleneck, as shown in Chapter 3.

Replication can be used in order to increase both the scalability and the reliability of a system. Distributed databases use replication as de facto standard to provide availability and prevent downtime in the presence of failures. Many protocols to ensure reliability based on replication have been presented, among others by Oracle RAC [Ora14], Azure Storage [CWO+11], and Spanner [CDE+12].

2.2.4.2 Caching

Figure 2.4 shows how caching can be integrated at the database layer. Caching can be combined with any other architecture (classic, partitioning, and shared-data). Again, the principle is simple: The results of database queries are stored by dedicated cache servers (for simplicity, Figure 2.4 only shows one cache server). Typically, these servers keep the query results in main memory in order to provide low access latencies. Memcached [Fit04] and Redis [Car13] are the most widely used open-source solutions to support such main memory caches.

There are several different schemes to keep the cache consistent with regard to updates to the database. Figure 2.4 depicts an approach in which the application controls cache consistency (look-aside cache). This approach has been adopted by Google App Engine that operates a farm of dedicated cache servers. Unfortunately, Google has not published any details on its implementation.
Caching can also help the cloud computing promises with regard to cost and scalability. Cheap memory-optimized machines can be used for efficient caching. Furthermore, adding cache servers is straightforward at any point. Nevertheless, cache deployments need to be managed with care. For instance, the removal or failure of a heavily accessed cache server will redirect the load to the database that in turn can become overloaded. Solutions for managing large cache clusters have been proposed by Facebook [NFG+13] and Twitter [Twi14].

2.3 Cloud Data Services

This section describes data services offered by three of the big cloud players, namely Amazon, Google, and Microsoft. The presented services were among the first cloud services to be publicly launched and have set the stage for many additional services that have appeared since. Therefore, the presented systems only cover a subset of the possible services or combinations of services. Nevertheless, the fundamental principles and architectures among cloud data services are similar, and the presented systems provide a representative overview of existing architectural variations.

To run applications in the cloud, users can choose between IaaS and PaaS offerings (or a combination of both). Infrastructure services (e.g., running a database on a VM) are most flexible because they provide fine-grained control and free choice of components. Any available software system can be installed, configured, and operated on VMs. Furthermore, users can decide to either scale up (i.e., switch to a bigger VM) or to scale out (i.e., add a new VM) whenever required. As a drawback, IaaS requires time and effort to set up and maintain the infrastructure. PaaS systems address these issues: Platform services are managed solutions that are operated by the cloud vendor. They are based on the same common architectures as IaaS solutions but are optimized to serve many tenants. Platform services are typically accessed via proprietary APIs and provide limited means of customization. In the absence of established standards, platform services differ in many aspects: programming model, software components, and properties of the database layer.

The services studied in this dissertation include IaaS as well as PaaS systems. An overview, including differences and key characteristics, is given in Table 2.1. The architecture is a fundamental property. Therefore, it is highlighted in Table 2.1. Another relevant category is the hardware configuration (HW Configuration) that specifies if resources are managed
2.3. Cloud Data Services

<table>
<thead>
<tr>
<th>Business Model</th>
<th>AWS MySQL</th>
<th>AWS MySQL/R</th>
<th>AWS RDS</th>
<th>AWS SimpleDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cloud Provider</td>
<td>Flexible</td>
<td>Flexible</td>
<td>Amazon</td>
<td>Amazon</td>
</tr>
<tr>
<td>Application server</td>
<td>Apache Tomcat</td>
<td>Apache Tomcat</td>
<td>Apache Tomcat</td>
<td>Apache Tomcat</td>
</tr>
<tr>
<td>Database</td>
<td>MySQL</td>
<td>MySQL Replication</td>
<td>MySQL</td>
<td>SimpleDB</td>
</tr>
<tr>
<td>Storage / File Sys.</td>
<td>EBS</td>
<td>EC2 &amp; EBS</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Consistency</td>
<td>Repeatable Read</td>
<td>Repeatable Read</td>
<td>Repeatable Read</td>
<td>Eventual Cons.</td>
</tr>
<tr>
<td>App-Language</td>
<td>Java</td>
<td>Java</td>
<td>Java</td>
<td>Java</td>
</tr>
<tr>
<td>DB-Language</td>
<td>SQL</td>
<td>SQL</td>
<td>SQL</td>
<td>SimpleDB Queries</td>
</tr>
<tr>
<td>HW Configuration</td>
<td>manual</td>
<td>manual</td>
<td>manual</td>
<td>manual/automatic</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Business Model</th>
<th>AWS S3</th>
<th>Google App Engine</th>
<th>Microsoft Azure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cloud Provider</td>
<td>Flexible</td>
<td>Google</td>
<td>Microsoft.Net</td>
</tr>
<tr>
<td>Application server</td>
<td>Apache Tomcat</td>
<td>App Engine</td>
<td>Azure.Net</td>
</tr>
<tr>
<td>Database</td>
<td>none</td>
<td>DataStore</td>
<td>Azure SQL Database</td>
</tr>
<tr>
<td>Storage / File Sys.</td>
<td>S3</td>
<td>GFS</td>
<td>-</td>
</tr>
<tr>
<td>Consistency</td>
<td>Eventual Cons.</td>
<td>Snapshot Isolation</td>
<td>Snapshot Isolation</td>
</tr>
<tr>
<td>App-Language</td>
<td>Java</td>
<td>Java/App Engine</td>
<td>C#</td>
</tr>
<tr>
<td>DB-Language</td>
<td>low-level API</td>
<td>GQL</td>
<td>SQL</td>
</tr>
<tr>
<td>Architecture</td>
<td>Shared-data(+C)</td>
<td>Part.+Repl.(+C)</td>
<td>Classic+Repl.</td>
</tr>
<tr>
<td>HW Configuration</td>
<td>manual</td>
<td>automatic</td>
<td>manual/automatic</td>
</tr>
</tbody>
</table>

Table 2.1: Overview of studied cloud data services

manually or automatically. For most studied services, the user must manually configure how many virtual machines are used and on which kind of servers these virtual machines are deployed. Google App Engine is a PaaS service and thus allocates hardware resources fully automatically for all tiers depending on the workload. SimpleDB and Azure SQL Database are DBaaS systems. That is, the database and storage layers are automatically provisioned, while the application layer requires manual configuration of the hardware resources. Amazon provides a service called Auto Scaling that can be used to automatically scale out and scale down EC2 machines for the application tier.

The services of Table 2.1 are evaluated in Chapter 3. Therefore, we mention details relevant to the experimental evaluation throughout this Section. For instance, in order to better control the experiments and focus on the scalability at the database tier, we did not make use of the Amazon Auto Scaling service.
Chapter 2. State-Of-The-Art

2.3.1 Amazon (AWS)

Amazon is the world’s largest IaaS vendor. Its extensive infrastructure is a good basis to implement different architectural variants. The Amazon services we use are EC2 (for virtual machines), EBS (for a storage service that can be mounted like a disk), and S3 (for storage that can be used as a key-value store). Since these services could easily be provided by other cloud providers, the architectural variants implemented on top of Amazon’s infrastructure are declared as “flexible” in Table 2.1. Amazon has also released platform services such as RDS and SimpleDB. These services are only available on the Amazon cloud. All the Amazon services are described in full detail in [Ama14]. [Ama14] also lists the prices for using each service. The remainder of this section describes how we implemented five different architectural variants using AWS.

2.3.1.1 AWS MySQL

The first variant studied follows the classic architecture of Figure 2.1. This variant can be seen as a baseline because it follows a traditional (non cloud-enabled) model to deploy an enterprise application. In our implementation, we use a varying number of EC2 virtual machines in order to run the application servers and execute the application logic. More specifically, we use Tomcat as a combined web and app server. The number of EC2 machines can be varied depending on the workload. For the database server tier, we use the MySQL database with the InnoDB storage engine. The MySQL server is run on a separate EC2 instance. As a storage system, we use EBS for both the database and the logs. EBS guarantees persistence (the EBS data is replicated). In theory, the database could also be stored on the local disk of the EC2 machines that runs the MySQL server. However, in that approach, all data is lost if the virtual machine fails. That is why this option was not considered in the experiments in Chapter 3.

2.3.1.2 AWS MySQL/R

In order to study the classic architecture in combination with replication, we use MySQL and the built-in replication feature on a set of EC2 machines. This variant enables to use the (cheaper) local disks of the EC2 instances for storing the database because the durability of the data is guaranteed by the presence of several replicas. EBS is only used for the logs of the master copy. These logs are needed to recover when the master fails.
MySQL Replication uses the ROWA/master copy protocol described in Section 2.2.4 in order to synchronize all update requests. Read requests from the application can be processed by any replica database. The replication is not transparent. Consequently, each application server maintains a connection to the master copy and a connection to one replica. Requests of updating transactions are handled by the master, whereas requests of read-only transactions are issued to the replica associated to the application server. As for the AWS MySQL variant, Tomcat is used as an integrated web and app server, and the number of EC2 instances for that tier is varied depending on the workload.

2.3.1.3 AWS RDS

The Amazon Relational Database Service is a DBaaS that provides relational database instances as a service. In essence, RDS implements the same platform as provided by the AWS MySQL approach described above. Therefore, we expect both approaches to perform and cost similarly. The difference is that RDS is pre-packaged so that users do not need to worry about managing the deployment, software upgrades, and backups.

The RDS service was released in late 2009 and has evolved in the last years. MySQL was the first database product supported. Later, Amazon added support for SQL Server, Oracle Database, and PostgreSQL. More recently, RDS has been extended with optional replication. RDS replication enables to run one or more read replicas in the same data-center as the master copy and a fail-over replica in a remote data-center. The latter guarantees fault-tolerance if the primary data-center goes down. Consequently, RDS is based on the classic architecture and can be combined with replication.

RDS comes in twelve “sizes” (VM instance types) ranging from micro to octuple extra large database servers. Obviously, a micro server is sufficient for light workloads, whereas large servers are needed for heavy workloads with high update throughputs or complex queries. The instance types vary in performance with regard to CPU, memory, I/O, and network. The prices for RDS range from USD 0.017 per hour (t2.micro) to USD 3.78 per hour (r3.8xlarge). This way, RDS enables scale-up on the database tier.

We use RDS with MySQL in order to better compare with the other variants. As replication was not available at the time we conducted our experiments, we use RDS without this feature.
2.3.1.4 AWS SimpleDB

SimpleDB is a proprietary NoSQL service developed and operated by Amazon. It provides a simple interface that allows to insert, update, and delete records. Furthermore, it allows to retrieve records based on their key values or based on ranges of primary and secondary keys. Details of the implementation of SimpleDB have not been published. From personal communication with Amazon engineers, we have learned that the SimpleDB architecture can be best characterized as a combination of partitioning (Figure 2.2) and replication. Unlike MySQL/R, SimpleDB does not synchronize concurrent access to different copies of the data. As a result, only a low level of consistency (i.e., eventual consistency) is provided. SimpleDB introduced support for consistent read as a higher level of consistency. Unfortunately, this release came too late for consideration in our experiments.

SimpleDB measures the machine utilization of each request and charges based on the amount of machine capacity used. Therefore, the more requests are performed, the higher the cost. Unfortunately, this cost model is not transparent as users do not have the possibility to estimate the cost of a request in advance.

At the application layer, the AWS SimpleDB variant is implemented using the same configuration as the other AWS variants (i.e., Tomcat with a varying number of EC2 machines). Since SimpleDB does not support SQL, SQL operators such as join and aggregation have to be implemented at the application level. Obviously, this approach results in shipping all the relevant base data from SimpleDB to the application servers. Consequently, the performance is expected to be lower compared to full-fledged SQL database systems.

In 2012, Amazon announced a new NoSQL solution called DynamoDB. Although DynamoDB was supposed to replace SimpleDB, SimpleDB remains in operation until today. DynamoDB has taken over many features from SimpleDB and uses the same architecture, but it is based on a different cost model. DynamoDB requires to provision the expected load (i.e., operations/second) beforehand on each existing domain. This mechanism enables Amazon to reserve resources in order to guarantee that the load can be handled. Although the cost model of DynamoDB is more transparent compared to SimpleDB, the cost for the provisioned resources are charged even if no load is generated. Chapter 3 presents the results obtained with SimpleDB in detail and refers to DynamoDB whenever we observed a divergence in the results.
2.3. Cloud Data Services

2.3.1.5 AWS S3

As a fifth architectural variant, we implement applications directly on top of Amazon S3. This variant corresponds to the shared-data architecture depicted in Figure 2.3. S3 only provides a low-level put/get interface so that any higher-level services such as SQL query processing and B-tree indexing have to be implemented as part of the application. We did that as part of a library that provides basic database features. In order to improve performance, the library stores several tuples in a single S3 object. Furthermore, the library implements a protocol to synchronize concurrent access from multiple application servers to the same S3 objects. For this purpose, we use the basic protocol proposed in [BFG+08]. This protocol can be implemented in a straightforward way on top of S3. Like the AWS SimpleDB variant, this protocol supports eventual consistency. Higher levels of consistency can be implemented using the other protocols of [BFG+08] but were not considered as part of our study in order not to lose focus. To improve performance, S3 objects are cached in the application servers. Specifically, S3 objects that represent B-tree index pages were cached across transactions. The basic protocol described in [BFG+08] ensures correctness when data and index pages are cached.

[BFG+08] uses SQS, a persistent queuing service provided by Amazon, in order to implement durability and eventual consistency. We have not followed that recommendation and implemented the basic protocol on our own implementation of queues. With regard to the experiments, the queues are deployed on a varying number of EC2 machines, and EBS is used in order to store the logs of the queues persistently. Again, we vary the number of EC2 machines depending on the workload. The TTL period for caching is set to 120 seconds and the checkpoint interval (defined in [BFG+08]) is 10 seconds. Checkpoints are effected by a watchdog process [BFG+08].

The integrated web/app/database server is based on Tomcat and the library to implement basic SQL constructs and consistency. We vary the number of EC2 machines, again depending on the workload.

2.3.2 Google

Google started as a PaaS provider in 2008 and has since expanded into a “full-blown” cloud vendor. In addition to platform services, Google now provides similar IaaS services as Amazon. For instance, Google Compute Cloud, Google Cloud Storage, and Google Cloud
Chapter 2. State-Of-The-Art

SQL are competing with Amazon EC2, S3, and RDS respectively. We study Google App Engine (AE), a PaaS service that enables to deploy whole applications without providing control over the computing resources. Google AE automatically scales out the resources consumed by an application if required by the workload. Google AE supports Python, Java, PHP, and Go as programming languages, all with embedded SQL for accessing the database. We use the Java version of Google AE with the Google SDK and data mappings based on JPA.

The database layer of Google AE is called Datastore and is based on [BBC+11]. Google AE Datastore has adopted a combined partitioning and replication architecture. To access data, Google provides a simplified SQL dialect, referred to as GQL. Whenever GQL is not sufficient, we implemented the missing functionality as part of a library in the same way as for the AWS S3 and AWS SimpleDB variants. For instance, GQL does not support group by, aggregate functions, joins, or LIKE predicates.

A relevant feature in the Google AE software stack is a main memory cache. Programmers can use a simple interface to put and retrieve objects from the cache. If the cache is used, Google AE closely follows the partitioning architecture (Figure 2.2) in combination with replication and caching (Figure 2.4). For the experimental study in Chapter 3, we studied both variants (caching and non-caching).

2.3.3 Microsoft

Microsoft has joined the cloud business in 2010 with the Azure platform. Like Amazon and Google, Azure provides a large set of cloud services that cover many IaaS and PaaS use-cases. To study the Azure platform, we use Azure Virtual Machines, an IaaS service identical to Amazon EC2. For data storage, we use the managed service Azure SQL Database (previously named SQL Azure). Azure SQL Database was the first storage service offered by Microsoft. At the time we conducted our study, SQL Database was the only mature data management service available in the Azure cloud. More recently, Microsoft has released new services that complement SQL Database such as DocumentDB, Storage, and HDInsight.

SQL Database is based on an adapted variant of Microsoft’s relational database SQL Server [CKE10, BCD+11] and has adopted the classic architecture in combination with replication (master-slave replication). Therefore, SQL Database should be directly comparable to the AWS MySQL/R variant. SQL Database is available in different sizes. The
smallest and weakest variant limits the size of the database to 2 GBs. The largest and most powerful database stores up to 500 GBs. In order to support larger amounts of data, Microsoft has introduced a federation feature to distribute data across several SQL Database instances. The federation feature forwards queries to the right database instance and enables SQL Database to operate in a partitioning architecture (Figure 2.2) that is transparent to the application. Unfortunately, queries that span multiple instances are not supported. For our experiments, a single database instance was sufficient and we did not use the federation feature.

2.4 Concluding Remarks

Cloud computing and the on-demand access to vast amounts of computing resources has changed the requirements towards data management. New data processing systems must scale to a large number of nodes and have to dynamically adjust resource usage. In a rush to obtain market shares, cloud vendors have developed and released many alternative services for data processing. As a result, users are confronted with a diversity of options. The absence of established standards resulted in a complex market, and the available services vary considerably in properties and provided guarantees. Analyzing the architectures is a first step in categorizing cloud data services.

In the next chapter, we present a new benchmark methodology for cloud data services and conduct an evaluation of the systems and services described in this Section. We show how the alternative solutions compare in terms of performance and cost. More importantly, we verify to what extent the type of architecture allows to draw conclusions on the results obtained.
This chapter evaluates the services presented in Section 2.3 and analyzes the performance and cost for transactional OLTP workloads. Before presenting results, we initially discuss why existing database benchmarks are not suited for cloud services and present ideas for a new benchmark.

Traditional database benchmarks are designed to evaluate static, non-changing systems with well defined properties. Such an approach is not applicable to dynamic cloud services, and therefore in the first part of this chapter, we present a benchmark methodology that takes into account the specific properties of cloud data services.

The second part of this chapter presents the results of the experiments. Our benchmark is implemented on all the services of Section 2.3. The evaluation measures throughput, cost, and cost predictability and enables to draw conclusions with regard to the architecture of the different systems. Obviously, the results reported are just a snapshot of the state-of-the-art. The key contribution is to establish a framework that allows users to compare products and enables cloud vendors to gradually improve their services.

### 3.1 Methodology for a Cloud Benchmark

Database benchmarks assess the performance of transactional data management systems for particular workloads. The most prominent examples are the various standardized TPC-
benchmarks that specify workloads derived from real-world applications (e.g., TPC-H for OLAP [Tra13], TPC-C for OLTP [Tra10], or TPC-W [Tra02] for the whole application stack). The TPC-benchmarks are designed for RDBMS and have strict requirements on how to execute the workload and report the results. For instance, they enforce ACID transactions and different levels of isolation as defined by the SQL standard.

With the advent of cloud computing, providers have released data services with different capabilities and guarantees (Chapter 2). In particular, many services (e.g., NoSQL systems) relax consistency guarantees and simplify the query language to improve availability and elasticity. These services do not fulfill the requirements defined in the official TPC-benchmark specifications.

In the following, we propose a benchmark methodology that allows for comparing alternative cloud services. To start with, we outline the requirements for a cloud benchmark. Then, we highlight the problems of existing database benchmarks taking the example of the prominent TPC-W benchmark. Finally, we propose modifications and introduce new experiments to measure scalability, cost, and fault-tolerance.

### 3.1.1 Requirements

Today’s cloud services differ with regard to cost, performance, fault-tolerance, consistency guarantees, service level agreements (SLA), and programming languages. As developers are confronted with a variety of options and trade-offs, the primary goal of a cloud benchmark should be to help developers choose the right services for their applications.

**Features and Metrics.** Traditional benchmarks measure performance and cost of static systems. The used metrics are still relevant for cloud applications but need to be adjusted in order to consider dynamic systems in which resources come and go.

In addition, cloud providers promise scalability, fault-tolerance, and a pay-as-you-go cost model [Hay08]. Unfortunately, these features are often differently fulfilled. For example, most cloud providers claim to provide nearly infinite scalability for their services, but an unanticipated combination of services (or the limitations of one service) can incur scalability issues (e.g., limitations on persistent storage). Furthermore, different price plans lead to varying overall cost. For instance, PaaS cost models are typically based on CPU utilization. A web application that supports a single transaction per hour is priced
for this single transaction. In contrast, IaaS virtual machines are typically charged per hour. In this model, the costs for a VM instance are the same regardless if one or a thousand transactions are performed. Finally, fault-tolerance differs significantly between providers. The number of node failures a system can sustain without user notice, or single vs. multi data-center replication are just some examples. The previous illustrations highlight the need for new cloud-specific metrics. These metrics should enable qualified assessments with regard to scalability, fault-tolerance and cost.

Architectures. In the absence of standards, cloud services implement a variety of architectures. Section 2.3 demonstrated how alternative services can be used in different ways to achieve the same goal. It is not clear which of the architectural variants is best suited for a particular use-case. Furthermore, services from different cloud providers cannot always be freely combined (e.g., Azure SQL Database with Google App Engine). Thus, it is more important to know how different services play together rather than to know which provider is particularly good in a specific product (e.g., in a key-value store). A cloud benchmark should not impose an architecture but be general enough to cover multiple architectural variants. In addition, the complete application stack should be measured instead of micro-benchmarking single services.

