Diss. ETH No. 18832

Building Database Applications in the Cloud

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY
ZURICH

for the degree of
Doctor of Sciences

presented by
TIM KRASKA
Master of Science in Information Systems, Westfälische
Wilhelms-Universität Münster
Master of Information Technology, University of Sydney
born 29 September 1980
citizen of Germany

accepted on the recommendation of
Prof. Donald Kossmann, examiner
Prof. Elgar Fleisch, co-examiner
Prof. Samuel Madden, co-examiner
Prof. Nesime Tatbul, co-examiner

2010

Abstract

Cloud computing has become one of the fastest growing fields in computer science. It promises infinite scalability and high availability at low cost. To achieve this goal, cloud solutions are based on commodity hardware, are highly distributed and designed to be fault-tolerant against network and hardware failures. Although sometimes considered as a hype, the main success of cloud computing is not technology-driven but economical. Cloud computing allows companies to outsource the IT infrastructure and thus, to profit from a shorter time-to-market, the economies of scale and the leverage effect of outsourcing.

Although the advantages of using cloud computing for building web-based database applications are compelling, so far cloud infrastructure is also subject to certain limitations. By today, there exists no consensus on cloud services, thus different providers offer different functionality and interfaces, which makes it hard to port applications from one provider to another. Furthermore, the systems sacrifice functionality and consistency to allow for better scaling and availability. If more functionality and/or consistency is required it has to be built on top. In addition, cloud providers use different programming languages for different layers of the application, creating a jungle of languages and services.

The purpose of this thesis is to explore how web-based database applications with low or high consistency requirements can be developed and deployed on different cloud infrastructure providers. The main contributions of this dissertation are associated with three topics:

The first part of the thesis proposes an architecture for building web-based database applications on top of an abstract cloud API. Using the API as a generalization of existing cloud offerings, the system is not bound to a single cloud provider and can easily be ported to other providers. The architecture is modular and utilizes state-of-the-art database techniques. In order to ensure different levels of consistency, the architecture is based on a set of newly developed protocols guaranteeing eventual consistency up to serializability.

In the second part of the thesis, we propose Consistency Rationing as a new transaction paradigm, which not only allows to define the consistency guarantees on the data instead of at transaction level, but also allows for automatically switching consistency guarantees at run-time. As high consistency implies high cost per transaction and, in some situations, reduced availability, low consistency is cheaper but it might result in higher operational cost because of, e.g., overselling of products in a web shop. Consistency Rationing provides the framework to manage this trade-off between consistency and cost in a fine-grained way. We present a number of techniques that make the system dynamically adapt the consistency level by monitoring the data and/or gathering temporal statistics of the data. Thus, consistency becomes a probabilistic guarantee which can be balanced against the cost of inconsistency.

The last part of the dissertation is concerned with XQuery as a unified programming model for the cloud and, in particular, the missing capabilities of XQuery for windowing and continuous queries. XQuery is able to run on all layers of the application stack, is highly optimizable and parallelizable, and is able to work with structured and semi-structured data. For these reasons, XQuery has already been proposed as a programming language for the web. However, XQuery so far lacks the ability to process continuous queries and windowing, which are important aspects for message processing, monitoring systems and even document formatting (e.g., for pagination or table formatting). In the last part of the thesis, we present two extensions for XQuery. The first extension allows the definition and processing of different kinds of windows over an input sequence, i.e., tumbling, sliding, and landmark windows. The second one extends the XQuery data model (XDM) to support infinite sequences. These extensions enable using XQuery as a language for continuous queries. In 2008, the windowing extension for XQuery has been accepted for the upcoming XQuery 1.1 standard.

Zusammenfassung

Cloud Computing ist eines der am schnellsten wachsenden Gebiete in der Informatik. Es verspricht nahezu unbegrenzte Skalierbarkeit und höchste Erreichbarkeit zu niedrigen Kosten. Um diese Eigenschaften zu erreichen, basieren Cloud-Lösungen auf Standardhardware, sind hoch verteilt, und so konzipiert, dass sie eine hohe Fehlertoleranz gegenüber Netzwerk- und Hardwareausfälle aufweisen. Cloud Computing wird gelegentlich als ein Hype-Thema bezeichnet. Dabei basiert der Erfolg bisher nicht in erster Linie auf technologischen Neuerungen, sondern in der Tat auf ökonomischen Überlegungen. So erlaubt es Cloud Computing, Infrastruktur outzusourcen, und so von kürzeren Markteintrittszeiten und (wie beim herkömmlichen Outsourcing) von Skalen- und Leverageeffekten zu profitieren.

Obwohl die Vorteil von Cloud Computing-gestützten webbasierten Datenbankanwendungen auf der Hand liegen, geben die derzeitigen Angebote an Cloud-Infrastruktur doch Grenzen vor. So besteht nur ein geringer Konsens bzgl. der offerierten Dienste. Die verschiedenen Anbieter bieten unterschiedlichste Funktionalitäten und Schnittstellen an, was einen Wechsel von einem Anbieter zum anderen sichtlich erschwert. Im allgemeinen wurde bei der Entwicklung der Systeme Skalierbarkeit und Erreichbarkeit der Vorrang gegeben und dabei auf Datenbankfunktionalität (wie z.B. eine Query Sprache) und höhere Datenkonsistenz verzichtet. Zusätzliche Funktionalität muss vom Anwender entwickelt werden. Des Weiteren verwenden die diversen Anbieter unterschiedliche Programmiersprachen für die verschiedenen Layer der Anwendung und vervollkommnen so den Dschungel an Diensten und Sprachen.

Das Ziel dieser Dissertation ist es, einen Weg aufzuzeigen, wie webbasierte Datenbankanwendungen mit sowohl niedrigen als auch hohen Konsistenzanforderungen entwickelt und flexibel auf verschiedene Cloud-Lösungen aufgesetzt werden können. Drei Themengebiete werden hierbei aufgegriffen:

Im ersten Teil dieser Dissertation wird eine allgemeine Coud API definiert und darauf aufbauend eine Architektur für die Entwicklung webbasierter Datenbankanwendungen vorgeschlagen. Die Cloud API stellt eine Verallgemeinerung bestehender Cloud-

Services dar und erlaubt so einen problemlosen Wechsel von Anbietern. Die anvisierte Architektur ist modular und verwendet aktuelle Datenbanktechniken. Für die unterschiedlichen Konsistenzanforderungen wurden eine Reihe neuer Protokolle entwickelt, die je nach Anforderungen kombiniert werden können.