Different data services are not always directly comparable even if based on the same architecture. For instance, achieving strong consistency in a highly distributed system is more expensive (in terms of latency and throughput) than weak consistency and hence affects scalability. Cloud solutions position themselves somewhere in the design space of relaxed consistency, high scalability, and high availability. A benchmark should on the one hand consider these different design options to avoid comparing apples and oranges, while on the other hand it should not force all solutions into a single setup.

3.1.2 Limitations of the TPC-W

The TPC-W benchmark specifies an online bookstore that consists of 14 web interactions (WIs) enabling to browse, search, display, update, and order products. The system under test (SUT) consists of one or several application servers implementing the business logic, and a data processing system to manage and store the data (i.e., a RDBMS).

The TPC-W workload specifies a remote browser emulation (RBE) system that simulates an defined number of users sending (HTTP) requests to the SUT. Each request executes
Chapter 3. Evaluation of Cloud Data Services

one web interaction. The goal of the RBE is to realistically simulate the simultaneous browsing behavior of many users. The TPC-W benchmark defines three different mixes: Browsing mix, Shopping mix, and Ordering mix. A mix specifies for every emulated user an alternative sequence of web interactions with a varying ratio of browse and order operations. Browse operations only read data, whereas order operations perform data updates. Benchmarking with the different mixes indicates the performance impact of a varying ratio of update operations.

The TPC-W benchmark evaluates the entire application stack and does not make any assumptions on the architecture and technologies used. Thus, two of the previously defined requirements are already fulfilled. Nevertheless, the TPC-W has several limitations:

First, the TPC-W enforces data operations to be performed as part of ACID transactions. Obviously, the TPC-W has been designed to evaluate transactional database systems. However, cloud data services do not always provide such strong consistency guarantees. In addition, existing TPC-W implementations for the cloud (e.g., [BFG+08]) violate the official specification. With divergent benchmark implementations, comparing results becomes pointless.

Second, although the TPC-W is an OLTP benchmark, several SQL queries are complex and challenging for cloud data services that do not have a rich query interface (e.g., S3 or SimpleDB). Particularly problematic is the Bestseller web interaction that includes a very expensive query. Implementing this query at the application level would be prohibitive because almost the whole database needs to be scanned.

Third, the primary throughput metric of the TPC-W is the maximum number of web interactions per second (WIPS) the SUT can handle. The maximum number of WIPS is determined by increasing the number of emulated browsers (EBs), for several consecutive benchmark runs, until more than 10% of the response times exceed a specified threshold. Thus, to determine the peak number of WIPS, a system configuration needs to be benchmarked over and over again with different load. This procedure is unsuited for cloud environments as resources are typically added the moment load increases. As a result, the SUT adjusts automatically to the fixed load during the execution of a benchmark run. If resources are added every time the number of EBs are increased, the amount of WIPS would ideally never reach a maximum.

Finally, another metric of the TPC-W is the ratio of cost to performance (i.e., $/WIPS). The metric considers the total cost of ownership of the SUT including software, hardware,
maintenance, and administration expenses (for three years). The total cost is divided by
the maximum number of WIPS to determine the $/WIPS ratio. Again, the approach to
calculate this metric is not applicable to the cloud. For a cloud benchmark two problems
arise: First, as discussed earlier, it is unclear if a maximum number of WIPS exists. Thus,
there is no fixed load for which the cost can be calculated. Second, different price plans
prevent to calculate a single $/WIPS number. The cost vary depending on the load, and
consequently it is difficult to estimate the total cost for three years in advance.

Finally, the TPC-W benchmark lacks adequate metrics and experiments for measuring
specific cloud features such as the ability to adapt to load changes or fault-tolerance.

3.1.3 A Benchmark for the Cloud

Despite its limitations, the web-based e-commerce scenario makes the TPC-W benchmark
a potential candidate for benchmarking cloud services. Taking into consideration the pop-
ularity of the benchmark both in industry and academia, we propose to adjust the TPC-W
in order to fulfill the cloud requirements. Accordingly, this section studies necessary mod-
ifications to transform the TPC-W into a cloud benchmark. In the following, we address
the limitations of the TPC-W and propose ideas for new experiments.

3.1.3.1 Configurations

Individual runs of the TPC-W benchmark can use different settings for the scale of the
database as well as choose between different web interaction mixes. In addition to these
configurations, we believe that the consistency level is an additional parameter that can
be varied for the benchmark execution.

The consistency configuration parameter addresses the problem of weaker consistency
guarantees. Cloud data services offer a level of consistency that ranges from the weak
BASE guarantees (Basically Available, Soft-State, Eventually Consistent) to the strong
transactional ACID. In order to analyze the spectrum of different consistency guarantees
using a cloud benchmark on the one hand, and the need to produce comparable bench-
marking results on the other hand, we propose that a benchmark can choose between three
levels of consistency:
Chapter 3. Evaluation of Cloud Data Services

- **Low**: All web interactions use only the BASE guarantees.

- **Medium**: The web interactions use a mix of consistency guarantees ranging from BASE to ACID. For instance, user reviews can use BASE guarantees, while product orders need transactional ACID guarantees.

- **High**: All web interactions use only the ACID guarantees.

If a data service does not provide the desired consistency level, either the benchmark can not be executed using this consistency level, or the benchmark implementation must add additional functionality to the application to provide the necessary consistency guarantees.

The metrics reported by the cloud benchmark always refer to a particular configuration including database scale, web interaction mix, and consistency level.

3.1.3.2 Metrics and Experiments

The standard approach to determine WIPS and $/WIPS is not suited for dynamic cloud systems (Section 3.1.2). Thus, we propose to adjust the procedures to determine WIPS and $/WIPS. Specifically, we suggest to vary the number of EBs during a benchmark run in order to observe how a system adapts to changing load. Furthermore, we outline additional experiments to measure adaptability (i.e., workload peaks) and fault-tolerance.

**Scalability.** Ideally, cloud services scale linearly and infinitely with a constant cost per web interaction. However, service restrictions, consistency requirements, price plans, and physical limitations can prevent perfect scaling.

In order to measure scalability and the ability of a dynamic system to adapt to changing load, we suggest to increase the number of issued WIPS over time and continuously count the number of web interactions within the response time threshold. The number of EBs and as a result the number of web interactions issued to the SUT is increased at a pre-defined rate. Ideally, a system scales linearly and answers all issued WIPS in time. In Figure 3.1, the dashed blue line shows the issued WIPS. This line also indicates ideal scaling behavior. If a system does not scale, more and more WIs will violate the response time (RT) constraint, and as a result less WIPS in RT are processed (indicated by the solid red line in Figure 3.1).
### 3.1. Methodology for a Cloud Benchmark

The TPC-W benchmark specifies that the size of the database grows with the number of EBs. As we increase the number of EBs during the benchmark run, we propose to define a fixed database size independent of the amount of load generated. In the experiment of Section 3.2, we carried out all experiments with a data set that complies to a standard TPC-W database for 100 EBs.

Another adjustment to the TPC-W specification is the duration of the benchmark. Assuming that perfect scaling does not exist, we suggest to define the end of the benchmark as the time when the difference between the issued WIPS and the WIPS in RT exceeds a pre-defined limit. Still, reaching this value might take arbitrarily long. Therefore, the benchmark should additionally define a maximum execution time for which the results are assumed to be sufficiently significant. For instance, in the experiments of Section 3.2, we varied the number of EBs from 1 (≈ 500 requests per hour) to 9000 (≈ 1250 requests per second).

Furthermore, to avoid that the Bestseller web interaction biases the results, we replace the query that computes the bestsellers with one that randomly returns a set of products. This new query can be efficiently executed on common cloud data services.

**Cost.** In accordance with the TPC-W benchmark, we measure cost in USD per WIPS ($/WIPS). Thus, the cost of running the system (including all self-owned infrastructure and administration costs) is divided by the current WIPS rate. In contrast to the TPC-W
specification, we do not compute a single $/WIPS value for the peak throughput but calculate the cost depending on the load (i.e., number of EBs) as shown in Figure 3.2. Ideally, the cost is constant independently of the scaling factor of the system (perfect pay-as-you-go). However, price plans might cause variations of the $/WIPS with respect to the current scaling factor. Figure 3.2 demonstrates the effect of lot-sizes on the cost per WIPS. For example, the jump in the cost could be caused by additional EC2 instances. A EC2 instance comes at a fixed cost and can cause a substantial increase in the proportional cost per WIPS.

To measure the cost variation, we suggest to not only measure the average cost per WIPS but also the standard deviation of the cost during scaling (i.e., $\sigma(\$/WIPS))$. The standard deviation is an important indicator on how the cost might vary and can help to better plan a system. A low value indicates perfect pay-as-you-go pricing, whereas a large value corresponds to a traditional non-cloud scenario.

While the $/WIPS metric is a good indicator for the cost given a defined throughput, another interesting metric is the (projected) total cost per day ($/day). Again, the value is dependent on the load. Obviously, the lower the cost, the better.

Peaks. In addition to scalability, another important property of cloud services is the ability to adapt to peak loads. In this experiment, we not only focus on scale-up but also on scale-down after a peak. To measure the behavior, we suggest again to vary the issued
WIPS over the time as shown in Figure 3.3. In contrast to the scalability experiment (Figure 3.1), the benchmark executes a base load that is increased for short periods of time.

In this experiment, we define the ratio between WIPS in RT and issued WIPS as an indicator that reflects the adaptability to peak loads. A value of 1 indicates perfect absorbing of peaks, whereas a smaller value indicates higher adaptation times. Additionally, the cost and cost standard deviation have to be measured. The lower and the more stable the cost, the better a system is suited for adapting to peak-loads.

The base load, the peak load, and the speed at which the load is increased are three parameters in this experiment. A too high base load might already push the system to the scalability limit. Furthermore, a slow increase in the load is in general better absorbable than a fast and sudden increase. Thus, we suggest to use several pre-defined variations. The idea is to start with a modest base load and vary the intensity of the load increase until the ratio, as defined before, becomes smaller than 1. As a result, we can determine the *elevation* factor, the highest load increase that can be absorbed without user notice.

**Fault-Tolerance.** Most cloud services operate on commodity hardware. In such deployments, hardware failures are common and not the exception. Thus, a cloud benchmark must evaluate system behavior in the presence of failures.

To measure fault-tolerance, we suggest that a specified amount (in percent) of the total resources used by the benchmark application is shut down, regardless if it is a storage service or an EC2 instance. Figure 3.4 indicates the manually induced failures with stars. As platform services are typically self-healing, the resources are automatically replaced. Hence, the shutdown of resources can be repeated several times.

As for the peak workload experiment, the ratio between WIPS in RT and issued WIPS is calculated. A ratio of 1 indicates that the system is reliable and can perfectly handle the induced failures. The failure metric depends on the amount of resources shut down. Defining a representative percentage or varying the scenario are both viable solutions and open for discussion. Again, in addition to the ratio, it is possible to report the maximum percentage of failures supported until the ratio falls below 1. In contrast to the previous experiments, the ability to handle failures is most likely only reportable by cloud providers and not by cloud users.
Chapter 3. Evaluation of Cloud Data Services

3.1.4 Related Work

Benchmarking and performance analysis of data management systems received significant attention in the 1980s and early 1990s [BDT83, Gra92]. The research efforts eventually resulted in the series of standardized TPC-benchmarks for assessing RDBMS performance and cost for different workloads (e.g., TPC-H for OLAP [Tra13], TPC-C for OLTP [Tra10], or TPC-W [Tra02] for the whole application stack).

More recently, the emergence of new application domains has motivated the development of new benchmarks that target specific fields such as XML data stores [SWK+02], data streaming [ACG+04], key-value stores [CST+10], table stores [PPR+11], hybrid OLAP/OLTP databases [FKN11], and temporal databases [KFMK14].

With the emergence of cloud computing, several studies have assessed the performance and scalability of cloud computing infrastructures. In the database community, recent work compared the performance of Apache Hadoop with more traditional (SQL-based) database systems [PPR+09]. This work focuses on read-only, large-scale OLAP workloads, whereas our work focuses on OLTP workloads. The results of a related study on cost-consistency trade-offs for OLTP workloads in the cloud have been reported in [KHAK09].

Berkeley’s Cloudstone [SSS+08] project is similar to our benchmark approach. Cloudstone specifies a web 2.0 application and a workload for an end-to-end study of cloud infrastructures. However, Cloudstone’s primary goal is to study cost, while our approach enables to evaluate more cloud service properties.

The Yahoo Cloud Serving Benchmark (YCSB) [CST+10] is a benchmark alternative for cloud data services. The YCSB models a simple workload. That is, a variation of get, put, and range operations on key-value pairs. Because of it’s simplicity, the YCSB can be executed on most data processing systems and has become popular in the NoSQL community. In contrast to our approach, it neither models an OLTP workload nor covers the entire application stack.

In order to facilitate the evaluation of cloud services, several projects [RA12, DPCCM14] propose frameworks to simplify setup and execution of benchmark workloads. The frameworks act as a testbed and provide the infrastructure for benchmarking a variety of systems. Our benchmark could be integrated in one of these frameworks to automate the assessment of the services discussed in Section 2.3
3.2 Experiments and Results

The goal of this performance evaluation is to study the scalability (with regard to throughput) and cost of alternative cloud services under different workloads. This section presents the results of running the TPC-W benchmark with the modifications proposed in Section 3.1 on the cloud service variants described in Section 2.3. The focus of the experiments is on scalability and cost. Evaluating the other promises of cloud computing such as fault-tolerance and peak performance is left for future work.

The experiments with AWS RDS and Microsoft Azure were carried out in February 2010. The experiments with all other variants were carried out in October 2009. In October 2014, we re-estimated the cost of all experiments based on the current pricing models of the providers (but we did not re-run the experiments). As a result, we were able to analyze how the cost of cloud services changed over the last four years (Section 3.2.4). If not mentioned otherwise, the cost results are based on the data from February 2010.

Obviously, all providers continuously make changes that affect the results, just as hardware and technology trends affect the results of any other performance study. Nevertheless, we believe that the results of this study reveal important insights with regard to the architectures for cloud computing platforms.

3.2.1 Implementation and Environment

The experiments reported in this dissertation were performed with the following benchmark configuration: The database was populated with data according to the TPC-W database scale with 100 EBs. This database involves 10,000 items and has 315 MB of raw data. The total database size is about 1 GB with indexes. In all experiments the TPC-W Ordering mix was used because it is the most update-intensive mix: About one third of the requests involve an update to the database. The consistency level of the AWS SimpleDB and S3 variants is low, while all other variants provide ACID guarantees and thus, have a high consistency level (Table 2.1). Consequently, we expect SimpleDB and S3 to provide significantly higher performance.

Our benchmark measures WIPS and cost, thereby varying the number of EBs (i.e., number of simulated concurrent users). As mentioned previously, we varied the load from 1 EB
(light workload) to 9000 EBs (heavy workload). In summary, the following metrics were measured:

- **WIPS(EB):** The throughput of valid web interactions per second depending on the number of EBs (valid means “within the TPC-W response time threshold” as described previously). The higher, the better.
- **$/WIPS(EB):** The cost per WIPS, again depending on the number of EBs. The lower, the better.
- **CostPerDay(EB):** The (projected) total cost of running the benchmark with a certain number of EBs for 24 hours. The lower, the better.
- **$\sigma$/WIPS:** The standard deviation of the $$/WIPS for a set of different EB settings (from EB=1 to EB=max where max is the EB value for which the highest throughput could be achieved). This metric is a measure for the predictability of cost. The lower $\sigma$, the better.

In addition to these metrics, we measured the time and cost to bulk-load the benchmark database as well as the size and monthly cost to store the benchmark database.

Depending on the variant, we had two different experimental setups in order to determine the cost and WIPS for each EB setting. For the SimpleDB, S3, and Google App Engine variants, we measured the WIPS for a number of EB settings (EB=1, 250, 500, 1000, 2000, 3000, 4000, and 9000, if possible) during a period of 10 minutes. For these variants, we were not able to measure the whole spectrum of EB settings for budget reasons. For the three MySQL variants (MySQL, MySQL/R, and RDS) and Azure SQL Database, we measured all possible EB settings in the range of EB=1 to EB=9000. This was done by starting with EB=1 and increasing the workload by one EB every 0.4 seconds. In all cases, we did a warm-up run of two minutes before each experimental run. The cost and throughput of this two minute warm-up phase are factored out in the results presented.

We did a number of additional experiments and measures in order to guarantee the stability of the results. For MySQL, MySQL/R, RDS, and Azure SQL Database, all experiments were repeated seven times and the average WIPS and cost of these seven runs are reported in this dissertation. For the SimpleDB, S3, Google AE, and Google AE/C variants, all experiments were repeated only three times, again, because of budget constraints. Furthermore, we ran several data points for longer periods of time (up to thirty minutes) with
3.2. Experiments and Results

a fixed EB setting in order to see whether the providers would adjust their configuration to the workload. However, we could not detect any such effects. Only for Microsoft Azure, we observed a small discontinuity. In our first experiments, Azure became shortly unavailable for EB=2000 and EB=5500. We believe that at these points, Azure migrated the TPC-W database to bigger machines so that the increased workload could be sustained. This effect happened only for the very first experiment. It seems that Azure does not migrate databases back to less powerful machines when the workload decreases. All subsequent experiments on Azure were carried out on the (presumably) big database machine. In general, the observed results were surprisingly stable, and we had only one outlier in one of the MySQL experiments. It is well known that the quality of service of cloud computing providers varies, but a long term detailed study on these variances is beyond the scope of this work.

In all experiments, the emulated browsers of the TPC-W benchmark were run on EC2 machines in the Amazon cloud. For the experiments with Google App Engine and Azure, the client machines were therefore geographically located in different data-centers than the server machines that handled the requests. For fairness, we made sure that EC2 client machines and EC2 server machines were located in different data-centers for all Amazon variants. This was done by explicitly choosing a data-center when starting the EC2 instances for clients and servers.

We used medium compute optimized (c1.medium) EC2 machines to run the web/app servers in the Amazon cloud. Accordingly, we used medium machines to run the web/app servers in the Azure cloud. The medium EC2 and Azure machines have roughly the same performance and cost so that the results are directly comparable. Furthermore, we adjusted the number of machines that ran the web/app servers manually in the Amazon and Azure clouds. With the help of separate experiments (not reported here), we discovered that one medium instance was able to sustain a load of 1500 EBs. Accordingly, we provisioned one web/app server per 1500 EBs, up to six machines for the maximum workload of 9000 EBs. In the S3 variant, an integrated web/app/DB server was only able to sustain 900 EBs so that up to ten EC2 machines were used for this variant. We pre-allocated EC2 and Azure machines in order to make sure that they were available when needed (as the load increased). In the experiments with Google App Engine, we had no influence on the choice and number of machines used because the servers were automatically provisioned by Google (Table 2.1).
We also used medium EC2 machines in order to run database servers for AWS MySQL and AWS MySQL/R. If not stated otherwise, we used a large (db.m1.large) instance for AWS RDS because it has similar performance characteristics as a medium EC2 instance. For all other variants, the database machines could not be configured.

As mentioned earlier, we did not conduct any “peak” experiments as proposed in Section 3.1.3.2. The purpose of “peak” experiments is to evaluate how quickly a provider adapts to sudden changes in the workload. As mentioned above, we could not observe any significant adjustments by the providers (with the noteworthy exception of Azure in the first experimental run) so that we believe that our results represent the steady-state behavior of the systems. However, a more detailed analysis of peak performance and adaptability is an important avenue for future research.

In all experiments, the images used as part of the TPC-W benchmark (e.g., pictures of products) were stored on a separate file system and not inside the database. In the Amazon cloud, all images were stored on the local EC2 filesystem to save cost. In Azure, the images were stored as part of the web project.

### 3.2.2 Big Picture

Table 3.1 summarizes the overall results of this study. More detailed analyses are given in the subsequent subsections. The first column shows the maximum throughput (WIPS) that could be achieved for each variant. The second and third columns list the cost per

<table>
<thead>
<tr>
<th></th>
<th>Throughput (WIPS)</th>
<th>$/WIPS Low TP</th>
<th>$/WIPS High TP</th>
<th>Cost Predictability (mean ± σ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS MySQL</td>
<td>477</td>
<td>0.635</td>
<td><strong>0.005</strong></td>
<td>0.015 ± 0.077</td>
</tr>
<tr>
<td>AWS MySQL/R</td>
<td>454</td>
<td>2.334</td>
<td><strong>0.005</strong></td>
<td>0.043 ± 0.284</td>
</tr>
<tr>
<td>AWS RDS</td>
<td>462</td>
<td>1.212</td>
<td><strong>0.005</strong></td>
<td>0.030 ± 0.154</td>
</tr>
<tr>
<td>AWS SimpleDB</td>
<td>128</td>
<td>0.384</td>
<td>0.037</td>
<td>0.063 ± 0.089</td>
</tr>
<tr>
<td>AWS S3</td>
<td>&gt;1100</td>
<td>1.304</td>
<td>0.009</td>
<td>0.018 ± 0.098</td>
</tr>
<tr>
<td>Google AE</td>
<td>39</td>
<td>0.002</td>
<td>0.042</td>
<td>0.029 ± 0.016</td>
</tr>
<tr>
<td>Google AE/C</td>
<td>49</td>
<td><strong>0.002</strong></td>
<td>0.028</td>
<td><strong>0.021 ± 0.011</strong></td>
</tr>
<tr>
<td>MS Azure</td>
<td>&gt;1100</td>
<td>0.775</td>
<td><strong>0.005</strong></td>
<td>0.010 ± 0.058</td>
</tr>
</tbody>
</table>

**Table 3.1:** Throughput, $/WIPS, and cost predictability (Feb. 2010)
3.2. Experiments and Results

WIPS for low workloads (EB=1, second column) and high workloads (EB=max, third column). The fourth column shows the mean and standard deviation of the cost for the whole range of workloads (from EB=1 to EB=max). EB=max refers to the maximum number of EBs that a variant could sustain. That is, the EBs at which the maximum throughput was achieved. In all columns, the winner is highlighted in bold and italics.