Der zweite Teil der Dissertation stellt mit dem Konzept des Consistency Rationings ein neues Transaktionsparadigma vor. Konsistenzgarantien werden auf der Daten- und eben nicht auf der Transaktionsebene definiert und es besteht die Möglichkeit, automatisch während der Laufzeit zwischen verschiedenen Garantien zu wechseln. Höhere Konsistenzanforderungen implizieren höhere Kosten je Transaktion und gegebenenfalls eine reduzierte Erreichbarkeit. Hingegen erscheint eine weniger strikte Konsistenzgarantie auf den ersten Blick billiger, kann aber zu weiteren operationellen Kosten, z.B. durch die Verärgerung eines Kunden im Falle der Nichtlieferung eines Produktes, führen. Consistency Rationing bietet in diesem Zusammenhang die Möglichkeit, Konsistenz und (Gesamt-)Kosten gegeneinander abzuwägen. Es werden eine Anzahl an Techniken und Strategien vorgestellt, die eine dynamische Anpassung der Konsistenzanforderung auf Basis der Datenbewegungen und / oder temporärer Statistiken steuern. Konsistenz wird somit zu einer stochastischen Garantie, die es gegen die Kosten durch Inkonsistenz abzuwägen gilt.

Der letzte Teil der Dissertation beschäftigt sich schließlich mit XQuery als universelle Programmiersprache für die Cloud. Insbesondere werden Windowing und kontinuierliche Queries behandelt. XQuery ist grundsätzlich auf allen Layern einer Anwendung einsetzbar, hoch optimierbar und parallelisierbar, und kann strukturierte und semistrukturierte Daten verarbeiten. Aus diesem Grunde wurde XQuery schon als *die* Sprache für das Web vorgeschlagen. Allerdings unterstützt XQuery bisher weder kontinuierliche Queries noch Windowing, beides wichtige Aspekte beim Verarbeiten von Nachrichten, der Überwachung von Systemen oder dem Formatieren von Dokumenten (z.B. Pagination oder Tabellenformatierung). In diesem Teil werden zwei Erweiterungen für XQuery vorgestellt, die eben diese Mängel adressieren. Zum einen werden Semantik und Syntax für die Unterstützung sogenannter Tumbling, Siding und Landmark Windows aufgezeigt, zum anderen das XQuery Data Model (XDM) erweitert, um auch kontinuierliche Queries zu erlauben. Die Windowing-Erweiterung von XQuery wurde mittlerweile als Bestandteil des künftigen XQuery 1.1 Standards akzeptiert.

Acknowledgements

From the first week of my Master thesis in the database group of Donald Kossmann at ETH Zurich, it was clear to me that this was the place where I wanted to do my PhD and, luckily, there was not much need to convince Donald about this idea. Although there were ups and downs during my years at ETH, I have never regretted it since. We were among the very first who combined the arising topic of cloud computing with database technology, and thus in the lucky position of doing research on the bleeding edge of technology. Among many other things, I had the possibility to work on many different projects around cloud computing and XML/XQuery technology, became a W3C member, helped to bootstrap the Zorba XQuery processor, and made contact with many leading international researchers, as well as people from the industry. All this broadened my horizon, gave me valuable insights for my research and helped me to see problems from different angles. Here, I would like to acknowledge those who assisted and supported me in this challenging and rewarding time.

First of all, I am truly grateful to my supervisor, Prof. Donald Kossmann, for guiding me not only through my life as a PhD student but also inspiring me in other ways. He is a remarkably visionary person who motivated me constantly. He often engaged me in many fruitful discussions, was open-minded in countless chats and offered me so many great opportunities.

I am also grateful for the work and availability of the other members of my committee; Prof. Elgar Fleisch, Prof. Samuel Madden and Prof. Nesime Tatbul. They provided me with interesting comments and questions that helped me to improve my work. I thank Prof. Ueli Maurer for being available to chair my examination committee.

Throughout this thesis it was a great pleasure to collaborate and to be in contact with many colleagues from ETH, as well as people outside ETH in research and industry. In particular, I would like to thank Daniela Florescu. It is hard to keep up with her speed, but if you try, it is amazing what you can get out of it. I would also like to thank Peter M. Fischer. I was always able to count on him and he was in endless ways helpful with advice and friendship. Irina Botan and Rokas Tamosevicius for their

involvement in building the Windowing XQuery extension. Matthias Brantner and David Graf for all the nights spent at ETH while building the first prototype and protocols for the first part of this thesis. A special thanks goes to Martin Hentschel and Prof. Gustavo Alonso. Without them the Consistency Rationing part would not have been possible. Jena Bakula and Simonetta Zysset for proof-reading parts of this thesis and all the help during my PhD.

I would also like to thank all my friends and colleagues in the Systems Group and ETH Zurich, not only for the active discussions but in particular for all the fantastics things we did together, such as skiing or the yearly cheese fondue. To Carsten and Dani, Marcos, Cristian, Rokas, Peter and Anita, a special thank you. Without you, I would not have so many nice memories and, hopefully, the whale watching will never end.

My special appreciation goes to my parents, Bärbel and Klaus, and my little sister, Rike, for all the support and love they have given me throughout my life and the believe in me.

Finally, I would like to thank Karin, who was the biggest help and probably ”suffered” the most.

Contents

1 Introduction **1**

1.1 Background & Motivation . . . 1

1.2 Problem Statement . . . 2

1.3 Contribution . . . 4

1.3.1 Building a Database on top of Cloud Infrastructure . . . 5

1.3.2 Consistency Rationing . . . 5

1.3.3 Windowing for XQuery . . . 6

1.4 Structure . . . 7

2 State of the Art **9**

2.1 Cloud Computing . . . 9

2.1.1 What is Cloud Computing? . . . 9

2.1.2 Cloud Service Overview . . . 10

2.1.2.1 Infrastructure as a Service (IaaS) . . . 10

2.1.2.2 Platform as a Service (PaaS) . . . 13

2.1.2.3 Software as a Service (SaaS) . . . 15

2.2 Web-Based Database Applications in the Cloud . . . 16

2.2.1 Applications using PaaS . . . 16

2.2.2 Applications using IaaS . . . 16

2.3 Cloud Storage Systems . . . 18

2.3.1 Foundations of Cloud Storage Systems . . . 19

2.3.1.1 The Importance of the CAP Theorem . . . 19

2.3.1.2 Consistency Guarantees: ACID vs. BASE 20
2.3.1.3 Techniques 20
2.3.2 Cloud Storage Services 22
2.3.2.1 Commercial Storage Services 23
2.3.2.2 Open-Source Storage Systems 25
2.4 XQuery as the Programming Model 27
2.4.1 Programming Models Overview 27
2.4.2 What is XQuery? 29
2.4.3 XQuery for Web-Based Applications 30

3 Building a Database on Top of Cloud Infrastructure 33
3.1 Motivation 33
3.1.1 Contributions 34
3.1.2 Outline 35
3.2 Amazon Web Services 35
3.2.1 Storage Service 36
3.2.2 Servers 39
3.2.3 Server Storage 40
3.2.4 Queues 40
3.2.5 Indexing Service 41
3.3 Reference Cloud API for Database Applications 42
3.3.1 Storage Services 42
3.3.2 Machine Reservation 43
3.3.3 Simple Queues 44
3.3.4 Advanced Queues 45
3.3.5 Locking Service 46
3.3.6 Advanced Counters 47
3.3.7 Indexing 48
3.4 Database Architecture Revisited 49

3.4.1 Client-Server Architecture 49
3.4.2 Record Manager 52
3.4.3 Page Manager 53
3.4.4 Indexes 54
3.4.4.1 B-Tree Index 55
3.4.4.2 Cloud Index 55
3.4.5 Logging 56
3.4.6 Security 57
3.5 Basic Commit Protocols 58
3.5.1 Overview 60
3.5.2 PU Queues 61
3.5.3 Checkpoint Protocol for Data Pages 63
3.5.4 Checkpoint Protocol for Collections 66
3.5.5 Checkpoint Protocol for Client-Side B-Trees 69
3.5.6 Unsafe Propagation Time 70
3.5.7 Checkpoint Strategies 71
3.6 Transactional Properties 74
3.6.1 Atomicity 74
3.6.2 Consistency Levels 77
3.6.3 Atomicity for Strong Consistency 79
3.6.4 Generalized Snapshot Isolation 82
3.6.5 2-Phase-Locking 86
3.7 Implementation on Amazon Web Services 87
3.7.1 Storage Service 87
3.7.2 Machine Reservation 87
3.7.3 Simple Queues 87
3.7.4 Advanced Queues 88
3.7.5 Locking Service 89
3.7.6 Advanced Counters 89

3.7.7 Indexing 90
3.8 Performance Experiments and Results 91
3.8.1 Software and Hardware Used 91
3.8.1.1 TPC-W Benchmark 91
3.8.2 Architectures and Benchmark Configuration 93
3.8.3 Response Time Experiments 95
3.8.3.1 EU-BTree and EU-SDB 95
3.8.3.2 EC2-BTree and EC2-SDB 97
3.8.4 Cost of Consistency Protocols 98
3.8.4.1 EU-BTree and EU-SDB 98
3.8.4.2 EC2-BTree and EC2-SDB 100
3.8.5 Scalability 103
3.8.6 Tuning Parameters 104
3.8.6.1 Page Size 105
3.8.6.2 Time-to-Live 106
3.8.7 Bulk-Loading Times 107
3.9 Related Work 107
3.10 Summary 109

4 Consistency Rationing **111**
4.1 Introduction 111
4.1.1 Contributions 113
4.1.2 Outline 113
4.2 Use Cases 114
4.3 Consistency Rationing 116
4.3.1 Category C - Session Consistency 116
4.3.2 Category A - Serializable 117
4.3.3 Category B - Adaptive 117
4.3.4 Category Mixes 118

4.3.5 Development Model 119
4.4 Adaptive Policies 119
4.4.1 General Policy 120
4.4.1.1 Model 120
4.4.1.2 Temporal Statistics 121
4.4.1.3 Setting the Adaptive Threshold 122
4.4.2 Time Policies 123
4.4.3 Policies for Numeric Types 124
4.4.3.1 Fixed Threshold Policy 124
4.4.3.2 Demarcation Policy 125
4.4.3.3 Dynamic Policy 126
4.5 Implementation 127
4.5.1 Configuration of the Architecture 128
4.5.2 Logical Logging 128
4.5.3 Meta-data 129
4.5.4 Statistical Component and Policies 129
4.5.5 Implementation Alternatives 131
4.6 Experiments 132
4.6.1 Experimental Setup 132
4.6.2 Experiment 1: Cost per Transaction 135
4.6.3 Experiment 2: Response Time 136
4.6.4 Experiment 3: Policies 137
4.6.5 Experiment 4: Fixed Threshold 139
4.7 Related Work 140
4.8 Conclusion 142

5 Windowing for XQuery **145**
5.1 Introduction 145
5.2 Usage Scenarios 148

5.2.1 Motivating Example 148
5.2.2 Other Applications 149
5.3 FORSEQ Clause 150
5.3.1 Basic Idea 150
5.3.2 Types of Windows 152
5.3.2.1 Tumbling Windows 153
5.3.2.2 Sliding and Landmark Windows 155
5.3.3 General Sub-Sequences 156
5.3.4 Syntactic Sugar 157
5.3.4.1 End of Sequence 157
5.3.4.2 NEWSTART 157
5.3.5 Summary 157
5.4 Continuous XQuery 158
5.5 Examples 160
5.5.1 Web Log Analysis 160
5.5.2 Stock Ticker 160
5.5.3 Time-Outs 161
5.5.4 Sensor State Aggregation 162
5.6 Implementation 163
5.6.1 MXQuery 163
5.6.2 Plan of Attack 165
5.6.3 Run-Time System 166
5.6.3.1 Window Iterators 166
5.6.3.2 Window Management 166
5.6.3.3 General `FORSEQ` 168
5.6.4 Optimizations 168
5.6.4.1 Predicate Move-Around 169
5.6.4.2 Cheaper Windows 169
5.6.4.3 Indexing Windows 170