Looking at the first column (for throughput), it becomes clear that only S3 and Azure are able to sustain high workloads of 9000 EBs. For all other variants, the database layer becomes a bottleneck with a growing number of EBs. We believe that the S3 variant with a shared-data architecture is able to scale even beyond 9000 EBs. This architecture is the only architecture that has no potential bottleneck. Since Azure uses replication (Section 2.2.4), it reaches its limits as soon as the master database server is overloaded. It seems, however, that Microsoft makes use of high-end machines for the Azure SQL Database service so that this limit was not reached for 9000 EBs.

Turning to the $/WIPS results, it can be observed that all services, except Google AE, have lower cost per WIPS at high workloads than at low workloads. Ideally, the $/WIPS should be constant and should not depend on the workload. If the $/WIPS is higher for a low workload than for a high workload, then the service has fixed cost that have to be paid independent of the usage of the service. For instance, all Amazon variants require at least one EC2 instance in order to be able to respond to client requests even if there is no load at all. In addition, some Amazon variants must pay for instances at the database layer. Likewise, a monthly flat fee for Azure SQL Database and at least one machine for a web/app server must be paid in order to keep a web application online in the Azure cloud. Google App Engine is the only variant that does not have any fixed cost and is free if there is no load. Obviously, fixed costs are not compliant with the pay-as-you-go paradigm.

The fourth column of Table 3.1 shows the mean and standard deviation of the $/WIPS metric over the whole range of EBs that a service can sustain. With the exception of Google, all services have a high variance. This means that the cost for the service is highly dependent on the load and thus, becomes unpredictable (unless the system has a constant load).
Figure 3.5: Throughput comparison of cloud service architectures

3.2.3 Scale-Out

Figure 3.5 shows the WIPS achieved by each variant as a function of EB. For readability, Figure 3.5 does not show the results for Google App Engine without caching and MySQL/R. Google AE without caching showed almost the same performance as Google AE with caching. The achieved throughput without caching was a little worse, but the differences were marginal and the shape of the curve is almost identical. MySQL/R showed almost the same performance as MySQL, again, slightly worse in all cases. For update-intensive workloads such as the TPC-W Ordering mix, the master server becomes a bottleneck and scaling out read-only transactions to replicas does not result in throughput gains.

Figure 3.5 shows that the only two variants that scale are S3 and Azure. As mentioned in the previous subsection, S3 is the only variant that is based on an architecture that has no bottlenecks. Azure scales well for using powerful machines to run the database servers. Both of these variants scale almost ideally up to 9000 EBs and achieve the maximum throughput at 9000 EBs. For reference, the “ideal” throughput of a perfect system is shown as a dotted line in Figure 3.5. All other variants have scalability limits and cannot sustain the load after a certain number of EBs. The baseline variants, MySQL and RDS,
3.2. Experiments and Results

The behavior of the alternative variants in overload situations is surprisingly different. Figures 3.6 to 3.9 depict the throughput behavior of AWS S3, RDS, SimpleDB, and Google App Engine in more detail. These figures show the ideal throughput (dotted line), the WIPS in RT (red line), and the number of issued requests that were submitted (green line). Recall that the TPC-W benchmark specifies that a client (i.e., an emulated browser) waits for a response before issuing the next request (Section 3.1.2). Thus, if a system cannot sustain the load and does not produce responses anymore, then the number of issued requests is lower than in an ideal system. Obviously, the WIPS in RT must be equal or lower than the issued requests. Figures 3.6 to 3.9 show the following effects in overload situations for the different variants:

Figure 3.6: AWS S3 throughput

reach their limits at about 3500 EBs, SimpleDB at about 3000 EBs, and Google AE/C at a few hundred EBs.
• S3 (Figure 3.6): We were not able to produce overload situations for this variant. As shown in Figures 3.5 and 3.6, the throughput scales almost linearly with the load. However, recall that the S3 variant provides weak consistency guarantees. Thus, good scalability does not come as a surprise.

• RDS (Figure 3.7): The throughput of RDS plateaus after 3500 EBs and stays constant. That is, all requests return answers, but a growing percentage of requests are not answered within the response time constraints specified by the TPC-W benchmark. The MySQL variant is technically the same as RDS and therefore has the same behavior.

• SimpleDB (Figure 3.8): Figure 3.5 shows that the WIPS in RT grow up to about 3000 EBs and more than 200 WIPS. In fact, SimpleDB was already overloaded at about 1000 EBs and 128 WIPS in our experiments. At this point, all write requests to hot spots failed. In the TPC-W benchmark, item objects are frequently updated and these update requests were dropped by SimpleDB. According to the rules of the TPC-W benchmark, dropping more than 10% of the requests of any category is illegal (Section 3.1.2). As a result, Table 3.1 reports on a peak throughput of 128 WIPS for SimpleDB. In an overload situation, SimpleDB simply drops requests and returns errors. As failure is immediate, the issued requests grow linearly with the number of EBs. The low throughput of SimpleDB is all the more surprising as the service only provides eventual consistency.

DynamoDB suffers from the same problems as SimpleDB. For DynamoDB, the user has to specify the expected throughput for each table. We distributed 20,000 read and write Provisioned Throughput Capacity (PTC) among the TPC-W tables according to the expected load. We did not benchmark with more PTC because of budget constraints. Unfortunately, the provisioned capacity limit was quickly reached and requests got dropped as with SimpleDB. In summary, DynamoDB performed slightly better than SimpleDB and reached a peak throughput of 168 WIPS at about 1500 EBs.

• Google AE/C (Figure 3.9): Like SimpleDB, Google App Engine (with and without caching) drops requests in overload situations. This effect can be observed by a linearly growing issued requests curve in Figure 3.9. Unlike SimpleDB, Google App Engine is fair and drops requests from all categories. That is, both read and
3.2. Experiments and Results

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>10</th>
<th>100</th>
<th>500</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS MySQL</td>
<td>0.635</td>
<td>0.072</td>
<td>0.020</td>
<td>0.006</td>
<td>0.006</td>
</tr>
<tr>
<td>AWS MySQL/R</td>
<td>2.334</td>
<td>0.238</td>
<td>0.034</td>
<td>0.008</td>
<td>0.006</td>
</tr>
<tr>
<td>AWS RDS</td>
<td>1.211</td>
<td>0.126</td>
<td>0.032</td>
<td>0.008</td>
<td>0.006</td>
</tr>
<tr>
<td>AWS SimpleDB</td>
<td>0.384</td>
<td>0.073</td>
<td>0.042</td>
<td>0.039</td>
<td>0.037</td>
</tr>
<tr>
<td>AWS S3</td>
<td>1.304</td>
<td>0.206</td>
<td>0.042</td>
<td>0.019</td>
<td>0.011</td>
</tr>
<tr>
<td>Google AE</td>
<td>0.002</td>
<td>0.028</td>
<td>0.033</td>
<td>0.042</td>
<td>0.176</td>
</tr>
<tr>
<td>Google AE/C</td>
<td>0.002</td>
<td>0.018</td>
<td>0.026</td>
<td>0.028</td>
<td>0.134</td>
</tr>
<tr>
<td>MS Azure</td>
<td>0.775</td>
<td>0.084</td>
<td>0.023</td>
<td>0.006</td>
<td>0.006</td>
</tr>
</tbody>
</table>

Table 3.2: Cost per WIPS in m$ (vary EB, Feb. 2010)

write requests of all kinds are dropped in overload situations. In these scale-out experiments, Google AE performs worst among all variants. This phenomena can be explained by Google’s focus on supporting low-end workloads for the lowest possible price (see next section).

3.2.4 Cost

In this section, we evaluate the cost of all eight variants. First, we discuss the cost for a given throughput ($/WIPS). Then, we present the projected cost per day. Finally, we detail the different cost factors for the alternative architectures.

3.2.4.1 Cost/WIPS

Table 3.2 details the $/WIPS for the different variants for varying EBs. As discussed in Section 3.2.2, Google AE is cheapest for low workloads (below 100 EBs), whereas Azure is cheapest for medium to large workloads (more than 100 EBs). The three MySQL variants (MySQL, MySQL/R, and RDS) have (almost) the same cost as Azure for medium workloads (EB=100 and EB=1000), but they are not able to sustain large workloads (Section 3.2.3).

The success of Google AE for small loads has two reasons. First, Google AE is the only variant that has no fixed costs. There is only a negligible monthly fee to store the database (Table 3.6). Second, at the time these experiments were carried out, Google gave a quota
of six CPU hours per day for free. That is, applications that are below or slightly above this daily quota are particularly cheap.

Azure and the MySQL variants win for medium and large workloads because all these approaches can amortize their fixed cost for these workloads. Azure SQL Database has a fixed cost per month of USD 100 for a database of up to 10 GB, independent of the number of requests that needs to be processed by the database. For MySQL and MySQL/R, EC2 instances must be rented in order to keep the database online. Likewise, RDS involves an hourly fixed fee so that the cost per WIPS decreases when the load increases. It should be noted that network traffic is cheaper with Google than with both Amazon and Microsoft.

The cost analysis is an artifact of the business models used by Amazon, Google, and Microsoft. Obviously, all providers can change their pricing any time, and they have frequently done so in the past. However, the cost analysis indicates for which kind of workload a service is optimized: Google is apparently targeting the low-end market, whereas Microsoft seems to be focusing on enterprise customers. Furthermore, we believe that cost is indeed a good indicator for the efficiency of an implementation.

### 3.2.4.2 Cost per Day

Table 3.3 and 3.4 show the total cost per day for the alternative approaches and a varying load (EBs). Table 3.3 presents the original cost at the time of measurement (i.e., February 2010), while Table 3.4 estimates the cost based on new pricing schemes from October 2014. A “-” indicates that the variant was not able to sustain the load. Most of the services, cost models, and VM instances used in the experiments were still available in October 2014, and therefore Table 3.4 provides an accurate cost estimation based on new pricing schemes. Azure is an exception because Microsoft has retired the old virtual machines and changed the cost model of the SQL Database service. Thus, to estimate the cost, we selected a virtual machine type (windows A1) and a SQL Database (standard S2) with similar hardware and performance properties as those used during the experiments.

The results of Table 3.3 and 3.4 confirm the observations made previously: Google wins for small workloads, Azure wins for medium and large workloads. All the other variants are somewhere in between. The three MySQL variants come close to Azure in the range of workloads that they sustain. Again, Azure and the three MySQL variants roughly share the same architectural principles (classic combined with replication). SimpleDB is an outlier in this experiment and an exceptionally expensive service. For a large number of EBs,
3.2. Experiments and Results

<table>
<thead>
<tr>
<th>Service</th>
<th>1</th>
<th>250</th>
<th>1000</th>
<th>3000</th>
<th>9000</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS MySQL</td>
<td>8.25</td>
<td>64.48</td>
<td>183.66</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>AWS MySQL/R</td>
<td>16.40</td>
<td>72.90</td>
<td>194.44</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>AWS RDS</td>
<td>14.75</td>
<td>70.92</td>
<td>187.88</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>AWS SimpleDB</td>
<td>4.42</td>
<td>412.25</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>AWS S3</td>
<td>16.02</td>
<td>131.78</td>
<td>346.60</td>
<td>914.87</td>
<td></td>
</tr>
<tr>
<td>Google AE</td>
<td>0.02</td>
<td>82.25</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Google AE/C</td>
<td>0.02</td>
<td>60.57</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>MS Azure</td>
<td>9.15</td>
<td>61.22</td>
<td>176.87</td>
<td>521.13</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.3: Total cost per day in $ (vary EB, Feb. 2010)

<table>
<thead>
<tr>
<th>Service</th>
<th>1</th>
<th>250</th>
<th>1000</th>
<th>3000</th>
<th>9000</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS MySQL</td>
<td>4.24</td>
<td>15.94</td>
<td>49.48</td>
<td>144.17</td>
<td>-</td>
</tr>
<tr>
<td>AWS MySQL/R</td>
<td>8.41</td>
<td>53.94</td>
<td>150.86</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>AWS RDS</td>
<td>7.70</td>
<td>55.22</td>
<td>145.93</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>AWS SimpleDB</td>
<td>2.42</td>
<td>399.44</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>AWS S3</td>
<td>4.57</td>
<td>53.61</td>
<td>155.02</td>
<td>453.76</td>
<td></td>
</tr>
<tr>
<td>Google AE</td>
<td>0.02</td>
<td>44.79</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Google AE/C</td>
<td>0.02</td>
<td>33.90</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>MS Azure</td>
<td>5.90</td>
<td>49.78</td>
<td>139.23</td>
<td>409.96</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.4: Total cost per day in $ (vary EB, Oct. 2014)

The high cost of SimpleDB is particularly annoying because users must pay even though SimpleDB drops many requests and is not able to sustain the workload. Surprisingly, Table 3.3 and 3.4 enable the same conclusions. Although cloud prices decreased over the last years, they proportionally dropped at the same rate for all services. This is an effect of high competition in the cloud market. As soon as a provider cuts prices, the competitors immediately respond accordingly.

Turning to the S3 cost in Table 3.3, the total cost grows linearly with the workload. This behavior is exactly what one would expect from a pay-as-you-go model. In 2010, the high cost of S3 is matched by high throughputs so that the cost at high workloads is tolerable. Nevertheless, S3 was indeed more expensive than all the other approaches (except SimpleDB) for most workloads. Due to competing services (Azure Cloud Storage...
Chapter 3. Evaluation of Cloud Data Services

and Google Cloud Storage), Amazon was forced to considerably reduce the price for the S3 service in the last years. As a result, the cost of the S3 variant significantly dropped and is nowadays in the same order as Azure (Table 3.4).

3.2.4.3 Cost Analysis

Figure 3.10 and 3.11 show the cost per day spent on network traffic, CPU, and storage. Again, to highlight the changes over time, Figure 3.10 and 3.11 are based on the pricing schemes of February 2010 and October 2014 respectively. The network traffic is depicted in green. Network traffic costs are purely variable costs that entirely depend on the workload. The CPU costs have a variable and a fixed component. The fixed CPU costs are shown in red, and the variable CPU costs are shown in blue. Fixed CPU costs are required to reserve machines. Variable CPU costs are incurred by surcharges of actual usage. For instance, the cost of Google AE or SimpleDB depend on the usage. Storage costs also have a fixed and variable component. The fixed storage costs are the monthly costs for storing the database. The variable storage costs are the costs per request to fetch and put data into the database. As will be shown in Section 3.2.6, the fixed storage costs were negligible for our experiments so that Figure 3.10 and 3.11 show the aggregated storage cost (fixed + variable) as a single storage cost metric in purple. Figure 3.10 and 3.11 show the cost breakdown by resource (network, CPU, and storage) for each variant for EB=250. Again, the results for MySQL/R and Google AE without caching are omitted for brevity (they are similar to MySQL and Google AE/C, respectively).

Figure 3.11 confirms the previous observations and highlights that the prices for all cost factors have been lowered over time (in comparison to Figure 3.10). Most notably, the cost for CPU (fixed + variable) and for storage have been affected by tight competition. Network costs have only been lowered to a limited extent.

Figure 3.10 and 3.11 show that the total cost for the MySQL variant is dominated by network costs for EB=250. Another significant factor is the (fixed) cost for the reservation of EC2 instances for the web/app and database servers. All other cost factors are negligible. For SimpleDB, the total cost is dominated by the (variable) cost for the compute hours spent in order to query and update SimpleDB. As mentioned previously, SimpleDB is an expensive service. The remaining SimpleDB cost factors are in the same order as for the other variants. For the S3 variant, the major cost factor in Figure 3.10 is storage. The costs of storage are a direct effect of the high price to retrieve and put data into the S3
service. All other cost factors of the S3 variant are comparable to the cost factors of the other variants implemented on the Amazon cloud. The price for using the S3 service has considerably dropped in Figure 3.11, and consequently the S3 variant has become more attractive in recent years.
Looking at the Google cloud, it can be seen that Google AE is the only variant that has no fixed costs for reserving CPUs. All the CPU cost is variable but, even at a moderate workload of 250 EBs, it can be considerable. Nevertheless, the absence of any fixed cost (there is only a negligible fixed cost for storing the database), makes Google highly attractive for small applications.

Again, Azure seems to be targeting a complementary market segment compared to Google. Microsoft charges a fixed price to reserve machines for the application and database servers. However, once this fee has been paid, there are no variable costs for actually using these machines.

Figures 3.12 to 3.17 confirm these results. These figures visualize the percentage of each cost factor depending on the workload. The distribution of cost factors is based on the pricing schemes of February 2010. Figures 3.12 and 3.13 for MySQL and RDS are mostly green as network costs dominate the overall cost for these two variants. Only for small
workloads (EB < 100), the cost for reserving machines is significant (depicted red). An interesting effect is the zig-zag in both figures: Every step (at a spacing of 1500 EBs) indicates that an additional EC2 instance was added at the web/app layer in order to sustain the additional workload.

Figure 3.14 confirms that the cost of the SimpleDB variant is dominated by the CPU time resulting from querying and updating the SimpleDB service. Accordingly, Figure 3.15 shows that the cost of the shared-data architecture implemented on top of S3 is dominated by the usage of the S3 service. Taking into account the price cuts in recent years, the dominating cost factor of the S3 service has shifted from storage to network (Figure 3.11).

Figure 3.16 confirms that the dominant cost for Google AE is the (variable) CPU cost. Only for very small workloads (EB < 10), the cost of Google AE/C is dominated by storage costs (red), but for these workloads the overall cost is negligible (in the order of a few cents). In contrast, Figure 3.17 is mostly green, confirming that the cost of Azure is dominated by network traffic just as for the MySQL variants. Figure 3.17 also has the same zig zag as Figures 3.12 and 3.13 for adding a new web/app server each 1500 EBs.

The costs are an artifact of the pricing models of the providers. The results are likely to change in the future. Nevertheless, it is important to understand the cost factors of a system. Over the last years, for instance, the cost for CPU time has dropped faster than the cost for network traffic.
Chapter 3. Evaluation of Cloud Data Services

3.2.5 Scale-Up

One of the goals of cloud computing is to scale out. That is, increase throughput by adding new machines. Nevertheless, some cloud providers also support scale-up options. For the classic and partitioning architectures described in Section 2.2, such scale-up options are important because the database server can become a bottleneck and scaling up is the only way to achieve higher throughput. Concretely, Amazon RDS provides a scale-up option. Table 3.5 presents the scale-up results for throughput, \$/WIPS, and cost predictability. It can be seen that a larger database machine is able to sustain a higher workload. However, even the biggest RDS instance is not able to sustain 9000 EBs and a throughput of more than 1000 WIPS. Obviously, a bigger database server increases the fixed cost of a system (poor \$/WIPS for low workloads) and consequently lowers cost predictability (i.e., increases the variance).

New hardware and technology trends are continuously integrated in the infrastructure of cloud providers. In fact, Amazon has recently released several new RDS instances with stronger CPUs and more memory. These instances are built on the newest hardware generation and are likely to provide more throughput than the instance types evaluated in Table 3.5. Nonetheless, the architectural properties remain the same. That is, in a classic architecture, the database layer is a potential bottleneck.

3.2.6 Bulk-Loading

For completeness, Table 3.6 shows the bulk-loading times and monthly storage costs for the alternative variants. Again, the winners are shown in bold and italics. In all cases, the best possible offering for bulk-loading the database was used. For MySQL, the data was inserted using a JDBC connection. For SimpleDB, we used the batch-put operation that
### 3.3. Concluding Remarks

<table>
<thead>
<tr>
<th></th>
<th>BL Time (h:mm)</th>
<th>BL Cost ($)</th>
<th>DB Size (GB)</th>
<th>Storage Cost / Month ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS MySQL</td>
<td>0:10</td>
<td>0.17</td>
<td>0.81</td>
<td>&gt; 0.1</td>
</tr>
<tr>
<td>AWS MySQL/R</td>
<td>0:30</td>
<td>0.51</td>
<td>2.43</td>
<td>0.1</td>
</tr>
<tr>
<td>AWS RDS</td>
<td>0:07</td>
<td>0.46</td>
<td>-</td>
<td>0.1</td>
</tr>
<tr>
<td>AWS SimpleDB</td>
<td>1:50</td>
<td>5.16</td>
<td>1.32</td>
<td>0.33</td>
</tr>
<tr>
<td>AWS S3</td>
<td>1:33</td>
<td>0.44</td>
<td>0.99</td>
<td>0.15</td>
</tr>
<tr>
<td>Google AE</td>
<td>30:10</td>
<td>11.69</td>
<td>4.28</td>
<td>0.64</td>
</tr>
<tr>
<td>Google AE/C</td>
<td>30:10</td>
<td>11.69</td>
<td>4.28</td>
<td>0.64</td>
</tr>
<tr>
<td>MS Azure</td>
<td>0:23</td>
<td>0.36</td>
<td><strong>0.38</strong></td>
<td><strong>0.0</strong></td>
</tr>
</tbody>
</table>

**Table 3.6:** Bulk-loading time and cost, database size, and storage cost (Feb. 2010)

is able to store 25 records at once. S3 has no specific bulk-loading support so that the data was loaded using the standard protocol of [BFG+08]. For Google AE, we made use of a Python script to enable Google’s AE Datastore batch import. For Azure, the Azure SQL Database Migration Wizard was used. In summary, MySQL was the clear winner with regard to bulk-loading time. MySQL/R had exactly three times the time and cost of MySQL because we had a replication factor of three.