5.6.4.4 Improved Pipelining 171
5.6.4.5 Hopeless Windows 171
5.6.4.6 Aggressive Garbage Collection 171
5.7 Experiments and Results 172
5.7.1 Linear Road Benchmark 172
5.7.2 Benchmark Implementation 173
5.7.3 Results 174
5.8 Related Work 175
5.9 Summary 177

6 Conclusion **179**
6.1 Summary of the Thesis 179
6.2 Ongoing and Future Work 181

A Appendix **185**
A.1 Node Splitting 185
A.2 XQuery 1.1 Window BNF 187

List of Tables **189**

List of Figures **191**

List of Algorithms **193**

Bibliography **195**

Chapter 1

Introduction

1.1 Background & Motivation

The web has made it easy to provide and consume content of any form. Building a web page, starting a blog, and making them both searchable for the public have become a commodity. Nonetheless, providing an own web application/web service still requires a lot of effort. One of the most crucial problems is the cost to operate a service with ideally 24×7 availability and acceptable latency. In order to run a large-scale service like YouTube, several data centers around the world are needed. Running a service becomes particularly challenging and expensive if the service is successful: Success on the web can kill! In order to overcome these issues, utility computing (a.k.a., cloud computing) has been proposed as a new way to operate services on the internet [RW04].

It is the goal of cloud computing - via specialized utility poviders - to provide the basic ingredients such as storage, CPUs, and network bandwidth as a commodity at low unit cost. For example, in the case of utility storage, users do not need to worry about scalability because the storage provided is virtually infinite. In addition, utility computing provides full availability, i.e., users can read and write data at any time without ever being blocked. The response times are (virtually) constant and do not depend on the number of concurrent users, the size of the database, or any other system parameter. Furthermore, users do not need to worry about backups. If components fail, it is the responsibility of the utility provider to replace them and make the data available using replicas in the meantime. Another important reason to build new services based on cloud computing is that service providers only pay for what they get, i.e., pay by use. No upfront investments are required and the cost grows linearly and predictably with

the usage. Although sometimes considered as a hype, the main success of cloud computing is not technology-driven but economical. Cloud computing allows companies to outsource the IT infrastructure and thus, profit from the economics of scale and the leverage effect of outsourcing. For this reason, it is unlikely that it is just a short-term trend.

Although the advantages for building applications in the cloud are compelling, they come with certain limitations. By today, there exists no consensus on cloud services. Thus, different providers offer different functionality and interfaces, which makes it hard to port applications from one provider to another. Furthermore, the systems sacrifice functionality and consistency to allow for better scaling and availability. If more functionality and/or consistency is required it has to be built on top. Although some cloud providers offer best-practice guidelines on building applications in the cloud, the new trade-offs - especially for applications which may require stronger consistency guarantees (such as database applications) - are not addressed at all.

1.2 Problem Statement

Cloud computing is still a rather new field, which is not yet entirely defined. As a result, many interesting research problems exist, often combining different research areas such as databases, distributed systems or operating systems. This dissertation focuses on how to build web-based database applications on top of cloud infrastructure. In particular, the following problems are addressed:

- **Infrastructure API**: Today, the cloud service offerings vary significantly and no standard exists. Thus, portability between services is not guaranteed. There does not even exist a consensus what the right services are. Identifying a minimal set of services and a corresponding API, which would allow to build advanced applications, significantly lowers the burden of moving applications into the cloud. Such a reference API is important as a basis for all further developments in this area.

- **Architecture**: Although more and more web applications are moved to the cloud, the right architecture is still not defined. The optimal architecture would preserve the scalability and availability of utility services and achieve the same level of consistency as traditional database systems (i.e, ACID transactions). Unfortunately, it is not possible to have it all as Brewer's CAP theorem states. The CAP theo-

rem proves that availability, tolerance against network partitions, and consistency cannot be achieved at the same time [GL]. Given the fact that in bigger systems network partitions are happening, most infrastructure services sacrifice consistency in favor of availability [Ama09d, CDG+06, CRS+08]. If an application requires a higher consistency, it has to be implemented on top. One of the problems addressed in this thesis is how to design a database of infrastructure services to benefit from the existing features and at the same time provide different levels of consistency for the different kinds of application requirements. Additional questions, which arise when designing the architecture for web-based database applications concern the hosting location of the application logic and the implementation of indexes. For instance, is it beneficial to run the application logic on the client or should it run on a server provided by the utility provider (e.g., on a rented virtualized machine)? Is it beneficial to push query processing into the cloud and use services such as SimpleDB as opposed to implementing indexing on the middle-tier?

- **Application Consistency:** Consistency plays an important role in the context of cloud computing. It not only has a direct effect on the availability but also impacts the performance and cost. In large-scale systems, high consistency implies more messages which have to be sent between the servers if perfect partitioning is not feasible. Here, perfect partitioning refers to a partitioning scheme where all possible transactions can be executed on a single server of the system. In most application (e.g., a web shop, social network applications, flight booking etc.) this is not feasible if scalability through scale-out should be preserved. For example, in a web shop all products can be in a single shopping cart. Thus, it is impossible to partition the web shop by products, but other partitions (e.g., by customer) also do not help, as again, the products are the overlapping parts. Without perfect partitioning, consistency requires more messages to coordinate reads and writes. For example, protocols for weak consistency (e.g., eventual consistency) typically do not have a validation phase and often even allow detaching the update propagation from the transaction. As a result, those protocols have very low overhead. In contrast, strong consistency (e.g., serializability with 2-Phase-Locking or snapshot isolation with 2-Phase-Commit) requires several extra messages to gain locks, propagate changes etc. As message transfer is orders of magnitudes more expensive (in particular across data-centers) than in-server memory or disk access, high consistency implies higher latency. Further, as in the cloud service calls and traffic are priced, strong consistency also implies higher costs. However, strong consistency avoids penalty costs because of inconsistency (e.g., overselling of

products in a web shop). In comparison, low consistency leads to lower costs per operation but might result in higher penalty costs. How to balance costs, availability and consistency for different application requirements is a non-trivial task and addressed in this thesis.