Turning to the size of the database and the storage cost per month (the last two columns of Table 3.6), it can be seen that Azure is the winner. However, the storage costs are negligible and are not an important factor for the overall cost (Section 3.2.4). It should be noted that for all variants, except MySQL, the monthly storage costs grow linearly with the size of the database. The MySQL variant is based on EBS for storing the database (Section 2.3) and the cost function of EBS is more complex. In a prudent set-up, EBS resources are over-provisioned because if an EBS device is full, then a new EBS device with larger capacity must be provisioned and the data must be copied.

### 3.3 Concluding Remarks

This chapter introduced a new benchmark methodology for cloud computing. We presented a workload based on the popular TPC-W benchmark and proposed new metrics and experiments to assess cloud data services. We hope that this work contributes to a continuous monitoring of alternative services for data management in the cloud.
Furthermore, we presented the results of a first end-to-end study of the performance and cost of running an enterprise OLTP workload on multiple cloud services. The alternative services varied greatly both in cost and performance. Most services had significant scalability issues. An interesting observation was to see how the alternative services behave in overload situations. With regard to cost, it became clear that the alternative providers have different business models and target different kinds of applications: Google seems to be more interested in small applications with light workloads, whereas Microsoft Azure is the most affordable service for medium to large applications.

The more fundamental question of which is the right data management architecture for cloud computing could not be fully answered. With regard to throughput, the shared-data architecture (i.e., the AWS S3 variant) yielded promising results. However, it is unclear whether the observed results for this variant are an artifact of the weak level of consistency or fundamental to the chosen architecture. The next chapter dives deeper into the elemental properties of the shared-data architecture. We present and evaluate a distributed database systems that enables strong consistency guarantees and ACID transactions on top of shared data.
Tell: A Distributed Shared-Data Database

The shared-data architecture presented in Section 2.2 enables fundamental benefits with regard to flexible data management. In particular, decoupling query processing and transaction management from data storage and sharing data across all database instances eliminates many potential bottlenecks. In fact, both layers, database and storage, can be scaled out and scaled down independently. In Section 3.2 we have demonstrated that a system [BFG+08] based on the shared-data architecture enables scalable OLTP processing for weak consistency levels. In this chapter, we present a distributed database system, called Tell, that enables efficient transaction processing on top of shared data. Tell provides the rich features of classical RDBMS and in addition enables elastic resource management as required in dynamic cloud environments.

The challenges of sharing data are addressed by combining specific techniques with new hardware trends. For instance, distributed data synchronization is enabled by lightweight atomic RMW primitives [SVdWB08]. We use atomic RMW as a key operation to implement a MVCC protocol [BG81] and provide latch-free indexing structures [LSL13]. Moreover, we follow new technical trends and use fast networking technology (i.e., InfiniBand [Inf14]) as well as in-memory data storage to keep data access latencies as low as possible. It is the specific combination of the right techniques with new hardware trends that enables scalability not possible a few years ago.
Although we primarily focus on OLTP workloads, this dissertation outlines the implications of performing mixed workloads. That is, we analyze how to run an OLTP workload and perform analytical queries on separate instances but accessing the same single copy of the data.

The chapter is organized as follows: First, we detail key design principles and technical challenges of shared-data architectures. Second, we review related work. Then, we describe Tell and explain how concurrency control and recovery are implemented. Fourth, we detail data access and shared storage. Finally, we evaluate the effectiveness of Tell and compare it to state-of-the-art partitioned databases.

4.1 Design Principles

In the following, we detail the key design principles that motivated the design of Tell. Moreover, we highlight the necessity for ACID transactions and complex queries, two features that have recently been relaxed or simplified in many storage systems [DHJ+07, CDG+08, RST11].

4.1.1 Shared Data

Shared data implies that every database instance can access and modify all data stored in the database. There is no exclusive data ownership. Database instances can execute any query, and therefore a transaction can be executed and committed/aborted by a single instance. In contrast to partitioned databases, no application knowledge and no expert skills are required to correctly partition data and minimize the number of distributed transactions. Consequently, data sharing provides the following benefits: First, setting up a database cluster is straightforward. Second, the interaction with the database is simplified as partitioning is no longer reflected in the application logic. On the other hand, sharing data requires updates to be synchronized, a constraint we address in Section 4.2.

4.1.2 Decoupling Query Processing and Storage

The shared-data architecture is decomposed into two logically independent layers, transactional query processing and data storage. The storage layer is autonomous. It is implemented as a self-contained system that manages data distribution and fault-tolerance
transparency with regard to the processing layer. As a result, replication and data redistribution tasks are executed in the background without the processing layer being involved. The storage system is in essence a distributed record manager that consists of multiple storage nodes (SN). Data partitioning is not as performance critical as in partitioned databases as data location does not determine where queries have to be executed. Instead, to execute queries, the processing layer accesses records independent of the SNs they are located at. This fundamental difference in the communication pattern is highlighted in Figures 2.2 and 2.3.

The processing layer consists of multiple autonomous processing nodes (PN) that access the shared storage system. A mechanism is provided to retrieve data location (e.g., a lookup service) that enables the processing nodes to directly contact the storage node holding the required data.

Logical decoupling considerably improves agility and elasticity as PNs or SNs can be added on-demand if processing resources or storage capacity is required respectively. Accordingly, nodes can be removed if capacity is not required anymore. At the same time, the architecture enables workload flexibility. That is, the possibility to execute different workloads on separate nodes. For example, some PNs can run an OLTP workload, while others perform analytical queries on the same dataset.

### 4.1.3 In-Memory Storage

Commodity hardware clusters available today typically have several Terabytes of main memory at their disposal. These numbers surpass the capacity requirements of most applications. For instance, in the popular TPC-C benchmark [Tra10], a warehouse with thousands of items requires less than 200 MB. Consequently, it is no longer necessary to rely on slow storage devices (e.g., hard disks), and instead the whole data can be kept in main memory. The advantages are obvious: Main memory storage provides lower access latencies and avoids complex buffering mechanisms [SMA*07]. However, DRAM memory is volatile and data loss must be prevented in case of failures. A common approach to ensure fault-tolerance is data replication (Section 4.4.4).
4.1.4 ACID Transactions

Transactions ensure isolated execution of concurrent operations and maintain data integrity. From a developer’s perspective, transactions are a convenient way to perform consistent data changes and simplify application development. In particular, development is less error-prone as data corruption and anomalies do not occur. Transactions are indispensable for the majority of applications and are therefore a major feature.

Recently, relational databases have been challenged by the emergence of NoSQL data stores. As previously highlighted, NoSQL stores relax consistency guarantees and renounce the benefits mentioned beforehand in order to achieve high scalability and availability. These systems provide limited transactional properties (e.g., transactions on single records) if at all and are therefore not suited for many business relevant use-cases (e.g., financial transactions, product orders, or payment processes).

We aim at providing full transactional support. That is, ACID transactions can be performed without limitations on the whole data. Transaction management is part of the processing layer and does not make any assumption on how data is accessed and stored. How to achieve scalability while at the same time being highly consistent is one of the challenges addressed in the Section 4.2.

4.1.5 Complex Queries

SQL is the standard query language in relational databases. It enables complex queries and includes operators to order, aggregate, or filter records based on predicates. Although alternative storage systems often simplify the query model to improve performance, complex (SQL) queries are not an obstacle to system scalability [Rys11]. In the shared-data architecture, data location is logically separated from query processing, and as a result a PN retrieves the records it requires to execute a query. This notion can be described as data is shipped to the query and not the query to the data. The agility to run the same query on multiple nodes provides increased parallelism and enables scalability.

4.2 Technical Challenges

Although shared-data architectures avoid the limitations of their partitioned counterpart, namely Sharding and weak elasticity, they introduce new challenges.
4.2. Technical Challenges

4.2.1 Data Access

In a shared-data environment, data is likely to be located at a remote location and has to be accessed over the network. Caching data in local buffers is only possible to a limited extent as updates made by one PN have to be visible to the others instantly. As a result, to provide consistent access, most requests retrieve the latest record version remotely from the storage layer. Considering that processing a query usually requires multiple records, data access latencies can quickly become a dominant factor in query execution time. Therefore, an optimization goal in Tell is to minimize the interaction with the storage layer and to prevent unnecessary network requests.

In light of restricted buffering possibilities, traditional techniques to manage relational data have to be re-evaluated. A major concern in this context is the correct granularity of data storage. That is, whether it is beneficial to group records into pages in order to access several records with a request or whether choosing a finer storage granularity (e.g., single record) is a more favorable approach. We suggest to store data at the granularity of a record as this provides a good trade-off between the number of network requests and the amount of traffic generated. We justify this design choice and detail the implications in Section 4.5.

A second challenge of shared data relates to data access paths and indexing. Analogous to data, indexes are shared across multiple PNs and are therefore subject to concurrent modifications from distributed locations. As a result, the shared-data architecture requires distributed indexes that support atomic operations and that are at the same time highly scalable. As a solution, we propose a scalable latch-free distributed B+tree index in Section 4.5.3.

In addition to the right techniques, state-of-the-art networking technologies such as 10 Gbit Ethernet or InfiniBand contribute to reduce the overhead of remote data access. These technologies provide latency and bandwidth guarantees not available a decade back and consequently enable the shared-data architecture to scale to new levels. For instance, InfiniBand performs Remote Direct Memory Access (RDMA) in a few microseconds and is three order of magnitudes faster than a random read on a local hard-disk.
4.2.2 Concurrency Control

Shared data records can be updated by any processing node, and therefore concurrency control is required across all PNs. Although common pessimistic and optimistic concurrency control approaches [BHG87] are suited, distribution requires the mechanisms to be examined from a new angle. For instance, a lock-based approach requires centralized lock management. Assuming network latency dominates lock acquisition time, the lock manager quickly becomes a bottleneck that limits transaction throughput. Similarly, increased data update latencies might cause more write-conflicts in an optimistic protocol resulting in higher abort rates. Accordingly, a concurrency control mechanism that minimizes the overhead of distribution allows for the highest scalability.

In Section 4.4, we present a distributed MVCC protocol. A key feature is conflict detection using atomic RMW primitives. Atomic RMW operations have become popular in recent years as they enable non-blocking synchronization. In a shared-data architecture, they allow to perform data updates only if a record has not been changed since it has last been read. As a result, conflicts can be identified with a single call. Atomic RMW primitives are a lightweight mechanism that allows for efficient distributed concurrency control.

4.2.3 Limitations

The architecture and the techniques presented in this chapter perform best in the context of local area networks (LAN). LANs allow for low-latency communication and are a critical performance factor for several reasons: First, shared data implies limited buffering and causes heavy data access over the network. Second, in-memory data requires synchronous replication in order to prevent the loss of data in case of failures. Wide area networks (e.g., between data-centers) have higher communication costs than LANs and are therefore unsuited for these requirements. Optimizing Tell for wide area is subject to future work. Network bandwidth is another constraint that can potentially become a bottleneck. The processing and the storage layer constantly exchange data, and factors such as heavy load or large record sizes can cause network saturation. Some of the techniques we present in this chapter introduce additional constraints that are detailed at a later point.
4.3 Related Work

Distributed databases that share data have been subject to various research over the years. Oracle RAC [CB03], IBM DB2 Data Sharing [JMNT97], and Oracle Rdb [LARS92] are based on a shared-data architecture. These systems use a global lock-manager to synchronize data access and ensure concurrency control. Data is stored on disk, and the granularity of sharing data is a page. In contrast, our approach is based on in-memory storage and reduces data access granularity to single records. More fundamentally, these systems are based on traditional tightly-coupled RDBMS engines.

An initial design of a shared-data architecture based on the principles presented in Section 4.1 has been introduced by Brantner et al. [BFG+08]. Moreover, the benefits of decoupling data storage from query processing and transaction management with regard to flexibility and elasticity have been highlighted in several publications [LFWZ09, LLMZ11]. A commercial database system that implements our design principles and that has been developed in parallel to Tell is FoundationDB [Fou15]. FoundationDB provides a “SQL Layer” that enables complex SQL queries on top of a transactional key-value store. FoundationDB has many similarities with Tell such as support for in-memory storage and optimistic MVCC protocol. However, critical implementation details (e.g., indexing, commit validation) have not been published yet. The purpose of Tell is to specifically provide these details. The experimental evaluation highlights that it is the right combination of techniques that is fundamental to achieve high performance. In the TPC-C benchmark Tell outperforms FoundationDB by a factor of 30 (Section 4.7.5).

Several additional shared-data databases have been published recently. ElasTras [DAEA13] is a more recent shared-data database designed for multi-tenancy. Data storage is decoupled from transaction management, but data partitions (or tenants) are exclusively assigned to PNs. Transactions can only be performed on single partitions. In our architecture, PNs can access all data and the scope of a transaction is not limited. Another shared-data database is Google F1 [SVS+13]. F1 is built on top of Spanner [CDE+12] which has evolved from Megastore [BBC+11]. F1 enables scalable transactions but is designed with regard to cross-datacenter replication. The reported latencies to perform a commit are 50-150 ms. We assume low-latency data access in LANs. A system that implements many of our design principles is OMID [GFJK+14]. OMID implements MVCC on top of HBase [Apa14b] to provide transaction support. However, unlike Tell, OMID requires a centralized component for conflict detection and commit validation. Hyder [BRD11] is
a transactional shared-data record manager. Records are stored in a log structure. Updates are appended to the log in total order and processing servers rollforward the log to reach a common state. Hence, transaction management is decoupled from storage. ScaleDB [Sca14] is another shared-data database that decouples query processing from data storage. However, no implementation details have been published so far.

Distributed partitioned databases are nowadays widespread. A thorough review of the architectural benefits is provided by DeWitt and Gray [DG92]. Partitioned databases provide scalability for “partition-friendly” applications but suffer from distributed transactions. H-Store [KKN+08] and its commercial successor VoltDB [Vol14] horizontally partition tables inside and across nodes. H-Store sequentially processes transactions on single partitions without the overhead of concurrency control. However, distributed transactions are blocking. Calvin [TDW+12] speeds up distributed transactions by reaching agreement before a transaction is executed. Jones et al. [JAM10] propose to speculatively execute single-partition transactions while waiting for distributed transactions to complete. They show that speculative execution works well for workloads with few cross-partition interactions and few aborts. Microsoft Azure SQL Database [BCD+11] has already been presented. It is a cloud service that enables to partition data across instances but does not support cross-partition queries. In a shared-data system, the problem of distributed transactions does not arise as processing nodes can access all data. In Tell, partitioning is transparent with regard to the PNs and all transactions are executed locally. Accordion [SMA+14] and E-Store [TMS+14] are two recent systems that propose data placement systems to enable online repartitioning [SI09] and provide elasticity for partitioned databases. Although these approaches considerably improve agility, adding or removing node implies re-partitioning and moving data. In Tell, elasticity is more fine-grained and PNs can be added without any cost.

Main memory databases [KN11, LBD+11, TZK+13] take advantage of memory performance and cache affinity to optimize processing in the scope of a single machine. A database designed for processing mixed workloads is HyPer [KN11]. HyPer creates copy-on-write snapshots of the data in virtual memory to execute OLAP workloads independently of on-going OLTP processing. A recent extension, called ScyPer [MRR+13], allows for offloading analytical processing onto secondary machines to improve OLAP performance. ScyPer multicasts a redo log and therefore executes OLAP queries on a consistent but slightly delayed state. The main difference to Tell is that OLTP queries are only processed on the master server.
4.4. Transaction Processing

Table 4.1: Comparison of selected distributed databases and storage systems

<table>
<thead>
<tr>
<th></th>
<th>Shared Data</th>
<th>Decoupling</th>
<th>In-Memory Storage</th>
<th>ACID Transactions</th>
<th>Complex Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tell (Shared-Data Architecture)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Oracle RAC</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FoundationDB</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Google F1</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Google Megastore</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>OMID</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Hyder</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>(✓)</td>
</tr>
<tr>
<td>VoltDB</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Azure SQL Database</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Google BigTable</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Sinfonia</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Recently, relational databases have been challenged by the emergence of NoSQL systems. NoSQL stores violate the principles of the shared-data architecture and typically relax consistency guarantees in favor of more scalability and availability [Cat11]. Amazon’s Dynamo [DHJ+07] was one of the first scalable key-value stores that guaranteed eventual consistency to achieve high availability. Other systems such as BigTable [CDG+08] or Spinnacker [RST11] provide strong consistency and atomic single-key operations but no ACID transactions. G-Store [DAEA10] does support ACID transactions for user-defined groups of data objects. The major restriction is that these groups must not overlap. A system that does not have this limitation is Sinfonia [AMS+09]. Sinfonia is a partitioned store that uses 2PC to reach agreement on transaction commit. None of the previous NoSQL stores supports a query language as powerful as SQL. However, many strongly consistent NoSQL systems are equivalent to atomic record stores and support basic operations as required by the storage layer in our architecture. Table 4.1 provides an overview of the discussed systems with regard to our design principles.

4.4 Transaction Processing

Figure 4.1 presents an overview of the components of Tell. The processing node is the central component that processes incoming queries and executes transactions. PNs interact with the commit manager, a lightweight service that manages global transaction state (Sec-
4.4.1 Distributed Snapshot Isolation

SI is a MVCC protocol that has been implemented in several database systems (Oracle RDBMS, PostgreSQL, SQL Server). MVCC stores multiple versions of every data item. Each time a transaction updates an item, a new version of that item is created. When a transaction starts, it retrieves a list with all data item versions it is allowed to access. Furthermore, a management node monitors the system and initiates a recovery process the moment a failure is detected. Finally, data is stored in a distributed storage system consisting of multiple SNs. The storage layer manages data records, indexes, as well as an undo log for recovery (Section 4.4.4). To enable transactional processing, the storage system must support consistent get/put operations on single records. Moreover, Tell depends on the availability of the storage system in order to process queries.

Concurrency control ensures that transactions can run in parallel on multiple PNs without violating data integrity. In this section, we describe a MVCC protocol. More specifically, we detail a distributed variant of snapshot isolation (SI) [BBG+95] that uses atomic RMW primitives for conflict detection. The SI protocol is part of the transaction management component on each PN in Tell and guarantees ACID semantics. SI is an optimistic protocol [KR81] in which transactions are never prevented from making progress. Evaluating our design with alternative concurrency control techniques is the topic of Chapter 5.

Figure 4.1: Tell architecture and components
4.4. Transaction Processing

In SI, a transaction is only allowed to access the versions that were written by already finished transactions (and never the ones written by transactions that are still running). This is the so-called consistent snapshot the transaction operates with. Transactions check for conflicts at commit time. Updates (insert, update, and delete operations) are buffered and applied to the shared store during commit. In the remainder of the thesis, we use the term “apply” to indicate that updates are installed in the shared store (i.e., applying an update makes it visible to all PNs).

The commit of transaction $T_1$ is only successful if none of the items in the write set have been changed externally by another transaction $T_2$ that is committing or has committed since $T_1$ has started. If $T_1$ and $T_2$ run in parallel and modify the same item, two scenarios can happen: First, $T_2$ writes (applies) the changed item to the shared store before it is read by $T_1$. In that case, $T_1$ will notice the conflict (as the item has a newer version). Second, $T_1$ reads the item before it has been written by $T_2$. If this happens, $T_1$ must be able to detect the conflict before it writes the item back. For this purpose, we perform atomic RMW operations in the storage layer. A particular type of atomic RMW is load-link and store-conditional (LL/SC) [JHB87]. LL/SC is a pair of instructions that reads a value (load-link) and allows to perform an update only if it has not changed in the meantime (store-conditional). LL/SC is stronger than the popular compare-and-swap operation as it solves the ABA-Problem [Mag04]. That is, a write operation on a modified item fails even when the value matches the initially read value. LL/SC is the key to conflict detection. If all updates of transaction $T_1$ can be applied successfully, there are no conflicts, and the transaction can commit. If one of the LL/SC operations fails, there is a write-write conflict, and $T_1$ will abort. That is, $T_1$ reverts all changes made to the data store.

SI avoids many of the common concurrency control anomalies. However, some anomalies (e.g., write skew) prevent SI to guarantee serializability in all cases [FLO+05]. Proposed solutions for serializable SI [JFRS07, CRF08, YGF12] are not designed for distributed systems or require centralized commit validation. Still, we mean to provide serializable SI in the near future.

In order to provide SI semantics across PNs, each node has to interact with a dedicated authority: the commit manager.
4.4.2 The Commit Manager

The commit manager service manages global snapshot information and enables new transactions to retrieve three elements: a system-wide unique transaction id (tid), a snapshot descriptor, and the lowest active version number (lav).

**Transaction ids** are monotonically incremented numeric values that uniquely identify a transaction. Every running transaction requires a *tid* before it can execute data operations. Given its system-wide uniqueness, the *tid* is not only used to identify transactions but also defines the version number for updated data items. In other words, a transaction creates new versions for updated data items with the *tid* as version number. Due to the fact that *tids* are strictly incremented, version numbers will increase accordingly. Hence, as *tids* and version numbers are synonyms, the set of *tids* of completed transactions also defines the snapshot (i.e., the set of accessible version numbers) for starting transactions.

**The snapshot descriptor** is a data structure that specifies which versions a transaction can access. It consists of two elements: First, a base version number *b* indicating that *b* and all earlier transactions have completed. Second, a set of newly committed *tids* *N*. As the name indicates, *N* contains all committed transactions with *tid* > *b* but not *b* + 1. When *b* + 1 commits, the base version is incremented until the next non-committed *tid* is reached. Obviously, *b* and the elements of *N* are strictly smaller than the transaction’s *tid*. The implementation of the snapshot descriptor is simple and involves low computational cost. *b* is an integer and *N* is a bitset. Starting with *b* at offset 0, each consecutive bit in *N* represents the next higher *tid* and if set to 1 indicates a committed transaction. As a result, the snapshot descriptor is small even in the presence of many parallel transactions. For example, *N* ≤ 13 KB with 100,000 newly committed transactions.

When accessing a data item with a set of version numbers (or version number set) *V*, the transaction reads the version with the highest version number *v* matching the snapshot descriptor. More formally, we define the valid version number set the transaction can access *V’* as:

\[
V' := \{ x \mid x \leq b \lor x \in N \}
\]

The version with number *v* is accessed:

\[
v := \max( V \cap V' )
\]
4.4. Transaction Processing

The lowest active version number is equal to the lowest base version number among all active transactions. That is, the highest version number globally visible to all transactions. Version numbers smaller than the lav are candidates for garbage collection (Section 4.5.4).

In order to communicate with the processing nodes, the commit manager provides a simple interface. The following three functions are supported:

- **start() → (tid, snapshot descriptor, lav):** Signals the start of a new transaction. Returns a new tid, a snapshot of the current state, and the lav.

- **setCommitted(tid) → void:** Signals that transaction tid has successfully committed. Updates the list of committed transactions used to generate snapshot descriptors.

- **setAborted(tid) → void:** Same as above but for aborts.

To scale and prevent a single point of failure, several commit managers can run in parallel. Commit manager nodes use the shared store to synchronize on tids and on the snapshot. The uniqueness of every tid is ensured by using an atomically incremented counter value in the storage system. The counter is a simple integer managed in the shared store. PNs update the counter value using atomic RMW operations to ensure that tids are never assigned twice. To prevent counter synchronization from becoming a bottleneck, PNs can increment the counter by a high value (e.g., 256) to acquire a range of tids that can be assigned to transactions on-demand. We opted for continuous ranges of tids because it is simple to implement. However, the approach has limitations (e.g., higher abort rate). Using ranges of interleaved tids [TZK+13] is subject to be implemented in the near future.

To synchronize the current snapshot (i.e., the list of committed transactions), we use the storage system as well. In short intervals, every commit manager writes its snapshot to the store, and thereafter it reads the latest snapshots of the other commit managers. As a result, every commit manager gets a globally consistent view that is at most delayed by the synchronization interval. Operating on delayed snapshots is legitimate and does not affect correctness. Nevertheless, the older the snapshot, the higher the probability of conflicts. In practice, a synchronization interval of 1 ms did not noticeably affect the overall abort rate (Section 4.7).
4.4.3 Life-Cycle of a Transaction

This section describes the life-cycle of a transaction and enumerates the different transaction states:

1. **Begin:** In an initial step a transaction contacts the commit manager to retrieve its snapshot.

2. **Running:** While running, a transaction can access or update any data item. Read operations retrieve the item from the store, extract the valid version, and cache it in a buffer only visible to the transaction in case the item is re-accessed. Updates are buffered until commit. Data storage and buffering are detailed in Section 4.5.

3. **Try-Commit:** Before starting the commit procedure, a log entry with all the ids of updated items is written to the transaction log. This is a requirement for recovery and fail-over (Section 4.4.4). We proceed with applying updates to the store using LL/SC operations. On success, the transaction commits. Otherwise, we abort.

4. (a) **Commit:** All data updates have been applied. Next, the indexes are altered to reflect the updates, and a commit flag is set in the transaction log entry. Finally, the commit manager is notified.

   (b) **Abort:** On conflict, applied data updates are rolled back. A transaction can also be aborted manually. In this case, no updates have been applied (as we skipped the **Try-Commit** state). Last, the commit manager is notified.

4.4.4 Recovery and Fail-Over

This section illustrates how node failures are addressed. Failures are detected by the management node using an eventually perfect failure detector based on timeouts. Although we describe fail-over for single node failures, handling multiple failures concurrently is supported.

4.4.4.1 Failure of Processing Node

PNs act according to the crash-stop model. That is, in case of transient or permanent failures, all active transactions on a failed node are aborted. As soon as a failure is detected,
4.4. Transaction Processing

A recovery process is started to roll back the active transactions and revert uncommitted changes. Correct recovery is enabled by a transaction log that contains the status of running transactions.

The transaction log is an ordered map of log entries located in the storage system. Before applying any updates, a transaction must append a new entry to the log. Every entry is identified by the \textit{tid} and consists of the PN id, a timestamp, the \textit{write set}, and a flag to mark the transaction committed. The \textit{write set} is the list of ids of updated records.

When the recovery process is started, it first discovers the active transactions of the failed node. This involves retrieving the highest \textit{tid} from the commit manager and iterating backwards over the transaction log until the lowest active version number is reached. The \textit{lav} implicitly acts as a rolling checkpoint. Once we encounter a relevant transaction, we use the \textit{write set} and revert the changes made by the transaction. That is, the version with number \textit{tid} is removed from the records. On completion of the recovery process, all transactions of the failed node have been rolled back. The management node ensures that only one recovery process is running at a time. However, a single recovery process can handle multiple node failures.

4.4.4.2 Failure of a Storage Node

The storage system must handle node failures transparently with regard to the processing nodes. In order to remain operational, PNs require data access and assume a highly available storage system. Obviously, a failure must not lead to data loss. Moreover, downtime must be minimized as it will cause transaction processing to be delayed. Availability is guaranteed by replicating data. As data is kept in volatile storage (main memory) a SN ensures that data is replicated before acknowledging a request. This implies synchronous replication regardless of the replication protocol used (ROWA, majority quorum, etc.). If a node fails, the storage system fails-over to the replicas and enables on-going processing of requests. Eventually, the system re-organizes itself and restores the replication level. Ensuring high availability in simple record stores is a well studied field [DHJ07, CDG08].

The storage layer in Tell uses a management node to detect failures. The same node manages partitioning, restores the replication factor, and enables PNs to look-up the location of replicas. To prevent a single point of failure, several management nodes with a synchronized state are required. Synchronizing the state of the management nodes is
not a performance critical operation and can be implemented using common consensus protocols [Lam98, HKJR10].

4.4.4.3 Failure of a Commit Manager

In a single commit manager configuration, the failure of the commit manager node has system-wide impact. PNs are no longer able to start new transactions, and the system is blocked until a new commit manager is available. Once all active transactions at the moment of failure have committed (the commit manager is not required for completion), a new commit manager can be started. To restore its state, the commit manager retrieves the last used $tid$ and the most recently committed transactions from the transaction log.

To prevent a single point of failure and increase availability, multiple commit managers can operate in a cluster. As pointed out previously, commit managers synchronize their state. If a commit manager becomes unavailable, PNs automatically switch to the next one. New commit managers can determine the current state from the data of the other managers. Recall that the commit managers regularly write their state to the storage system. As a result, state information is accessible independently of the availability of the commit managers.

4.4.5 Discussion of Alternatives

The described variant of SI in this section performs version management and garbage collection in the processing layer. This comes at the price of additional network traffic as all versions of a record have to be retrieved to select the right one. Alternatively, version management can be transferred into the storage layer. As a result, every single version of every record would be stored separately. To access a record, a transaction would have to specify its snapshot, and the storage system could select and return the correct version by itself. Managing versions in the storage system would reduce the amount of network traffic as less data is transferred. However, it also adds the burden of version management and garbage collection to the storage layer. Details regarding the implementation of garbage collection are provided in Section 4.5.4. For the evaluation in Section 4.7, SI and garbage collection are performed in the processing layer. Analyzing the alternative approach is subject to future work.
4.5 Data Access and Storage

Tell provides a SQL interface and enables complex queries on relational data. The query processor is a component of the processing node. It parses and optimizes incoming queries and uses the iterator model to access records. Records are retrieved and modified using basic data operations (e.g., read, write, or atomic write for LL/SC semantics) and are stored in a key-value format. In this section, we explain how relational data is mapped to the key-value model and detail data access methods. A major advantage of the key-value format is simplicity. A simple data model is the foundation to implement alternative storage abstractions. For example, an HDFS interface on top of key-value data would enable to perform MapReduce jobs [DG04] while the relational interface allows for executing short-running OLTP transactions simultaneously. The study of alternative storage abstractions is subject to future work.

4.5.1 Data Mapping

The mapping of relational data to the key-value model is unambiguous: Every relational record (or row) is stored as one key-value pair. The key is a unique record identifier (rid).
Rids are monotonically incremented numerical values. The value field contains a serialized set of all the versions of the record. This row-level storage scheme is a significant design decision as it minimizes the number of storage accesses. For instance, with a single read operation, a transaction retrieves all versions of a record and can select the one that is valid according to the snapshot. On update, a transaction adds a new version to the record and writes back the entire record during commit. Again, a single atomic write request applies the update or identifies a conflict (if the update is rejected).

Grouping records into pages, as disk-oriented databases do in order to reduce the number of I/O operations [GR92], is of limited use in a shared-data architecture. A record needs to be re-fetched on every access because it can be changed anytime by remote PNs. Consequently, a coarse-grained storage scheme would not reduce the number of requests to the storage system but only increase the amount of network traffic. In contrast, a more fine-grained storage scheme (e.g., store every version as a key-value pair) would require additional requests to identify added versions. Committing transactions would have to check for new versions before applying updates and consequently conflict detection would become more expensive. Although storing single versions reduces network traffic, we opt for an approach that minimizes the number of network requests. Recall that minimizing the number of network requests is an optimization goal of Tell (Section 4.2).

To read a record, PNs rely on index structures. Indexes contain references to the rid and enable a PN to retrieve records with all versions (Figure 4.2). To update a record, it is first accessed by the transaction. Next, a new version of the tuple reflecting the changes of the update is added to the set of versions. The entire record is kept in the transaction buffer, and further updates to the record directly modify the newly added version. Finally, once the transaction is about to commit, the record is written back to the storage system. As previously explained, the update is executed using an atomic operation that will only succeed if the record has not been changed since it was initially read.

In order to minimize the number of network requests to the storage system and improve performance, Tell aggressively batches operations (i.e., several operations are combined into a single request). Tell uses two orthogonal batching strategies: explicit and implicit. Explicit batching is performed when a transaction needs to access several records. For instance, a range-read is performed by sending a single multi-read request to the storage system. The storage system processes every operation that is part of the batch individually and combines the results before sending the reply. Explicit batching is also used to apply the updates of a transaction during commit. Implicit batching attempts to join operations
from different transactions. That is, in the presence of several consecutive read operations waiting in the I/O queue, the storage interface on the PN combines the operations into a batch request.

### 4.5.2 Mixed Workloads

OLTP workloads typically consist of short-running transactions that access few records. Every transaction only requires a limited amount of remote accesses and utilizes low bandwidth. As a result, OLTP execution can be highly parallelized and scales with the number of PNs (Section 4.7). OLAP workloads, on the other hand, perform long running queries that access large amounts of data. For instance, an analytical query might execute an aggregation that involves a full table scan (i.e., access all the records of a table). To process this query, a PN must read (and transfer over the network) all the records of the table. Obviously, processing this type of query has limitations with regard to latency and bandwidth utilization.

To enable the efficient execution of mixed workloads, we propose to push down selected relational operators into the storage layer. For instance, executing simple operations such as selection or projection in the SN would enable to reduce the size of the result set and lower the amount of data sent over the network. A more advanced solution is to execute a shared scan over the entire data in the storage layer as suggested in [UGA+09, Ron12]. A dedicated thread could scan all stored records and pre-filter data for complex OLAP queries [RSQ+08]. The challenge is to perform the scan efficiently without affecting ongoing point (single get/put) operations. Mixed workloads are beyond the scope of this dissertation. Nevertheless, the concept of performing basic processing tasks in the storage layer is a promising direction for future work.

### 4.5.3 Latch-Free Index Structures

Indexes provide an efficient lookup mechanism to identify matching records for a given attribute (or a set of attributes). Indexes can be accessed by multiple transactions in parallel and therefore require concurrency control. In a shared-data architecture, integrity must be maintained across nodes as an index can be modified by multiple PNs. Traditionally, index integrity is enforced using latches. However, latch-free algorithms have become popular recently [Hor13]. In a distributed setting, latch-freeness is even more a
Chapter 4. Tell: A Distributed Shared-Data Database

desirable property as network communication increases latch acquisition cost. Moreover, the absence of latches ensures system-wide progress. Accordingly, Tell provides a latch-free B+tree.

4.5.3.1 B+Tree Index

The B+tree is one of the most used index structures in relational databases. Over the last decades, many variations have been optimized for concurrent access [LY81, Gra10]. Particularly noteworthy is the Bw-tree [LSL13], a latch-free variant for modern hardware. Taking the techniques of the Bw-tree as a reference, we have implemented a latch-free B+tree concurrently accessible by multiple PNs.

The B+tree index is entirely stored and maintained in the storage layer (every index node is stored as a key-value pair) and can be accessed by all processing nodes. Synchronization of index updates is based on LL/SC operations. That is, every node in the tree is atomically modified in the storage system. On lookup, a transaction accesses the root node and traverses the tree until the leaf-level is reached. B+tree nodes are cached in the processing layer to improve traversal speed and minimize storage system requests. Caching works as follows: All index nodes with exception of the leaf level are cached. The leaf level always retrieves the latest version from the storage system. If the range of the leaf node (lowest and largest value) does not match the data in its parent (i.e., the leaf node must have been split or merged), then the parent nodes are recursively updated (i.e., the latest version is retrieved from the storage system) to keep the cache consistent. The caching mechanism is independent of the synchronization of index changes.

4.5.3.2 Design Considerations

In order to prevent unnecessary network requests and increase scalability, versioning information is excluded from all indexes. Instead of maintaining a key-reference entry for every existing version, only a single index entry per record is maintained. As a result, it is not necessary to insert new index entries on every record update (i.e., whenever a new version is added) but only when the indexed key is modified. Less index insertions significantly improve index concurrency.

Version unaware indexes have limitations. In particular, it is no longer possible to identify for which versions an index entry is valid. In the absence of that information, a transaction
might retrieve a record that is not valid according to its snapshot descriptor. Obviously, transactions will ignore false positive results. Although these reads are unnecessary, they are performed independently of the index and therefore do not affect index concurrency. In addition, index-only scans are not possible as the validity of an entry cannot be verified.

4.5.4 Garbage Collection

Every update adds a new version and over time records become increasingly larger. To prevent data items from growing infinitely, garbage collection (GC) is necessary. Garbage collection deletes the versions and index entries that are never going to be accessed again. We use two garbage collection strategies: The first one is *eager* and cleans up data records and indexes as part of an update or read operation respectively. The second strategy is *lazy* and performs GC in a background task that runs in regular intervals (e.g., every hour). The latter approach is useful for rarely accessed records.

Record GC is part of the record update process. Before a transaction applies an updated record, it first verifies if some older versions can be removed. To determine evictable versions, a transaction uses the lowest active version number it has obtained from the commit manager on transaction start. Recall that the $lav$ is the highest version number globally visible to all transactions. Consequently, with the exception of the latest version of the record, all versions with a number smaller than the $lav$ will never be accessed again and can safely be garbage collected. Given a record with version numbers $V$ and the $lav$, we define $C$ as the version number set of a record visible to all transactions:

\[
C := \{ x \mid x \in V \land x \leq lav \}
\]

The set of garbage collectible version numbers $G$ is:

\[
G := \{ x \mid x \in C \land x \neq \text{max}(C) \}
\]

The version $\text{max}(C)$ is not garbage collected to guarantee that at least one version of the item always remains. All versions whose number is in $G$ can safely be deleted before the record is written back to the storage system. As a positive side effect, this approach to GC does not require extra requests to the storage layer.

Index entries are also subject to garbage collection. Index GC is performed during read operations. When a transaction performs an index lookup and afterwards reads all matching records, it verifies if some of the entries can be removed. Given an index entry $k$ with
key \( a \), we define \( V_a \) as the version number set of all versions that contain \( a \) with \( V_a \subseteq V \).

\( k \) can be removed from the index if the following condition holds:

\[
V_a \setminus G = \emptyset
\]

In other words, the index entry is removed if \( a \) is only referenced by versions whose numbers are present in the garbage collectible version number set \( G \). Index nodes are consistently updated. If the atomic RMW operation fails, GC is retried with the next read.

### 4.6 Buffering Strategies

In this section, we present three approaches to improve buffering in a shared-data architecture. Buffering data records in main memory lowers data access latencies and improves processing speed. As mentioned earlier, the limited ability to buffer data is a notable constraint of the shared-data architecture. The effectiveness of the three strategies is illustrated in Section 4.7.

#### 4.6.1 Transaction Buffer

Transactions operate on a particular snapshot and do not necessarily require the latest version of a record. Accordingly, data records, once read by a transaction, are buffered and re-used in the scope of the transaction. Every transaction maintains a private buffer that caches all accessed records for the duration of the transaction’s lifetime. This caching mechanism has obvious limitations: There is no shared buffer across transactions and buffering is only beneficial if transactions access the same record several times.

#### 4.6.2 Shared Record Buffer

The second approach is based on a shared record buffer used by all transactions on a processing node. It acts as a caching layer between the transaction buffers and the storage system. Although records can be modified by remote PNs and new transactions need to access the storage system to retrieve the most recent record version, transactions running in parallel can benefit from a shared buffer. For instance, if a transaction retrieves a record, the same record can be “re-used” by a transaction that has started before the first one (i.e., a transaction with an older snapshot).
This strategy is implemented using *version number sets* (as defined in Section 4.4.2). In the buffer, we associate each record with a version number set that specifies for which version numbers the record is valid. Comparing the version number set of the transaction’s snapshot descriptor with the version number set of the buffered record allows to determine if the buffer entry can be used or if the transaction is too recent. If a transaction with version number set \( V_{tx} \) wants to read a buffered record with version number set \( B \), the following conditions are used to verify the validity of the entry:

1. \( V_{tx} \subseteq B \): The buffer is recent enough. We can access the record from the buffer and no interaction with the storage system is necessary.

2. \( V_{tx} \not\subseteq B \): The cache might be outdated. We first get \( V_{max} \), the version number set of the most recently started transaction on the processing node. Then, we retrieve the record from the storage system and replace the buffer entry. Finally, \( B \) is set to \( V_{max} \). As all transactions in \( V_{max} \) committed before the record was read, it is certain to be a valid version number set.

The rational behind \( V_{max} \) is to keep \( B \) as big as possible to improve the likelihood that condition 1 holds for future data accesses. Once a record is retrieved from the buffer, we check if the versions contained in the record are valid according to \( V_{tx} \). This step prevents that a transaction reads a *phantom* (i.e., a record inserted into the cache by an active transaction).

To insert a record into the buffer, we apply the procedure of condition 2. If the buffer is full a replacement strategy (e.g., Least-Recently-Used) is used to evict entries. Record updates are applied to the buffer in a *write-through* manner. Each time a transaction applies an update, the changes are written to the storage system and if successful to the buffer as well. \( B \) is set to the union of \( tid \) and \( V_{max} \).

\[
B := \{tid\} \cup V_{max}
\]

\( V_{max} \) is a valid version number set for updated records because if a transaction in \( V_{max} \) would have changed the record, the LL/SC operation to the storage system would have failed. Updates to the buffered record and the version number set are executed as atomic operations to ensure consistent buffer changes.

In the presence of multiple concurrent transactions accessing the same items on a processing node, global buffering reduces the number of read requests to the storage system.
and thus lowers data access latencies. This comes at the price of additional management overhead and increased memory consumption on the PNs.

4.6.3 Shared Buffer with Version Set Synchronization

This variant is an extension of the previously described shared buffer. The key idea of this approach is to use the storage system to synchronize on version number sets of records. The advantage is that a PN can verify if a buffered record is valid by retrieving its version number set. If the buffer is not valid, the record is re-fetched. This strategy saves network bandwidth as version number sets are small compared to records.

Record updates are handled the same way as for the shared record buffer except that a transaction not only writes the data changes to the storage system but also updates the version number set entry. The mechanism for data access is slightly different. A buffered record is accessed according to the following conditions:

1. $V_{tx} \subseteq B$: The buffer entry is valid.
2. $V_{tx} \not\subseteq B$: The cache might be outdated. We fetch the record’s version number set $B'$ from the storage system.
   
   (a) If $B' = B$ the buffered record is still valid.
   
   (b) If $B' \neq B$ the record must be re-fetched. Moreover, $B$ is replaced by $B'$.