- **Programming Languages:** Another challenge of moving applications into the cloud is the need to master multiple languages [Hay08]. Many applications rely on a backend running SQL or other (simplified) languages. On the client side, JavaScript embedded within HTML is used and the application server, standing between the backend and the client, implements the logic using some kind of scripting language (such as PHP, Java and Python). All layers typically communicate through XML messages. To overcome the variety of languages, XQuery/XScript has already been proposed as a unified programming language, that is able to run on all layers [FPF+09, CEF+08, CCF+06, 28m09]. However, XQuery so far lacks capabilities to process message streams, which is an important aspect for cloud applications in order to be able to combine RSS feeds or to monitor services. One of the main concepts for stream processing is windowing. Windowing allows to partition a stream into sub-sequences. However, this functionality is also required e.g. for pagination of web-pages making such an extension even more valuable for web-application development.

1.3 Contribution

Instead of presenting a whole system, this thesis concentrates on specific aspects of building web-based database applications in the cloud. The complete system leveraging the research results of this thesis is currently built by the 28msec Inc. with the Sausalito product [28m09].

The main contributions of this dissertation are associated with three topics: How to build a database on top of cloud infrastructure, consistency rationing, and a XQuery extension in order to build applications with XQuery. In the following, we outline the main contributions in each of those areas below.

1.3.1 Building a Database on top of Cloud Infrastructure

The first building block studies how to build the database layer for web-based applications on top of cloud infrastructure. In particular, the following contributions are made:

- We define a set of basic services and a corresponding (abstract) API, forming the foundation to develop database applications in the cloud for all kinds of consistency requirements. The API is a generalization of existing cloud offerings. Hence, the API can be implemented on top of different cloud provider's services, which increases the portability.
- We propose an architecture for building web-based database applications on top of cloud infrastructure. Using the pre-defined service API, the application is not bound to a single cloud provider and can be easily ported to other providers. The architecture is modular and utilizes state-of-the-art database techniques. Furthermore, the architecture enables to host the application logic (including the database operations) at the middle-tier as well as at the client-tier.
- We present a set of newly developed alternative protocols in order to ensure different levels of consistency. As stated earlier, most storage services guarantee only eventual consistency, which has no support to coordinate and synchronize parallel access to the same data. This thesis presents a number of protocols in order to orchestrate concurrent updates and achieve higher levels of consistency, such as monotonicity, snapshot isolation or serializability.
- We provide results of performance experiments with the TPC-W benchmark to study the costs (response time and $) of running a web-based application at different levels of consistency, client-server architectures, and indexes on top of utility services.

The outcome of this part, published at SIGMOD 2008 [BFG+08], is patented under US20090177658, and in use by 28msec Inc. in the Sausalito product [28m09].

1.3.2 Consistency Rationing

We propose consistency rationing as a new transaction paradigm, which not only allows to define the consistency guarantees on the data instead of transaction level, but also

allows to automatically switch consistency guarantees at run-time. That is, consistency rationing provides the framework to manage the trade-off between consistency and costs in a fine-grained way. In more detail, the following contributions are made:

- We introduce the concept of *Consistency Rationing*, a new transaction paradigm, that not only allows designers to define the consistency guarantees on the data instead at the transaction level, but also enables to automatically switch consistency guarantees at runtime.
- We define and analyze a number of policies to dynamically switch consistency protocols at run-time. Our experiments show that dynamically adapting the consistency outperforms statically assigned consistency guarantees.
- We introduce the notion of probabilistic guarantees for consistency (i.e., a percentile) using temporal statistics for numerical and non-numerical values.
- We present a complete implementation of Consistency Rationing using the previously developed architecture. We report on the costs ($) and performance of running the TPC-W benchmark at several consistency categories, mixes of categories, and different policies of automatically switching the consistency requirements. The results of the experiments provide important insights into the cost of running such systems, the cost structure of each operation, and the cost optimization using appropriate models.

The result of this part has been published at VLDB 2009 [KHAK09].

1.3.3 Windowing for XQuery

The last part of the dissertation is concerned with XQuery as a programming language and in particular, the missing capabilities of XQuery for stream processing and pagination. We propose syntax and semantics to define and process complex windows using XQuery. Windowing describes the concept of selecting sub-sequences of an underlying stream. Although originally intended for stream processing, windowing also turned out to be useful in many other scenarios like pagination of results for web-pages. In summary, the following contributions have been made to enable stream processing for XQuery:

- Window Queries: We define the syntax and semantics of a new `FORSEQ` clause in order to define and process complex windows using XQuery.

- Continuous Queries: We propose a simple extension to the XQuery data model (XDM) in order to process infinite data streams and use XQuery for continuous queries.
- Implementation Design: We show that the extensions can be implemented and integrated into existing XQuery engines with little effort and that simple optimization techniques are applicable in order to get good performance.
- Linear Road Benchmark: We report the results of running the Linear Road benchmark [ACG+04] on top of an open source XQuery engine which was enhanced with the proposed extensions. The benchmark results confirm that the proposed XQuery extensions can be implemented efficiently.

The results of extending XQuery for stream processing have been published at VLDB 2007 [BFF+07]. Furthermore, the proposed window capabilities have been incorporated in the upcoming XQuery 1.1 standard [Kra08, CR08].

1.4 Structure

The remainder of this thesis is organized as follows:

- Chapter 2 provides an overview of cloud computing and XQuery as the programming model. A definition of cloud computing is given and a large number of sample systems are listed. Common features and architectures of cloud systems are discussed. Furthermore, a short overview about best practice in development of web-applications and XQuery as a universal programming language is given.
- Chapter 3 investigates how to build a database on top of cloud infrastructure services.
- Chapter 4 presents Consistency Rationing.
- Chapter 5 describes the XQuery windowing extensions for stream processing.
- Chapter 6 provides some concluding remarks and outlines future work.

Chapter 2

State of the Art

2.1 Cloud Computing

This section defines cloud computing in the context of this thesis. It further iterates over existing building blocks of cloud providers.

2.1.1 What is Cloud Computing?

The roots of cloud computing are in utility and grid computing [FZRL09]. Utility computing has been studied since the 90s, for instance with the OceanStore project at UC Berkeley. However, it probably enjoyed its biggest success as *grid computing* in the scientific community [FK04, Bje04]. Grid computing was designed for very specific purposes - mostly, to run a large number of analysis processes on scientific data. Amazon has brought the idea to the masses and strongly influenced the term cloud computing.