Although this strategy reduces network traffic, it comes at the expense of additional update overhead. For each record update, two requests are performed in the storage layer. Both requests can not be batched as we have to ensure that the record was applied successfully before modifying the version number set.

An optimization to reduce the number of additional storage system requests is record grouping. Instead of maintaining one version number set per record, multiple records are grouped in a cache unit and share a common version number set. By default, multiple sequential records of a relational table are assigned to a cache unit. The mechanism for record access and update remains the same except that once the version number set is updated, all buffered records of a cache unit are invalidated. The advantage of this strategy is that much less version number sets have to be written and read from the storage system.
4.7. Experimental Evaluation

For instance, multiple updates to the same cache unit only require the version number set to be updated once. On the other hand, records are more frequently invalidated.

This strategy benefits from a workload with a high read ratio as cache units will be invalidated less often. The higher the update ratio, the more buffered records are invalidated and eventually the update overhead outweighs the number of saved requests.

4.7 Experimental Evaluation

This section provides an experimental evaluation of Tell. We first describe environment and methodology before presenting experimental results. Our experiments are based on the TPC-C benchmark that is popular both in industry and academia.

4.7.1 Implementation and Environment

The PN in Tell is implemented in about 15,600 lines of C++ code. Transaction management and data access are performed with the techniques described in Sections 4.4 and 4.5 respectively. PNs have a synchronous processing model. That is, a thread processes a transaction at a time. While waiting for a network request to complete, another thread takes over. The PN provides an interface that enables to process SQL statements. However, our prototype does not include a query parser and an optimizer yet, and therefore every query we execute is implemented with a custom query plan.

PNs interact with the RamCloud storage system (RC) [OAE+10]. RC is a strongly consistent in-memory key-value store designed to operate in low-latency networks. Data is accessed using a client library that supports atomic get and put operations (i.e., LL/SC) and allows to structure key-value pairs in tables. RC manages main memory using an approach similar to that of log-structured filesystems [RKO14]. This allows RC to use memory at 80-90% utilization without major performance degradation. Tables are hash-partitioned in order to distribute load across SNs. To guarantee fault-tolerance, RC supports remote backup with fast recovery [ORS+11]. The backup mechanism synchronously replicates every put operation to the replicas and thereafter asynchronously writes it to persistent storage. The replication factor (RF) specifies the number of data copies. RC uses replication for fault-tolerance only. All requests (get and put) to a particular partition are sent to the master copy.
Workload Mix | Throughput | Transaction Mix |
--- | --- | ---
| Write Ratio | Metric | New Order | Payment | Delivery | Order Status | Stock Level |
Write-Intensive (Std.) | 35.84% | TpmC | 45% | 43% | 4% | 4% |
Read-Intensive | 4.89% | Tps | 9% | 0% | 0% | 84% | 7% |

Table 4.2: Read and write-intensive workloads for the TPC-C benchmark

The benchmarking infrastructure consists of 12 servers. Each machine is equipped with two quad core Intel Xeon E5-2609 2.4 GHz processors, 128 GB DDR3-RAM and a 256 GB Samsung 840 Pro SSD. A server has two NUMA units that consist each of one processor and 50% of total memory. A process (or node) is by default assigned to one NUMA unit. This assignment allows to run two processes per server (usually a PN and a SN) and thus doubles the maximum number of logical nodes (24). We did not notice a performance difference compared to single process operation. The servers are connected to a 40 Gbit QDR InfiniBand network. All nodes are connected to the same switch.

4.7.2 Benchmarks

The TPC-C is an OLTP database benchmark that models the activity of a wholesale supplier. The benchmark consists of five transactions that include entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses.

Load is generated by terminal clients that emulate users entering data on an input screen. Every terminal operation results in one database transaction. Terminals are run separately from the system under test. Our TPC-C implementation differs from the official specification (Rev. 5.11) [Tra10]. Specifically, we have removed wait times so that terminals continuously send new requests to the PNs. The number of terminal threads is selected so that the peak throughput of the SUT is reached.

The size of the database is determined by the number of warehouses (WH). Each warehouse is populated with 30,000 customers and 30,000 orders with an average of 10 order-lines each. The default population for every TPC-C run is 200 WHs. In our implementation, a warehouse occupies approximately 189 MB of RamCloud memory space.

The TPC-C standard transaction mix is update-intensive with an overall write ratio of 35.84% and 92% of all transactions performing at least one update query. The primary
throughput metric is the new-order transaction rate (TpmC). That is, the number of successfully executed new-order transactions per minute. Not included are aborted transactions or transactions that exceed the TPC-C response time threshold. The TpmC represents around 45% of all issued transactions. The TPC-C standard mix inserts large amounts of data (i.e., at high throughput up to 30 GB/min) and restricted us to limit the measurement interval to 12 minutes for every TPC-C run. The benchmark was executed 5 times for each configuration and the presented results are averaged over all runs. The measured performance was predictable and the variations were very low.

In order to evaluate read-intensive scenarios, we propose an additional TPC-C mix that is read-mostly. This mix only consists of three transactions and provides a read-ratio of 95.11%. As we change the percentage of new-order transactions, throughput is measured in number of transactions per second (Tps). Table 4.2 gives an overview of both workload mixes. We additionally measure latency and transaction abort rate as these metrics provide valuable information.

4.7.3 Scale-Out

This section studies scale-out behavior and shows that the shared-data architecture can scale with an increasing number of nodes. To that end, we conducted experiments in which the number of PNs, SNs, and commit managers is varied individually.

4.7.3.1 Processing Nodes

Figure 4.3 presents results for the standard TPC-C mix. For this experiment, we increased the number of PNs. All other components had enough resources and did not limit performance (7 SNs, 1 commit manager).

With no replication (RF1), throughput increased with the number of processing resources from 143,114 TpmC with 1 PN to 958,187 TpmC with 8 PNs. The TPC-C benchmark suffers from data contention on the warehouse table, and therefore throughput did not increase linearly. Contention is reflected in the overall transaction abort rate (for all TPC-C transactions) that grew from 2.91% (1 PN) to 14.72% (8 PNs).

Increasing the replication factor adds overhead as update operations are synchronously replicated. With RF3, we reached a peak throughput of 350,257 TpmC with 8 PNs, thus 63.2% less than with RF1. Synchronous replication increases latency and consequently
affects the number of transactions a worker thread can process. In the 8 PN configuration, the average transaction response time increased from 10.69 ms with RF1 to 29.67 ms with RF3. Increasing the number of processing threads to compensate for higher access latencies had no effect because of excessive context switching on the PNs. The transaction abort rate with RF3 is similar to the one with RF1 as the probability of conflict is alike (higher processing time compensates for lower throughput).

Figure 4.4 shows the results of the same experiment but running the read-intensive TPC-C mix. As read operations only retrieve records from the master copy (and do not require interaction with the replicas), their latency is not affected by an increase in the replication level. Consequently, the higher the read-ratio of a workload, the less the performance impact of replication. Figure 4.4 illustrates this observation. With RF3 and 8 PNs, the throughput is 25.7% lower than with RF1, a considerable improvement compared to the write-heavy scenario.

Figures 4.3 and 4.4 emphasize that Tell enables scalable transaction processing with an increasing number of processing resources. The performance loss due to synchronous replication under write-intensive workloads is a common effect related to replication as a technique that is not specific to the architecture or the specific implementation in RC.
4.7. Experimental Evaluation

![Graph showing scale out processing](image)

**Figure 4.4:** Scale out processing (read-intensive, vary PN, 7SN, vary RF, 200WH)

### 4.7.3.2 Storage Nodes

Figure 4.5 shows the scale-out of the storage layer. We executed the TPC-C standard mix with three storage configurations (3, 5, and 7 SNs) and measured the overall system throughput with increasing load. In all configurations the storage layer was not a bottleneck, and therefore the throughput difference is minimal. The configuration with 3 SNs could not run with more than 5 PNs because of insufficient main memory capacity. With 6 PNs and RF3, the benchmark generates too much data to fit into the combined memory capacity of the 3 SNs. As a consequence, the number of SNs in a cluster should be determined by the required memory capacity and not by the available CPU power.

### 4.7.3.3 Commit Managers

Table 4.3 shows the performance impact of running several commit managers. Commit managers use a simple protocol to assign unique *tids* and keep track of completed transactions (Section 4.4). In particular, they are not required for complex tasks such as commit validation. The state among commit managers is synchronized using the approach described in Section 4.4.2. Table 4.3 shows that running the TPC-C benchmark
Figure 4.5: Scale out storage (write-intensive, vary PN, vary SN, RF3, 200WH)

<table>
<thead>
<tr>
<th>Commit Managers</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>TpmC</td>
<td>958'187</td>
<td>955'759</td>
<td>955'608</td>
</tr>
<tr>
<td>Tx abort rate (%)</td>
<td>14.75</td>
<td>15.59</td>
<td>15.91</td>
</tr>
</tbody>
</table>

Table 4.3: Scale out commit managers (write-intensive, 8PN, 7SN, RF1, 200WH)

with a synchronization delay of 1ms caused no significant impact on throughput and on the overall transaction abort rate. Consequently, the commit manager component is not a bottleneck.

4.7.4 Comparison to Partitioned Databases

This section evaluates Tell against two state-of-the-art partitioned databases: VoltDB and MySQL Cluster.

VoltDB [SW13] is an in-memory relational database that partitions data and serially executes transactions on each partition. The more transactions can be processed by single partitions, the better VoltDB scales. Consequently, we use single-partition trans-
actions whenever possible. Transactions are implemented as stored procedures that are pre-compiled to optimize execution. VoltDB supports replication (called \textit{K-factor}) and enables to asynchronously write a transaction log to the SSDs (VoltDB \textit{command log}). As a result, the system provides similar durability guarantees as RamCloud. VoltDB Enterprise 4.8 is run in configurations of 1, 3, 5, 7, 9, and 11 nodes (8 cores each) with 6 partitions per node (as advised in the official documentation). The nodes communicate with TCP/IP over InfiniBand. Our implementation follows the performance related recommendations described in [Vol14]. In particular, the terminals invoke transactions asynchronously and use multiple node connections.

**MySQL Cluster** [MyS14] is another partitioned database with an in-memory storage engine (i.e., NDB). A cluster configuration consists of three components: \textit{Management nodes} (MN) that monitor the cluster, \textit{Data nodes} (DN) that store data in-memory and process queries, and \textit{SQL nodes} that provide an interface to applications and act as federators towards the DNs. MySQL Cluster synchronously replicates data. Our benchmark environment runs MySQL Cluster 7.3.2 and uses the InfiniBand network as well. The benchmark is executed with 3, 6, and 9 DNs. We vary the number of SQL nodes and use two MNs. Terminals use prepared statements to execute transactions against the SUT.

With regard to partitioned databases, the data model and the workload require careful consideration. The TPC-C data model is ideal for Sharding. Most tables reference the \textit{warehouse id} that is the obvious partitioning key. The read-only item table can be fully replicated across all partitions. The TPC-C workload is more problematic. According to the TPC-C specification [Tra10], remote new-order (clause 2.4.1.8) and remote payment (clause 2.5.1.6) transactions access data from several warehouses and therefore require cross-partition transactions. In the TPC-C standard mix, the ratio of cross-partition transactions is about 11.25%.

Figure 4.6 compares VoltDB and MySQL Cluster to Tell. The figure presents the peak TpmC values for the TPC-C standard mix with RF3 on a varying number of CPU cores (i.e., the sum of all cores available to the database cluster). Each data point corresponds to the configuration with the highest throughput for the given number of cores. For instance, a minimal VoltDB cluster consists of 3 nodes (24 cores). The minimal Tell configuration with 22 cores consists of 1 PN (4 cores), 3 SNs (12 cores), 2 commit managers (4 cores), and 1 MN (2 cores). In comparison to Tell, VoltDB and MySQL Cluster do not scale with
Chapter 4. Tell: A Distributed Shared-Data Database

![Graph showing comparison of TPC-C transaction response time for different databases](image)

**Figure 4.6:** Partitioned databases (TPC-C standard, vary #cores, RF3, 200WH)

<table>
<thead>
<tr>
<th>Database</th>
<th>Small 22-24cores (mean ± σ, ms)</th>
<th>Large 70-72cores (mean ± σ, ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tell</td>
<td>14 ± 17</td>
<td>32 ± 41</td>
</tr>
<tr>
<td>MySQL Cluster</td>
<td>34 ± 26</td>
<td>43 ± 40</td>
</tr>
<tr>
<td>VoltDB</td>
<td>706 ± 2159</td>
<td>655 ± 1875</td>
</tr>
<tr>
<td>FoundationDB</td>
<td>149 ± 183</td>
<td>120 ± 138</td>
</tr>
<tr>
<td>Tell</td>
<td>14 ± 17</td>
<td>32 ± 41</td>
</tr>
<tr>
<td>VoltDB</td>
<td>62 ± 102</td>
<td>22 ± 59</td>
</tr>
</tbody>
</table>

**Table 4.4:** TPC-C transaction response time for small and large configuration

the number of cores. While Tell reaches a throughput of 374,894 TpmC with 78 cores, MySQL Cluster and VoltDB achieve 64,185 and 23,183 TpmC respectively. The scalability of the partitioned databases is presumably limited by distributed transactions that require coordination across nodes. In particular, VoltDB suffers from cross-partition transactions as throughput decreases the more nodes are added. This observation is emphasized by VoltDB’s high latency (Table 4.4). MySQL Cluster is slightly faster than VoltDB because
4.7. Experimental Evaluation

Single-partition transactions are not blocked by distributed transactions. The experiment highlights that the techniques presented in this work efficiently interact. Moreover, the shared-data architecture enables much higher performance than partitioned databases for OLTP workloads that are not fully shardable.

In a second experiment, we evaluate Tell given a workload that is optimized for partitioned databases. To that end, we propose a modification of the TPC-C benchmark that removes all cross-partition transactions. This alternative workload, called TPC-C shardable, replaces remote new-order and payment transactions with equivalent transactions that only access a single warehouse.

Figure 4.7 compares Tell to VoltDB using the TPC-C shardable with RF1 and RF3. Again, we vary the number of cores. With this workload, VoltDB fulfills the scalability promises and achieves more throughput than Tell. The peak performance of VoltDB with RF1 is 1.387 Mio TpmC. Tell achieves 1.225 Mio TpmC, thus 11.7% less than VoltDB. In addition, a shardable workload results in much better latency for VoltDB (Table 4.4). The numbers emphasize that the shared-data architecture enables competitive OLTP performance. Even with a perfectly shardable workload, the achieved throughput is in the same ballpark as a state-of-the-art partitioned database.
4.7.5 Comparison to Shared-Data Databases

Section 4.3 introduced FoundationDB [Fou15], a database system built on the same design principles as Tell. FoundationDB provides a “SQL Layer” that allows for performing SQL transactions on top of a transactional key-value store. Data can be either stored in-memory or on SSDs. Our environment runs FoundationDB 3.0.6 with in-memory data storage and RF3 (redundancy mode triple). The benchmark is executed on three configurations with 3, 6, and 9 node in both layers. The smallest configuration (24 cores) achieves 2,706 TpmC and the largest (72 cores) reaches 10,047 TpmC. Although FoundationDB scales with the number of cores, the throughput is more than a factor 30 lower than Tell (the results are even worse than for VoltDB and MySQL Cluster and therefore not included in Figure 4.6). Without sufficient implementation details available, it is difficult to explain this huge difference. Nevertheless, the results highlight that it is not only the architecture itself but more fundamentally the specific combination of techniques that determines the performance of a shared-data database.

4.7.6 Network

Throughout the chapter, we have emphasized that fast data access is a major requirement to achieve scalability. In this experiment, we highlight the importance of low-latency data access in a shared-data architecture. Figure 4.8 compares the throughput using an InfiniBand network to the performance of 10 Gbit Ethernet. In the experiment, we varied the number of PNs. The RF is 1 and the number of SNs is constant. The TpmC results on InfiniBand are more than six times higher than the results achieved with Ethernet independent of the number of PNs. This difference is a direct effect of network latencies. InfiniBand uses RDMA and by-passes the networking stack of the operating system to provide much lower latencies. Table 4.5 summarizes the results for the fastest configuration with 8 PNs. The second column shows the mean transaction response time and the standard deviation. Both values reflect the measured difference in throughput. The last two columns show the 99th and 99.9th percentile response time. The low number of outliers indicates that both networking technologies are not congested. In the InfiniBand configuration, the total bandwidth usage of one SN is 125.9 MB/s (in and out). Thus, the network is not saturated.
4.7. Experimental Evaluation

![Graph showing network comparison](image)

**Figure 4.8:** *Network comparison (write-intensive, vary PN, 7SN, RF1, 200WH)*

<table>
<thead>
<tr>
<th></th>
<th>TpmC</th>
<th>Latency (mean $\pm \sigma$, ms)</th>
<th>TP99 (ms)</th>
<th>TP999 (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>40 Gbit InfiniBand</td>
<td>958,187</td>
<td>10.69 $\pm$ 13.02</td>
<td>76.48</td>
<td>100.62</td>
</tr>
<tr>
<td>10 Gbit Ethernet</td>
<td>151,632</td>
<td>69.41 $\pm$ 87.99</td>
<td>542.03</td>
<td>644.15</td>
</tr>
</tbody>
</table>

**Table 4.5:** *Network latency (write-intensive, 8PN, 7SN, RF1, 200WH)*

4.7.7 Buffering Strategies

Figure 4.9 shows a comparison of the buffering strategies presented in Section 4.5. The transaction buffer (TB) used in all previous experiments is the best strategy for the TPC-C and reaches the highest throughput. The shared record buffer (SB) performs worse because the overhead of buffer management outweighs the caching benefits. The cache hit ratio of 1.42% is very low. The shared buffer with version set synchronization (SBVS) tested with cache unit sizes of 10 and 1000 has a considerably higher cache hit ratio (37.37% for SBVS1000). Nevertheless, this cannot compensate for the cost of additional update requests to the storage system. A key insight from these results is that with fast RDMA the overhead of buffering data does not pay off for workloads such as the TPC-C.
4.8 Concluding Remarks

In this chapter, we introduced Tell, a transactional database system based on the shared-data principle. Tell decouples transactional query processing and data storage into two independent layers and thus enables elasticity and workload flexibility. Data is stored in main memory in a distributed record manager that is shared among all database instances. To address the synchronization problem in shared-data environments, we presented a protocol to perform distributed multi-version concurrency control using lightweight atomic RMW operations for conflict detection. Furthermore, we explained how relational data can be efficiently stored and accessed to enable scalable query processing.

The experimental evaluation highlights the ability of the shared-data architecture to scale out to a high number of cores and emphasizes the importance of low-latency networks to achieve high performance. Furthermore, we compared Tell to alternative distributed databases and achieved a higher throughput than VoltDB and MySQL Cluster in the popular TPC-C benchmark. The shared-data architecture enables competitive performance while at the same time being elastic and independent of any workload assumptions. It is therefore a ideal data management architecture for heterogeneous and evolving workloads.
Concurrency Control Techniques

Database concurrency control is an essential mechanism in every data management system. It ensures that transactions can run in parallel without violating data integrity and has great impact on database scalability. Consequently, we extend our study of the shared-data architecture with an evaluation of different concurrency control approaches. In the previous chapter, we presented a MVCC protocol (i.e., snapshot isolation) that provided scalable performance for OLTP workloads. In this chapter, we introduce and evaluate alternative concurrency control mechanisms that have been adapted to operate in a shared-data context. In addition to SI, we present common techniques based on locking and timestamp ordering that enable different isolation levels. The techniques are not new and have been extensively compared in the past [CS84, BHG87, ACL87]. However, to the best of our knowledge, this is the first study that evaluates these techniques in the context of shared-data architectures given latest hardware trends.

Table 5.1 provides an overview of the studied approaches. Conceptually, the concurrency control category (optimistic or pessimistic) is most relevant. In optimistic concurrency control [KR81], transactions do not interfere with each other while running but check for conflicts (i.e., a violation of the integrity rules) on the accessed data before committing. In pessimistic concurrency control, transactions block as long as data access would cause a conflict. Pessimistic approaches are usually implemented with locks. Another important category is the isolation level. The strongest level, *serializable*, does not suffer from anomalies and is typically the most expensive isolation level to achieve. This level ensures that
Chapter 5. Concurrency Control Techniques

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Snapshot Isolation</th>
<th>Timestamp Ordering</th>
<th>Centralized Locking</th>
<th>Data-Coupled Locking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimistic/Pessimistic</td>
<td>Optimistic</td>
<td>Optimistic</td>
<td>Pessimistic</td>
<td></td>
</tr>
<tr>
<td>Isolation Level</td>
<td>Snapshot Isolation</td>
<td>Repeatable Read</td>
<td>Repeatable Read</td>
<td></td>
</tr>
<tr>
<td>Possible Conflicts</td>
<td>W-W</td>
<td>R-W and W-W</td>
<td>R-W and W-W</td>
<td></td>
</tr>
<tr>
<td>Conflict resolution</td>
<td>Abort Tx</td>
<td>Abort Tx</td>
<td>Block</td>
<td></td>
</tr>
<tr>
<td>Deadlock detection</td>
<td>-</td>
<td>-</td>
<td>Timeout</td>
<td></td>
</tr>
<tr>
<td>Deadlock resolution</td>
<td>-</td>
<td>Abort Tx</td>
<td>Abort Tx</td>
<td></td>
</tr>
<tr>
<td>External Service</td>
<td>Commit Manager</td>
<td>-</td>
<td>Locking Service</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5.1: Overview of concurrency control mechanisms for shared-data architectures

data is accessed as if transactions were executed in serial order. Lower isolation levels are vulnerable to anomalies (e.g., repeatable read is vulnerable to phantom reads). Our study implements the approaches of Table 5.1 with regard to efficiency in a shared-data context and therefore provides a specific isolation level for each approach. Obviously, additional isolation levels are possible for the different mechanisms. For instance, a lock-based mechanism can also provide serializability. Supporting additional isolation levels is subject to future work.