Cloud computing is characterized by off-site access to shared resources in an on demand fashion [Hay08, Eco08]. It refers to both, the service delivered over the Internet and the hardware and software in the data centers that provide those services [MG09]. Although not much agreement exists on an exact definition of cloud computing [VRMCL09, MG09, YBS08, FZRL09, AFG+09], most agree on the following four aspects which combined in this form are new for cloud computing:

- *On demand*: Requiring resources on demand, thereby eliminating the need for up-front investments and planning ahead for hardware provisioning.
- *Pay per use*: The offer to pay for used computing resources in utility manner (e.g. processor units per hour, storage per day), similar to an electricity bill.

- *Scalability*: The illusion of the availability of infinite computing resources on demand, eliminating the need for cloud computing users to plan ahead.
- *Maintenance and fault tolerance*: If a component fails, it is the responsibility of the service provider to replace the component. Furthermore, the systems are built in a highly reliable way, often replicated across several data centers, to minimize outages.

Although sometimes considered as a hype, the main success of cloud computing is not technology-driven but economical. Cloud computing allows companies to outsource the IT infrastructure and thus, profit from the economies of scale and the leverage effect of outsourcing [AFG+09, Eco08, PBA+08, Gar08]. Furthermore, cloud computing shifts the risk of provisioning to the cloud providers, which further decreases the cost. For this reason, it is unlikely that it is just a short-term trend.

The economical aspects also explain the new trend towards building *private clouds* [PBA+08, AFG+09]. In contrast to public clouds, typically referred to by the general term cloud computing, which offer a publicly available service, private clouds refer to smaller installations hosted inside a company offering internal services. The term cloud is used in this context, as cloud computing created a new mindset of how to develop software. That is, the focus is first on low cost, scalability, fault tolerance, and ease of maintenance and only afterwards, on performance, response time, functionality etc. Thus, the software developed in the context of cloud computing can also help to reduce the cost in local deployments. This is in particular desired, if trust or security issues prevent using public clouds.

2.1.2 Cloud Service Overview

Cloud services can be categorized into three types: Infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS) [VRMCL09, MG09, YBS08]. This is also visualised in Figure 2.1.

2.1.2.1 Infrastructure as a Service (IaaS)

IaaS is the most general form of cloud services. It refers to lower-level services such as the access to virtual machines, storage services, databases or queue services which are required to build an application environment from scratch. IaaS makes it easy to

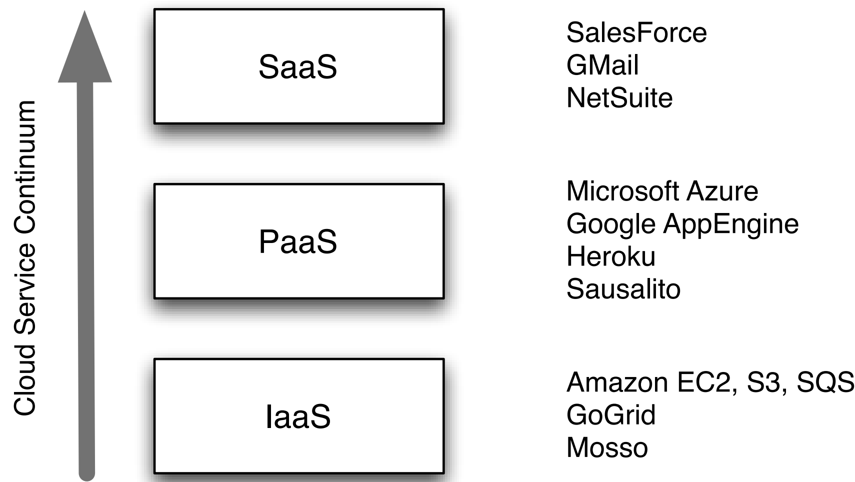

Figure 2.1: Cloud Continuum

provision and afford resources such as servers, network and storage. The services are typically billed by the amount of consumed resources. IaaS provides the biggest freedom to develop applications while at the same time requiring users to deal with lower-level details such as virtual machines, operating systems, patches etc. The three biggest IaaS providers today are Amazon, GoGrid and Mosso. The services offered by those providers are shown in table 2.1.

Hosting services: The most universal infrastructure services the providers offer are *hosting services* which allow clients to rent machines (CPU + disks) for a client-specified period of time. Examples for such services are Amazon's EC2, GoGrid Cloud Hosting and Rackspace Cloud Server. Technically, the client gets a virtual machine (VM) which is hosted on one of the provider's servers. The services are typically priced between 0.01 and 1 USD per hour and instance depending on the configuration (e.g. memory, CPU power, disk space), plus the additional cost per network traffic which is typically priced between 0.1 and 0.5 USD per GB. The VMs are able to host almost all kinds of linux and windows and, hence, allow for installing and operating almost every kind of software in the cloud. The biggest advantage of hosting services is the possibility to increase and decrease the number of virtual machines on demand. For example, if an application requires more resources over lunch-time those resources can be added in the form of additional VMs. Although most providers help spreading the load across the servers with basic load-balancing mechanisms, the programmer still needs to take care of sessions, data persistence, consistency, caches etc. Furthermore, the service

Vendor	Hosting Service	Storage Service	Other Services
Amazon [Ama09a]	Elastic Compute Cloud (EC2)	•Elastic Block Store (EBS) •Simple Storage Service (S3)	•Simple Queue Service (SQS) •CloudFront •SimpleDB
ServerPath [Off09]	GoGrid Cloud Hosting	GoGrid Cloud Storage	-
Rackspace [Rac09]	Cloud Servers	Cloud Files	-

Table 2.1: Infrastructure as a Service Providers and Products

providers do not guarantee high reliability for the VMs. That is, if a VM crashes (either caused by the user or by the provider), the complete state of the VM including the state of the local attached disk is lost.

Storage services: To persist data across the lifetime of a VM, service providers offer separate storage services, such as Amazon's Simple Storage Service or Elastic Block Store, GoGrid Cloud Storage and Mosso's Cloud Files. Again, those services are priced in a utility manner, today around 0.1 - 0.3 USD per GB per month plus in- and outgoing network traffic and quite often additional charges per write and read request. The services themselves differ in the degree of concurrency they allow (single user per entity at a time vs. multiple users per entity), consistency guarantees (strong ACID guarantees vs. eventual consistency), the degree of reliability (single data center vs. multi data center replication) and the features (simple key-value vs. queries). As the storage part plays a significant role in the application development and all the services differ significantly in the guarantees they provide, we discuss the cloud storages in more detail in Section 2.3.2.