Tell is used as a platform to implement the alternative approaches. The concurrency control mechanism is integrated into the transaction management component of each processing node (Figure 4.1). PNs interact with the shared storage system and data access is performed as explained in Section 4.5. All approaches buffer data updates until commit time in order to apply the changes in a batched request and minimize the number of network interactions. Depending on the concurrency control approach, an external service (e.g., the commit manager for snapshot isolation) is required. To provide fault-tolerance, PNs write to a common transaction log located in the storage layer. The transaction log maintains a history of all data changes and enables recovery in the event of a PN failure. The recovery process details, including the decision to rollback or commit (i.e., rollforward) transactions, depend on the concurrency control mechanism.

This chapter is structured as follows: In a first part, we present the alternative concurrency control approaches. For each mechanism, we start with a brief description and explain required components. Then, we present the life-cycle of a transaction using the specific concurrency control approach. Finally, we explain the data recovery process in the
5.1 Snapshot Isolation

The snapshot isolation protocol has been described in Section 4.4 to illustrate how Tell works. Recall that SI is an optimistic MVCC protocol. Transactions operate on a valid “snapshot”. That is, a transaction is only allowed to access the record versions that were written by already completed transactions. Data updates are temporarily buffered and applied to the storage system during commit. To detect conflicts, updates are applied using LL/SC operations. The failure of a conditional write operation indicates a Write-Write conflict (i.e., the record has been altered by another transaction) and causes the transaction to abort. SI is not susceptible to Read-Write conflicts because read operations only access record versions that are not going to change. Furthermore, deadlocks never occur because updates are applied individually and atomically. SI requires an external service, the commit manager, that keeps track of active transactions and specifies the valid snapshot the moment a transaction is started.

5.2 Timestamp Ordering

Timestamp ordering (TO) [BG81, WMKN13] is an optimistic approach that enforces serial access using logical timestamps. Every transaction gets a timestamp assigned on begin that is used to determine if records can be accessed or not. In TO, a transaction is only allowed to access records that have not been updated by younger transactions (i.e., transactions with a higher timestamp). The protocol guarantees monotonically increasing and unique timestamps.

TO stores single record versions. Each record maintains a read and a write timestamp to remember the last time it was read and updated respectively. A read operation is successful if the timestamp of the transaction is greater or equal to the write timestamp of the record. If the condition does not hold, the read operation fails and the transaction aborts. A write operation can be performed if no younger transaction has read or modified the record. In other words, the transaction timestamp must be greater or equal to the read and the write timestamp of the record. As the success of a write depends on previous
read and write operations, TO is susceptible to Read-Write and Write-Write conflicts. Algorithm 1 illustrates the necessary conditions for read and write access.

The TO conditions prevent to read data that has been changed by a more younger transactions. However, they do not prevent transactions from accessing uncommitted data. Therefore, to enable recoverability and provide strict TO [BH87], we use a dirty flag to mark a record as being currently invalid. The flag is set before the record is modified and unset when the transaction has completed. If a read or write operation encounters an active dirty flag, it blocks until the record is available again. In order to prevent deadlocks, we ensure that a transaction never waits for younger transactions.

Our implementation provides the isolation level repeatable read and is vulnerable to phantom reads. The following example illustrates the anomaly: A new record inserted by an old transaction $T_1$ (with timestamp 1) will become visible in a range read of a younger transaction $T_2$ (with timestamp 2) that already accessed the range before $T_1$ committed. The inserted record has a write timestamp of 1 so that the read operation of $T_2$ does not violate the conditions of Algorithm 1. A solution to provide serializability would be to include timestamp information in the index and prevent newly inserted data from being accessed by active transactions.
5.2. Timestamp Ordering

Timestamp management is integrated into the storage system. We have extended RamCloud to provide new API functions to access records, modify timestamps, and set/unset dirty flags.

5.2.1 Life-Cycle of a Transaction

In the following, we describe the states of a transaction:

1. **Begin:** When started, a transaction gets a system-wide unique tid that acts as timestamp. tids are obtained by atomically incrementing a counter in the storage system.

2. **Running:** While running, a transaction can access any record. If a read operation fails because the TO conditions are not met, the transaction aborts. If a data operation encounters a dirty flag, it blocks until the flag is released. Updates are temporarily buffered.

3. (a) **Commit:** After writing the transaction log, all updates are applied to the storage system. Again, if the TO conditions are not met, a failed update causes the transaction to abort. Otherwise, the update sets the dirty flag and modifies the record. A delete operation sets an additional delete flag to indicate that the record should be removed on completion. Once all update operations have been executed, the indexes are updated. Finally, the log entry is deleted and the dirty flags are released.

   (b) **Abort:** Aborts require no actions, except if updates have already been applied. Applied records are rolled back and the dirty flags released.

5.2.2 Recovery and Fail-Over

TO writes to the transaction log before commit begins and deletes the log entries once the transactions have completed. As a result, the log only contains entries of transactions that are currently in the commit state. On PN failure, the log is used as a redo-log. That is, the recovery process will complete the commit of the affected transactions on behalf of the failed PN. The recovery process traverses the log, locates the transactions, and uses the write set of the corresponding log entries to complete the commit procedure. Either
all updates can be applied, or the entire transaction is rolled back. Moreover, blocked records (i.e., the dirty flag is set) of partially committed transactions are released. While recovery is executed, healthy nodes can continue processing transactions.

TO meta-data (i.e., timestamps and flags) is managed in the storage system and therefore is affected by SN failures. The read and write timestamps of every record are expendable and can get lost. They are reset to zero on recovery and Algorithm 1 ensures that serial execution order is restored. The dirty and delete flags, however, must not get lost because they indicate that a record was in an inconsistent state (i.e., uncommitted) the moment the SN failed. Therefore, the flags are synchronously replicated together with the updated data record. To unset the flags, a second request to the replicas is issued once the transaction has completed. This mechanism increases replication overhead but ensures correct recovery on SN failure.

5.2.3 Discussion of Alternatives

An alternative TO approach maintains multiple record versions (MVTO) [Ree83]. The advantage of MVTO is that read operations are always successful. In the presence of multiple versions, a transaction can access the version with the most recent write timestamp smaller or equal to its own timestamp (i.e., the TO read condition is met). Write operations are executed the same way as in strict TO. A disadvantage of MVTO, that causes substantial overhead, is the need for garbage collection to prevent the database from growing infinitely. Garbage collection requires coordination to keep track of completed transactions and involves additional network communication.

5.3 Centralized Locking

The third concurrency control mechanism is based on locking. In a lock-based model, transactions must acquire a lock before being granted the right to access or modify a record. Locking is pessimistic. That is, it controls resource access and forces transactions to wait until a lock becomes available. As dedicated access can be enforced, we store a single record version.

In order to maintain and manage all locks, this variant uses a centralized locking service accessible to all PNs. Before accessing a record, a transaction contacts the locking service
5.3. Centralized Locking

to acquire the lock and thereafter retrieves the record from the storage system. Lock acquisition is a synchronous process. That is, the locking service replies once the lock could be granted or a timeout was reached. In the meantime, the transaction is blocked.

Transactions operate according to Strict Two-Phase Locking (S2PL) [BHG87] that has been adopted in many RDBMS. In S2PL, a transaction first acquires all required locks before releasing them after commit or abort. Locks are acquired at the level of records in either shared or exclusive mode. Lock upgrades are possible. Shared locks grant read access and enable several transactions to read a record concurrently. Exclusive locks grant dedicated access and are required for write access. Exclusive locks cannot be acquired while the lock is already in shared or exclusive mode. Therefore, locking is susceptible to Read-Write and Write-Write conflicts. Range locks are not implemented yet. Therefore, our implementation is vulnerable to phantom reads and the provided isolation level is repeatable read.

Data access implies at least two network interactions: One to the locking service and another to the storage system. An optimization to reduce the number of requests is the concept of global/local locks. That is, a shared lock acquired globally from the locking service can be used locally on a PN by several transactions before being globally released. However, this mechanism prevents global fairness and can cause starvation. Section 5.4 presents an alternative locking variant that further reduces the number of network requests.

5.3.1 Centralized Locking Service

The locking service enables PNs to acquire and release locks on records. If a lock is available on request it is assigned. If the lock is not available, the request is enqueued. Every lock object maintains a queue for waiting requests. On lock release, the service looks for waiting requests and directly hands over the lock if possible. Lock requests that could not be served within a configurable timespan are timed out and rejected. Deadlocks are detected using timeouts. Deadlocked transactions will wait until their lock requests eventually timeout and abort.

To scale out the locking service, we hash-partition the key space across multiple nodes. Assuming lock requests are uniformly distributed, the locking service can scale linearly. However, to efficiently provide range locks, an alternative partitioning scheme (e.g., by table) would be necessary.
A locking service node can fail anytime, and as a result locks are lost. Transactions are no longer able to acquire new locks for the failed locking service node and abort. However, transactions that possess all locks are allowed to commit (because consistent data access is still guaranteed). Once they have completed, a replacement node is started and processing can continue.

5.3.2 Life-Cycle of a Transaction

In the following, we describe the states of a transaction:

1. **Begin**: In order to start, a transaction gets a unique $tid$. In contrast to TO, $tids$ have to be unique but not strictly increasing. For that reason, PNs can atomically increment the id counter by a high amount (usually 1000) to acquire a range of unique ids that can be assigned to new transactions on-demand. Getting $tids$ in batches reduces the synchronization overhead on the id counter (and the number of network requests in the system).

2. **Running**: While running, a transaction acquires shared or exclusive locks before reading or updating records respectively. If a lock request fails, the transaction aborts. In accordance with S2PL, locks are kept until the transaction is in the commit procedure. All updates are buffered.

3. **Commit**: First, the transaction log is written. Second, all updates and index changes are applied in a batched request. All writes should succeed as the locks guarantee exclusive access. Finally, the log entry is marked as committed and the locking service is instructed to release all locks.

(b) **Abort**: A manual abort releases acquired locks. As no changes have been applied, no rollback is necessary.

5.3.3 Recovery and Fail-Over

As soon as a PN fails, a two-step recovery process is started. First, all updates applied by committing transaction are undone. Second, the locks of all related records are released.
In order to identify the active transactions of the failed node we rely on meta-data. The moment a PN retrieves a new set of unique identifiers, we remember the range of $tids$ in a meta-data table in the storage system. Using that data, the recovery process is able to narrow down all potential $tids$ the PN could have used. We iterate backwards over the list of $tids$ and probe the log for matching entries. If an entry is found that is not marked committed or aborted, we have found an active transaction that needs to be rolled back. The log entry contains the old record versions to be restored. The database is in a consistent state once all non-committed transactions in the $tid$ range have been rolled back. As a final step, the locking service is instructed to release the locks owned by the failed PN. For this purpose, the locking service maintains an index with the locks a PN has acquired but not yet released.

5.4 Data-Coupled Locking

Another lock-based technique is called data-coupled. This approach is similar to the one presented in Section 5.3 but instead of relying on a centralized locking service, the locks are coupled to data records and managed in the shared store. The storage system is extended to support the same lock management mechanism previously performed in a separate process (i.e., the locking service). Locks can be acquired in shared or exclusive mode and deadlocks are detected using timeouts.

Data-coupled locking increases the computational overhead in the storage system and therefore requires more storage resources. However, compared to centralized locking, it provides a significant benefit: Several operations (i.e., lock and read, write and unlock) previously performed separately can be combined and executed with a single network request. Combined operations promise to reduce data access latencies but also require adjustments in the commit and the recovery process. In particular, the $WriteAndUnlock$ operation has implications: Once a record is applied and unlocked, it might be immediately modified by another transaction, and the rollback of the initial transaction would become very complex. Consequently, after the transaction log is written, we consider a transaction committed and ensure that all its updates get applied. During normal operation, the transaction owns the exclusive locks and the $WriteAndUnlock$ operations will succeed. On failure, we have to rollforward completed but not yet applied transactions.
Chapter 5. Concurrency Control Techniques

5.4.1 Life-Cycle of a Transaction

In the following, we describe the states of a transaction:

1. **Begin**: The transaction gets a unique \( tid \) (as described in Section 5.3.2).

2. **Running**: While running, the transaction can execute any query. Reads are performed by \( LockAndRead \) operations that acquire the lock and subsequently access the according record. Furthermore, \( LockAndRead \) operations enable to acquire exclusive locks on items that are about to be updated or inserted. Updates are temporarily buffered.

3. (a) **Commit**: First, the log entry is written. Log entries are strictly ordered to ensure correct recovery after SN failures. As soon as the log is written to the storage system, the transaction is considered committed and all updates are applied by \( WriteAndUnlock \). Then, the shared locks are released. Finally, the log entry is deleted to prevent updates from being re-applied unnecessarily.

   (b) **Abort**: Same as in centralized locking (Section 5.3.2).

5.4.2 Recovery and Fail-Over

In case of PN failure, we rollforward committed transactions and ensure all changes become visible. The recovery process iterates backwards over the log table, checks every log entry if it belongs to the failed node and if so ensures that the updates are applied (with \( WriteAndUnlock \) operation). The log entry is deleted as soon as all changes have been applied.

If a SN fails, the locks of all records managed by the SN are lost. New \( LockAndRead \) requests timeout and the requesting transactions abort. Transactions in the commit process fail to execute \( WriteAndUnlock \) operations and stop (the commit is completed by the recovery process). Once the data of the SN has been restored, the recovery protocol verifies data consistency. The log is traversed backwards and the \textit{write set} of all non-completed transaction is applied. The backward traversal guarantees that only the latest value of a record is applied and not an outdated version. Once the recovery process has completed, the storage node is ready to process new requests.
5.5 Experimental Evaluation

This section provides an experimental evaluation of the four concurrency control techniques described in the previous section. We first describe environment and methodology before presenting experimental results.

5.5.1 Benchmark, Environment, and Implementation

Every concurrency control mechanism is implemented in Tell (Chapter 4). The PN component processes queries and enables to execute ACID transactions based on the concurrency control mechanisms described in the Sections 5.1 to 5.4. PNs have a synchronous processing model. Each PN is configured to run with up to 12 threads per core. As previously explained, further increasing the number of threads leads to aggressive context switching and consumes a significant part of CPU time in kernel space. The processing layer interacts with the RamCloud storage system. RC was modified to provide the functionality as required by timestamp ordering (Section 5.2) and data-coupled locking (Section 5.4). Recall that RC organizes data in tables of KV-pairs that are hash-partitioned across all storage nodes. Furthermore, RC can synchronously replicate data to prevent data loss during failures. The benchmarking environment is a cluster with 12 servers connected by a 40 Gbit InfiniBand network (Section 4.7.1).

The primary benchmark used in the evaluation is the TPC-C benchmark described in Section 4.7.2. We performed measurements with the two mixes presented in Table 4.2 to evaluate the concurrency control mechanisms for alternative workload scenarios (read-mostly and write-heavy). Furthermore, we varied the size of the database. Depending on the experiment, we set the initial database size to either 10 WHs or 200 WHs. Using 10 WHs, we observed a high contention with several hotspots. In particular, the warehouse table with only 10 records is a major point of contention. Contention is considerably reduced when running the benchmark with 200 WHs. The two different settings with 10 WHs and 200 WHs enable to evaluate the concurrency control mechanisms in the presence (and the absence) of data contention. The TPC-C benchmark is configured and the load is generated as specified in Section 4.7.2.
Chapter 5. Concurrency Control Techniques

Figure 5.1: Scale out processing (write-intensive, vary PN, 7SN, RF1, 10WH)

<table>
<thead>
<tr>
<th></th>
<th>Snapshot Isolation</th>
<th>Timestamp Ordering</th>
<th>Centralized Locking</th>
<th>Data-Coupled Locking</th>
</tr>
</thead>
<tbody>
<tr>
<td>10WHs</td>
<td>17.91%</td>
<td>82.01%</td>
<td>92.44%</td>
<td>89.57%</td>
</tr>
<tr>
<td>200WHs</td>
<td>2.39%</td>
<td>30.57%</td>
<td>5.56%</td>
<td>7.55%</td>
</tr>
</tbody>
</table>

Table 5.2: New-order transaction abort rate (write-intensive, 8PN, 7SN, RF1, vary WH)

5.5.2 Write-Intensive Workloads

We first present results for the write-heavy TPC-C standard mix. In this experiment, we increased the number of PNs and studied the effect on throughput. All other components had enough resources and did not limit performance (7 SNs, 1 commit manager, 8 lock servers).

Figure 5.1 shows the achieved throughput in the presence of high-contention (10 WHs). SI performs best because it is only vulnerable to Write-Write conflicts. The other approaches (TO and locking) are exposed to Read-Write conflicts that cause many transactions to abort. Table 5.2 shows that the abort rate for new-order transactions (relevant for the TpmC metric) grew up to 92.44% for centralized locking. SI is more tolerant with regard
to data contention. Nevertheless, for SI as well the abort rate increased with the number of PNs. It reached 17.91% with 8PNs and explains the non-linear increase in TpmC.

Figure 5.2 presents results for the same experiment but with less contention (200 WHs). All concurrency control mechanisms scaled with the number of PNs. SI provided linear scale-out and achieved best throughput. The throughput of TO was lower as many transactions are in conflict. This is an effect of enforcing order using logical timestamps. For 8 PNs, TO achieved 680,388 TpmC, while SI achieved 958,187 TpmC. The proportional difference in throughput between TO and SI is approximately equal to the TO abort rate of 30.57% (with 8 PNs). Both locking approaches suffered from the overhead of lock management. By acquiring locks, transactions get delayed in accessing data, and as a result performance is negatively affected. Increasing the number of processing threads on every PN did not compensate for higher data access latencies and consequently, throughput could not match the one of SI. Although data-coupled locking involves no separate locking service, the throughput is lower than for centralized locking. The reason lies in the reduced ability to batch requests. While centralized locking enables to implicitly batch all concurrent requests to the shared store, data-coupled locking uses combined operations (i.e., \texttt{ReadAndLock} and \texttt{WriteAndUnlock}) that are only batched in the scope of a transaction.
The results highlight that multi-version optimistic concurrency control performs best for update-intensive workloads. Another relevant insight for shared-data architectures is the importance of request batching.

### 5.5.3 Read-Intensive Workloads

Figure 5.3 shows the throughput achieved with the read-intensive TPC-C mix. SI and TO scaled linearly and almost reached 4 mio transactions per minute (Tpm). Given the high read ratio, TO had much less Read-Write conflicts and could maintain the same throughput as SI. The optimistic concurrency control approaches perform better than the pessimistic ones. The performance difference between SI and locking is in the same range as for the update-intensive mix and reflects the overhead of lock management. Indeed, for every read or write operation, locks have to be acquired and released. Lock acquisition might involve waiting until the lock becomes available.
5.5. Experimental Evaluation

![Graph showing TpmC (in 1'000) vs. Total number of CPU cores for different isolation strategies: Snapshot Isolation, Timestamp Ordering, Centralized Locking, Data-Coupled Locking, and MySQL Cluster.]

**Figure 5.4:** Scale out cores (write-intensive, vary #cores, RF1, 200WH)

### 5.5.4 Resource Utilization

Figure 5.4 shows the performance given a specific number of cores. That is, the throughput achieved is put into relation with the computational resources used. The experiment is based on the TPC-C standard mix. In contrast to the previous experiments, we varied all components (PNs, SNs, and external services). Figure 5.4 shows the aggregated count of CPU cores utilized by the entire SUT. In our environment, a PN, a SN, and a locking service node required 4 cores each, while a commit manager (CM) and the management node (MN) used 2 cores. A minimal setup with 1 PN, 3 SNs and a MN runs on 18 cores. The resources were scaled out to maximize throughput for a given amount of cores. SI provided the best scale-out and reached over 1.2 Mio TpmC with 78 cores (11 PNs, 7 SNs, 2 CMs, 1 MN). The steps in the lines represent additional SNs that have less impact on throughput as new PNs. TO does not rely on external components and thus provided a better throughput/resource ratio than locking. The locking approaches provided similar performance. Centralized locking required additional locking service nodes and data-coupled locking more SNs to meet the resource demands of lock management. The largest centralized locking configuration with 78 cores consisted of 8 PNs, 5 SNs, 6 lock servers, and 1 MN. As a reference, we included the MySQL Cluster baseline results from Figure 4.6.
5.5.5 Replication

In this section, we discuss the performance impact of replication. Figure 5.5 and 5.6 compare no replication (RF1) with RF3 for both TPC-C mixes. RF1 is depicted by the dashed lines, while RF3 is depicted by the solid lines. As both locking approaches perform similarly, Figure 5.5 and 5.6 only include results for centralized locking.

In Tell, updates are synchronously transferred to all replicas, and thus replication increases update latency. Read operations access the master copy and are not affected by replication. Consequently, the higher the write-ratio the heavier the performance impact. For the write-intensive mix, we noted a throughput loss of 63.2% (SI, 8 PNs) with RF3 compared
to RF1. For the read-intensive mix, performance decreased by 25.7%. In SI, record versions are stored in one entry and on update all versions are replicated. Integrating version management into the storage layer, as suggested in Section 4.4.5, would allow to replicate single record versions and reduce the latency penalty. Replication cost is expected to drop to the level of the locking approach. For locking, throughput decreased by 58.1% and 11.9% for the write and read-intensive mixes respectively (8 PNs). The proportional cost of replication in TO is higher than for the other approaches as every update requires two requests to the replicas (Section 5.2). Recall that we need a second request to unset the dirty flag. With 8 PNs, throughput dropped by 66.94% and 34.32% for the write and read-intensive mixes respectively. The two leftmost columns of Table 5.3 show the measured mean transaction processing time and the standard deviation for RF1 and RF3. Running the TPC-C standard mix with RF3, resulted in more than two times higher mean transaction latency (compared to RF1). As expected, TO is affected most.

The effect of replication on performance is considerable in relation to the choice of concurrency control mechanism. Nonetheless, single-version replication with a single request to the replicas helps reduce overhead and is a desirable property with regard to concurrency control in replicated databases.
Figure 5.7: *Network comparison (write-intensive, vary PN, 7SN, RF1, 200WH)*

5.5.6 Network

Figure 5.7 compares the achieved peak throughput using an InfiniBand network to the performance of 10 Gbit Ethernet. InfiniBand provided a six times higher throughput than Ethernet independent of the number of processing nodes. The first and the last column of Table 5.3 indicate the latency differences between both networking technologies and highlight the importance of low-latency data access. InfiniBand uses RDMA and by-passes the networking stack of the operating system to provide much lower latencies.

5.6 Related Work

Concurrency control has been extensively studied since the beginning of database systems. Several excellent books and surveys provide an overview of concurrency control principles and present common techniques [ACL87, BG81, BHG87, Kum95].

Recently, concurrency control for main memory databases has received considerable attention. We distinguish two fundamental approaches:
First, a shared-nothing approach that partitions data and assigns each partition to a CPU core. Optimally, transactions are executed in serial order on a single partition without any concurrency at all (lock-less) [HAMS08]. H-Store [KKN+08], VoltDB [Vol14], or Hyper [KN11] are systems that implement this idea. They horizontally partition tables inside a node and serially execute stored procedures on each core. While single-partition transactions are processed very efficiently, cross-partition transactions require coordination. For instance, a naive approach uses partition locks that block serial execution to process cross-partition transactions. Consequently, the performance of the shared-nothing approach very much depends on how good the partitioning scheme is. Several papers present techniques to reduce the coordination overhead for cross-partition transactions. Jones et al. [JAM10] compare a lock-based approach with an optimistic method for cross-partition synchronization. The latter method speculatively executes single-partition transactions while waiting for cross-partition transactions to complete (and might require cascaded rollbacks). The authors show that speculative execution is well-suited for workloads with few multi-partition interactions and few aborts. DORA [PJHA10] partitions locks to reduce the contention of centralized lock management. PLP [PTJA11] partitions data among trees and enables flexible repartitioning.

The second approach for concurrency control in main memory databases is based on a shared-memory design. The shared-memory approach is at its core similar to the shared-data architecture. Larson et al. [LBD+11] compare three concurrency control approaches for shared-memory databases: single-version locking, pessimistic, and optimistic MVCC. This work is the basis for concurrency control in Hekaton [DFI+13], the main memory component of SQL Server. They conclude that single-version locking can be implemented efficiently for short running transactions. In a distributed environment our results argue otherwise. Moreover, they observe that MVCC is robust against contention. A behavior we can confirm. Silo [TZK+13] uses a lock-based optimistic concurrency control protocol with low-level optimizations to minimize contention on shared-memory and provide serializability. Optimistic concurrency control enables to batch the changes of a transaction and therefore reduce write access to shared-memory. Tell uses the same principle in a distributed context. At commit time, the protocol locks all records in the write set and checks for changes in the read set. Applied to a shared-data architecture, the approach would result in additional network requests to the storage layer. Nevertheless, the approach provides a valid alternative to atomic RMW operations and is subject to be evaluated as part of future work. An interesting observation is that the shared-memory
design of Silo outperforms a shared-nothing store once more than 20% of the transactions are cross-partition.

With regard to distributed databases, we revisit many of the system presented in Section 4.3 but focus on the implementation of concurrency control. Early shared-data databases [LARS92, JMNT97, CB03] use distributed global lock-managers to ensure data integrity. However, as previously discussed, these systems are based on traditional tightly-coupled disk-based database architectures and are therefore fundamentally different from Tell. ElasTras [DAEA13] has been designed for multi-tenancy. Transactions are possible in the scope of tenants (data partitions) on a single node. Inside a partition, ElasTras uses an optimistic concurrency control protocol with parallel validation [ABGS87]. The optimistic concurrency control approaches implemented in Tell perform parallel validation as well. However, the scope of transactions is not restricted. Megastore [BBC+11] implements transactions on top of BigTable [CDG+08] and uses Paxos [Lam98] to provide multi-version optimistic concurrency control. Data is grouped into collections of objects called entity groups. Within an entity group the protocol limits a single transaction to commit at a time. Across entity groups, 2PC is necessary. Google F1 [SVS+13] provides database features and a SQL layer on top of Spanner [CDE+12]. While Spanner provides serializable pessimistic transactions using S2PL, F1 adds the option to perform optimistic transactions with TO. Yet, conflict detection (i.e., verify the TO conditions) requires a pessimistic Spanner transaction. Spanner partitions data in tablets and transactions across tablets use a 2PC protocol. F1 introduces a hierarchical data model in order to implicitly cluster data into tablets and reduce the overhead of distributed transactions. Unlike Tell, Megastore and F1 are designed with regard to cross-datacenter replication and the reported latencies to perform a commit are 50-150 ms. Although this is sufficient for user-facing web applications, many latency critical business use-cases cannot be covered. OMID [GFJK+14] implements snapshot isolation on top of HBase [Apa14b]. The approach is very similar to our implementation of SI. However, OMID requires a centralized component for conflict detection and commit validation. The commit manager in Tell is only required to maintain a global snapshot. Brantner et al. [BFG+08] have proposed the system used in our AWS S3 variant (Section 2.3.1.5). The paper presents techniques to implement database features and transaction processing on top of S3. However, the system does not provide full ACID semantics.

Several additional distributed storage systems are sharing data. Percolator [PD10] is a processing system designed for batched execution. It supports global transactions,
and concurrency control is based on snapshot isolation. Because Percolator is based on BigTable [CDG+08], the implementation requires locks to correctly install updates during transaction commit. In contrast, our concurrency control approaches rely on non-blocking atomic RMW operations. Hyder [BRD11] is a transactional record manager. Records are stored in a shared append-only log structure, and updates are appended to the log in total order. Concurrency control is optimistic, multi-version, and enables serializability. A transaction is executed on the latest snapshot and once completed, a set of changes is added to the log. The main difference to Tell is that the set of changes is broadcasted to all processing servers in order to roll forward the transaction and maintain a local consistent state. Apache Hive [TSJ+10] is a data warehouse build on top of Hadoop that shares data and provides large-scale query execution via MapReduce. Recent projects such as Shark [XRZ+13] or Presto [Pre14] extend the Hadoop software stack with rich query support and enable scalable SQL for OLAP workloads. So far, these approaches do not provide transactional guarantees. Nonetheless, the techniques and approaches evaluated in this dissertation set the course for extending the mentioned systems with ACID transactions.

5.7 Concluding Remarks

We presented an evaluation of concurrency control mechanisms for distributed shared-data architectures. Shared-data architectures enable elasticity by decoupling data storage from query processing and transaction processing. We integrated variants of snapshot isolation, timestamp ordering, and locking into the Tell database system and conducted a real-world performance analysis to compare the alternative approaches.

The experimental evaluation has revealed that optimistic multi-version concurrency control, specifically snapshot isolation, performs best for write-heavy and read-intensive workloads. Moreover, SI is robust in the presence of contention. The results have also emphasized that shared-data architectures are highly susceptible to changes in data access latencies. In particular, replication and networking technology require careful consideration.

Compared to traditional partitioned databases, systems based on a shared-data architecture can provide competitive OLTP performance. Considering the remaining architectural advantages such as simple scale-out and transparent data distribution, they become a natural choice for elastic and dynamic cloud environments.
Conclusions

This dissertation studied database architectures with regard to the new requirements of cloud computing. That is, we evaluated to what extent scalability, fault-tolerance, and elasticity can all at once be implemented in distributed databases. While NoSQL storage systems fulfill the demands of the cloud by sacrificing fundamental properties such as ACID semantics, our objective was to keep the benefits of traditional relational databases, namely rich query support and full transactional guarantees. Our solution, a database system called Tell, meets the cloud computing requirements and provides the powerful features of RDBMS. Tell is based on a shared-data architecture and decouples query processing and transaction management from data storage. Both layers can be scaled out independently with minimal effort and therefore enable to dynamically adjust resource utilization to the exact needs of the application. Thus, elasticity, an elemental feature with regard to cloud infrastructures, becomes an inherent property of the database architecture. Furthermore, we showed that Tell can scale with the number of nodes, and we described how to ensure fault-tolerance. In the following, we recapitulate the contents of this dissertation and point out the most relevant results.
6.1 Summary

The first part of this dissertation analyzed the data service offerings of the big cloud providers. We described alternative architectural variants for transaction processing and presented alternative services (or combination of services) that implement the different architectures. The architectures adopted by many data services vary and consequently motivated us to conduct a comprehensive evaluation of the alternative options. To that end, we have developed a benchmark and outlined new experiments to take into account the specific characteristics of cloud computing. The benchmark is based on a modified version of the TPC-W benchmark that is popular both in academia and industry. With this benchmark, we experimentally evaluated eight cloud data services with regard to throughput, cost, and cost predictability. The results were surprising because the tested services varied considerably in terms of performance and cost. While some services were particularly cheap for small workloads, other services excelled at higher load (because of cost amortization). Moreover, some services quickly reached a scalability limit, whereas others scaled linearly with increasing load. An interesting variant implements database features on top of an object store (Amazon S3). We referred to this type of architecture as shared-data. The key benefit of the shared-data architecture is that processing is decoupled from storage. Thus, both layers can operate and scale-out autonomously. The benchmarked variant of the shared-data architecture (with low consistency guarantees) showed promising results and motivated a more detailed analysis.

In the second part of this dissertation, we evaluated the shared-data architecture with regard to transactional OLTP workloads. To that end, we have developed a distributed database system called Tell. Tell is based on the shared-data architecture and provides all the powerful features of relational databases. Tell overcomes the common challenges of shared-data systems. That is, it reduces the synchronization overhead by using specific techniques. For instance, we implemented an optimistic multi-version concurrency control protocol that relies on lightweight atomic RMW primitives for conflict detection. Moreover, we used request batching to reduce the synchronization overhead as much as possible. In addition, lock-free index structures guarantee that transactions are never prevented from making progress. Another issue with shared-data architectures we addressed is bad data locality. As data cannot be fully buffered on the processing nodes, we minimized data access latency by using fast networking technology as well as main memory storage. On top of that, we described several strategies to improve buffering and reduce
remote storage accesses. In summary, it is the combination of efficient techniques and new hardware trends that enables Tell to provide scalable OLTP performance. In fact, the experimental evaluation highlighted that Tell can scale with the number of nodes.

In the final part of the dissertation, we extended our analysis of the shared-data architecture by presenting and evaluating different concurrency control mechanisms specifically adapted to operate in a shared-data context. In addition to the initially proposed distributed variant of snapshot isolation, we described an optimistic protocol based on single-version timestamp ordering. Furthermore, we studied two mechanisms based on locking. The first mechanism uses a dedicated, centralized locking service running on separate nodes. The second mechanism integrates lock management into the storage layer. The key advantage of the latter is combined operations. That is, we no longer need separate requests to lock/read or write/unlock records. The performance evaluation revealed that optimistic multi-version concurrency control performs best for the shared-data architecture. Snapshot isolation even excels for write-intensive workloads in which optimistic approaches typically suffer from high conflict rates. The results have also emphasized that shared-data architectures are highly susceptible to changes in data access latencies. Specifically, the replication factor or the network technology have a much higher performance impact in comparison to the right choice of concurrency control mechanism.

All in all, this dissertation revisited the old idea of sharing data in the context of distributed databases. Using the right combination of techniques and state-of-the-art hardware, we showed that the shared-data architecture is a viable alternative to traditional partitioned databases. Specifically, with regard to dynamic cloud infrastructures, the shared-data architecture provides key benefits that make it one of the few candidates to address present and future data processing challenges.

6.2 Directions for Future Work

During the dissertation we pointed out several possible directions for future work. In this Section, we outline the most relevant topics.

Mixed OLTP and OLAP Workloads. The efficient combined execution of OLTP and OLAP workloads inside a single system is a fundamental challenge for data management in the close future. Today, both workloads are usually processed in separate systems, and
data analytics is performed on stale (outdated) data. Enabling real-time analytics implies a major competitive advantage for many business, and as a result many efforts are geared towards supporting mixed OLTP and OLAP queries in a single database system. In this dissertation, we highlighted that the shared-data architecture is a promising candidate for mixed workloads. In particular, because processing is decoupled from storage, distinct processing resources can be appointed for different workloads (OLTP and OLAP) accessing the same data. However, the execution patterns of analytical queries typically access large amounts of data. Transferring this data to the processing nodes would quickly saturate available network bandwidth and result in poor performance. As a potential solution, we sketched the possibility to push down relational operators into the storage layer. For instance, executing a shared scan over the entire data in the storage layer would enable to pre-filter the results of several analytical queries simultaneously. Existing approaches as well as new techniques will require a detailed analysis and a practical evaluation in the context of shared-data databases.

Multi-Tenancy. Multi-tenancy characterizes the ability of a system to support multiple independent application contexts. In particular, cloud services are often simultaneously used by a large number of users running multiple applications. As a consequence, a DBaaS solution must provide support for many tenants (application databases). In this dissertation, we evaluated the shared-data architecture in the context of a single application database. Nevertheless, the architecture is also suited to operate with multiple tenants. Again, the separation of concerns enables to assign single tenants to different processing nodes. For small loads, processing nodes can be shared among several tenants. For high loads, a single tenant can get dedicated access to multiple processing nodes. The placement of tenants has to be managed by a fault-tolerant administration node (or service). Many placement strategies have been studied in the past and can be applied to the shared-data architecture. With regard to data storage, alternative partitioning schemes can be used. The current approach that evenly distributes data across all available storage nodes might not be the optimal partitioning scheme. Alternatively, the data of a tenant could be assigned to a dedicated set of storage nodes. Conceptually, the design space for multi-tenant distributed databases spreads from completely autonomous application “silos” to full sharing of processing and storage resources. Analyzing this design space is a major step with regard to the development of a “full-blown” DBaaS solution.
6.2. Directions for Future Work

**Storage Abstractions.** In Tell, relational data is mapped to a simple but yet powerful key-value model. The design choice to use a simple storage scheme was motivated by the assumption that the amounts of managed data will explode and that performing ETL processes between data management systems is becoming more and more expensive (in time and cost). Consequently, alternative storage abstractions need to access the same single copy of the data. As part of a research project, we implemented an abstraction of the Hadoop Distributed Filesystem (HDFS) on RamCloud. As a result, the relational data used by Tell to process OLTP transactions could be accessed simultaneously to perform Hadoop MapReduce tasks. This internal project demonstrated the feasibility of alternative storage abstractions. Nonetheless, implementing additional abstractions with satisfactory performance characteristics is an interesting venue for future work. In particular, new requirements (e.g., real-time analytics) and distinct workloads (e.g., data streams or graph data) can be addressed by providing alternative ways to access the same data. Furthermore, the study of alternative storage abstractions will contribute to answer the fundamental question if a single physical representation is sufficient to efficiently process the same data with alternative processing models.

The directions for future work provide intriguing opportunities with potentially significant impact on the world of data management systems. We hope that this work will raise interest and motivate many engineers and researchers to follow the path we have engaged.
# List of Tables

2.1 Overview of studied cloud data services . . . . . . . . . . . . . . . . . . . . 23

3.1 Throughput, $/WIPS, and cost predictability (Feb. 2010) . . . . . . . . . . 44
3.2 Cost per WIPS in m$ (vary EB, Feb. 2010) . . . . . . . . . . . . . . . . . . 49
3.3 Total cost per day in $ (vary EB, Feb. 2010) . . . . . . . . . . . . . . . . . 51
3.4 Total cost per day in $ (vary EB, Oct. 2014) . . . . . . . . . . . . . . . . . 51
3.5 RDS scale-up: throughput, $/WIPS, and cost predictability (Feb. 2010) . . 56
3.6 Bulk-loading time and cost, database size, and storage cost (Feb. 2010) . . 57

4.1 Comparison of selected distributed databases and storage systems . . . . 67
4.2 Read and write-intensive workloads for the TPC-C benchmark . . . . . . . 84
4.3 Scale out commit managers (write-intensive, 8PN, 7SN, RF1, 200WH) . . . 88
4.4 TPC-C transaction response time for small and large configuration . . . . 90
4.5 Network latency (write-intensive, 8PN, 7SN, RF1, 200WH) . . . . . . . . 93

5.1 Overview of concurrency control mechanisms for shared-data architectures 96
5.2 New-order transaction abort rate (write-intensive, 8PN, 7SN, RF1, vary WH) 106
5.3 Average transaction latency (write-intensive, 8PN, 7SN, vary RF, 200WH) 110
List of Figures

2.1 Classic database architecture .................................. 14
2.2 Partitioned database architecture ............................... 16
2.3 Shared-data database architecture .............................. 18
2.4 Replication and caching in distributed databases .......... 20

3.1 Scalability experiment for cloud data services ............... 37
3.2 Cost experiment for cloud data services ....................... 37
3.3 Peak experiment for cloud data services ....................... 38
3.4 Fault-tolerance experiment for cloud data services ......... 38
3.5 Throughput comparison of cloud service architectures ...... 46
3.6 AWS S3 throughput ........................................... 47
3.7 AWS RDS throughput ......................................... 47
3.8 AWS SimpleDB throughput .................................... 47
3.9 Google AE/C throughput ...................................... 47
3.10 Cloud services cost per day (250 EBs, Feb. 2010) .......... 53
3.11 Cloud services cost per day (250 EBs, Oct. 2014) ......... 53
3.12 AWS MySQL cost factors ..................................... 54
3.13 AWS RDS cost factors ........................................ 54
3.14 AWS SimpleDB cost factors .................................. 54
3.15 AWS S3 cost factors ........................................... 54
# List of Figures

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.16</td>
<td>Google AE/C cost factors</td>
</tr>
<tr>
<td>3.17</td>
<td>MS Azure cost factors</td>
</tr>
<tr>
<td>4.1</td>
<td>Tell architecture and components</td>
</tr>
<tr>
<td>4.2</td>
<td>Storage and access of data records</td>
</tr>
<tr>
<td>4.3</td>
<td>Scale out processing (write-intensive, vary PN, 7SN, vary RF, 200WH)</td>
</tr>
<tr>
<td>4.4</td>
<td>Scale out processing (read-intensive, vary PN, 7SN, vary RF, 200WH)</td>
</tr>
<tr>
<td>4.5</td>
<td>Scale out storage (write-intensive, vary PN, vary SN, RF3, 200WH)</td>
</tr>
<tr>
<td>4.6</td>
<td>Partitioned databases (TPC-C standard, vary #cores, RF3, 200WH)</td>
</tr>
<tr>
<td>4.7</td>
<td>Partitioned databases (TPC-C shardable, vary #cores, vary RF, 200WH)</td>
</tr>
<tr>
<td>4.8</td>
<td>Network comparison (write-intensive, vary PN, 7SN, RF1, 200WH)</td>
</tr>
<tr>
<td>4.9</td>
<td>Buffering strategies (write-intensive, vary PN, 7SN, RF1, 200WH)</td>
</tr>
<tr>
<td>5.1</td>
<td>Scale out processing (write-intensive, vary PN, 7SN, RF1, 10WH)</td>
</tr>
<tr>
<td>5.2</td>
<td>Scale out processing (write-intensive, vary PN, 7SN, RF1, 200WH)</td>
</tr>
<tr>
<td>5.3</td>
<td>Scale out processing (read-intensive, vary PN, 7SN, RF1, 200WH)</td>
</tr>
<tr>
<td>5.4</td>
<td>Scale out cores (write-intensive, vary #cores, RF1, 200WH)</td>
</tr>
<tr>
<td>5.5</td>
<td>Replication (write-intensive, vary PN, 7SN, vary RF 200WH)</td>
</tr>
<tr>
<td>5.6</td>
<td>Replication (read-intensive, vary PN, 7SN, vary RF, 200WH)</td>
</tr>
<tr>
<td>5.7</td>
<td>Network comparison (write-intensive, vary PN, 7SN, RF1, 200WH)</td>
</tr>
</tbody>
</table>
Bibliography


Bibliography


Bibliography


Bibliography


Bibliography


Bibliography