Other services: Next to the fundamental services (i.e., hosting and storage), Amazon offers additional services to help the developers in building their applications inside the cloud. Most importantly, SimpleDB offers a simple way to store and retrieve structured data. The name, however, is misleading as SimpleDB is rather an index than a database: no notion of transaction exists, the only datatype is text and queries are restricted to simple predicates including sorting and counting but excluding joins or more complex grouping functionality. SimpleDB is described in more detail in Section 3.2.5. Next to SimpleDB, Amazon offers the Simple Queue Service (SQS). Again, the name is misleading, as SQS does not offer queues with first-in-first-out semantics. Instead,

SQS is an exactly-once message service, where a message can be put into a so called ”queue” and retrieved from it. Section 3.2.4 discusses SQS in more detail. The last service in the group of Amazon’s infrastructure offerings is CloudFront, a content delivery service. It allows developers to efficiently deliver content using a global network of edge servers. CloudFront routes a request for an object to the nearest edge location and thus, reduces latency.

Although it is possible to combine services from different providers, e.g. Amazon´s S3 with the hosting service of GoGrid, such combinations are uncommon as the cost of network traffic and latency makes them more expensive and slower. Furthermore, so far no standard API exists between the cloud providers, thus making the migration between service providers hard.

2.1.2.2 Platform as a Service (PaaS)

PaaS offerings provide a higher-level development platform to write applications, hiding the low-level details like the operating system or load balancing from the developer. PaaS platforms are often built on top of IaaS and restrict the developer to a single (limited) programming language and a pre-defined set of libraries - a so-called platform. The biggest advantage of PaaS is that the developer can completely concentrate on the business logic without having to deal with lower-level details such as patches to the operating system or firewall configurations. Furthermore, scalability is provided automatically as long as the developer follows certain guidelines. Those restrictions constitute the major downside of PaaS. This is especially true if parts of the application already exist or certain libraries are required. Again, no standards between PaaS providers exist. Therefore, changing the service provider is even harder and more cost intensive than with IaaS as the application code depends heavily on the host language and APIs.

The most prominent cloud platforms are Google’s App Engine, Microsoft’s Azure Platform and Force.com (see Table 2.2). Google’s App Engine offers to host languages in Java and Python, whereas Microsoft’s Azure Platform supports the .NET framework and PHP. Both Microsoft and Google restrict the code which is able to run on their service. For example, Google’s App Engine does not allow Java programs to spawn threads, to write data to the local file system, to make arbitrary network connections or to use JNI bindings.

Similar to IaaS, data persistence is again critical and both platforms offer dedicated services for it. Google’s DataStore accepts storing structured and semi-structured data,

Platform	Runtime Environment	Storage Service	Additional Services
Google App Engine[Goo08]	•Java* •Python* *with restrictions	DataStore	•Google accounts •Image manipulation •Mail •Memcache
Microsoft Azure Platform[Mic09]	Windows Azure: .Net languages	•SQL Azure Database •Azure Storage Service	•Access control •Service bus •Live services •Sharepoint •...
Force.com Salesforce.com [Sal09]	Force.com Platform: •Apex (based on Java) •Visualforce (for UI) •Meta programming	Force.com Database Services	•Web service API •Workflow engine •Reporting & analytics •Access control •...
28msec, Inc. [28m09]	Sausalito: XQuery	Integrated into Sausalito	XQuery function libraries: •Authentication •Atom •Mail •...

Table 2.2: Platform as a Service Providers and Products

supports transactions, and even has a simple query language called GQL. Nonetheless, DataStore does not provide the same functionality as a ”full” database. For example, the transaction support is rather rudimentary and GQL, though as a restricted language being similar to SQL, lacks the functionality for joins and complex aggregations. On the other hand, SQL Azure Database is a small MS SQL Server database and provides the features of MS SQL Server. However, it is restricted in size (at the moment 10 GB maximum). The storage services are re-visited in more detail in Section 2.3.2. Both platforms offer additional services such as authentication, message busses etc. to support the developer in programming the application. These services are beyond the scope of this thesis.

Force.com offered by the Salesforce, Inc. [Sal09] applies an approach that is slightly different from Google's App Engine and MS Azure. The Force.com platform originates from the software-as-a-service CRM product Salesforce.com and is in particular tailored to create traditional business applications [WB09]. The whole platform applies a meta-data driven development approach. Forms, reports, workflows, user access privileges, business logic, customizations up to tables and index definitions exist all as meta-data in Force.com's Universal Data Dictionary (UDD). Force.com supports two different ways of creating customized applications: the declarative way of using the provided application framework and the programmatical approach of using the web-service APIs. The programming language supported by the platform is APEX, a language derived from Java, with data manipulation operations (e.g., insert, update, delete), transaction control operations (setSavepoint, rollback), as well as the possibility to embed the Salesforce.com query (SOQL) and search (SOSL) languages. So far known, Salesforce stores the entire data of one custumer/company inside one single relational database using a de-composed multi-tenancy data layout [WB09]. As a result, the scalability is restricted to one single database.

Next to the three big platforms, many smaller startups providing platforms in different languages appear on the market like Heroku [Her09], Morph [Lab09], BungeeConnect [Bun09] and 28msec [28m09]. Most of the architectures follow models similar to the ones of Google and Microsoft. Worth mentioning is 28msec's product Sausalito as it is based on results of this thesis. Sausalito combines the application and database server in one tier and uses XQuery as the programming language. Similar to Ruby on Rails the platform provides a configuration by convention concepts and is particularly suited for all types of web applications. Sausalito applies a shared-disk architecture as presented in Chapter 3 and is entirely deployed inside Amazon's Web Service infrastructure.

2.1.2.3 Software as a Service (SaaS)

SaaS is the highest form of services delivering a special-purpose software through the internet. The biggest advantage for the customer is that no up-front investment in servers or software licensing is required and the software is completely maintained by the service provider. On the provider side, normally one single version of the app is hosted and maintained for thousands of users, lowering the cost in hosting and maintenance compared to traditional software models. The most prominent example for SaaS is Salesforce, a customer-relationship-management software. However, discussing SaaS in more detail is beyond the scope of this thesis.

2.2 Web-Based Database Applications in the Cloud

With today's choices the possibilities of building applications inside the cloud are unlimited. Different services, possibly even from different providers, can be combined to achieve the same goal. Furthermore, by using virtualized machines, any kind of service can be created inside the cloud. Although there is no agreement on how to design applications inside the cloud, certain best practice recommendations exist [Tod09].

2.2.1 Applications using PaaS

The simplest and probably fastest way to deploy an application inside the cloud is by use of a PaaS (see Section 2.1.2.2). The platform hides the complexity of the underlying infrastructure (e.g. load balancing, operating system, fault tolerance, firewalls) and provides all the services, typically through libraries, which are required to build an application (e.g., tools for building web-based user interfaces, storage, authentication, testing, etc.). As invoking services outside the platform provider is rather expensive with respect to latency, best practice is to use the services and tools from a single PaaS provider. Although PaaS offers many advantages, the biggest drawback is the incompatibility between platforms and the enforced limitations of the platform. In addition, by today, the PaaS software is proprietary and no open-source systems exist, thus preventing the use of this approach for private cloud installation.

2.2.2 Applications using IaaS

Developers requiring more freedom and/or not willing to restrict themselves to one provider, may directly use IaaS offerings. These enable a more direct access to the underlying infrastructure. The canonical form of traditional web architectures is shown in Figure 2.2. Clients connect by means of a web browser through a firewall to a web server. The web server is responsible for rendering the web-sites and interacts with a (possibly distributed) file system and an application server. The application server hosts and executes the application logic and interacts with a database, and possibly with the file system and other external systems. Although the general structure is similar for all web-based database applications, a huge variety of extensions and deployment strategies exists [CBF03]. By means of virtualized machines all of those variations are also

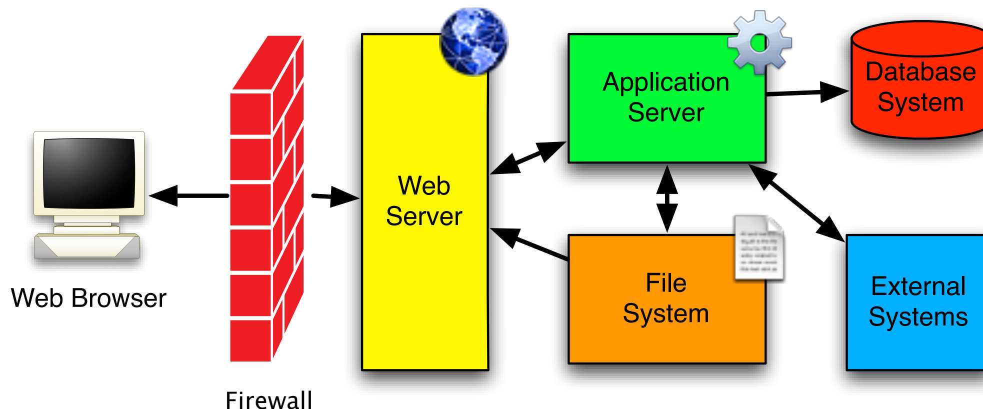

Figure 2.2: Canonical Web Architecture (modified from [Con02, CBF03])

deployable in the cloud. However, to profit from the elasticity of the cloud and the pay-as-you-go pricing model, not every architecture is equally applicable [Gar09]. For example, the quite popular monolithic (all-on-one server) architecture as epitomized by the LAMP stack (i.e., Linux, Apache, MySQL, PHP on one server) is not amenable to cloud applications. The architecture presents not only a single point of failure, but because it is limited to a single server instance, its scalability in case that traffic exceeds the capacity of the machine is limited.

The best practice deployment strategy for web-applications using IaaS is shown in Figure 2.3. The application and the web server are deployed together on one VM, typically by using a customized image. As discussed earlier, VMs normally loose all the data if they crash. Thus, the web/application tier is typically designed stateless and uses other services to persist session data (e.g. a storage service or database service). The storage service (such as Amazon's S3) is typically used for large objects like images, or videos whereas a database is used for smaller records. The elasticity of the architecture is guaranteed by a load balancer which watches the utilization of the services, initiates new instances of the web/application server and balances the load. The load balancer is either a service which might itself run on a VM, or a service offered by the cloud provider. In addition, the firewall is normally also offered as a service by the cloud provider.

Regarding the deployment of database systems in the cloud, two camps of thought exist. The first camp favours installing a full transactional (clustered) database system (e.g., MySQL, Postgres, or Oracle) in the cloud, again by using a VM service. This approach provides the comfort of a traditional database management system and makes the solution more autonomous from the cloud provider because it relies on less non-

standardized APIs. On the downside, traditional database management systems are hard to scale and not designed to run on VMs [Aba09, Kos08]. That is, the query optimizer and the storage management assume that they exclusively own the hardware and that failures are rather rare. All this does not hold inside a cloud environment. Furthermore, traditional database systems are not designed to constantly adapt to the load, which not only increases the cost, but frequently requires manual interaction in order to be able to react to the load.

The second camp prefers using a specially designed cloud storage or database service instead of a full transactional database. The underlying architecture of such a service is typically based on a key-value store designed for scalability and fault tolerance. This kind of service can either be self-deployed, typically using one of the existing open-source systems (see Section 2.3.2.2), or is offered by the cloud provider (see Section 2.3.2.1). The advantage of this approach lies in the system design for fault tolerance, scalability, and often self-management. If the service is operated by the cloud provider, it increases the outsourcing level. In this case, the cloud provider is responsible for monitoring and maintaining the storage/database service. On the downside, these services are often simpler than full transactional database systems offering neither ACID guarantees nor full SQL support. If the application requires more, it has to be built on top without much support available so far. Nevertheless, this approach is becoming more and more popular because of its simplicity and excellent scalability [Tod09].

2.3 Cloud Storage Systems

This section discusses cloud storage systems in more detail because of their fundamental role in building database application in the cloud. At the same time, it provides the foundation for Section 3. Here, we use the name cloud and database service interchangeably, as none of the database services in the cloud really offers the same comfort as a full-blown database (see Chapter 2.3.2) and, on the other hand, cloud storage services are extended with more functionality (e.g., simple query languages) making them more than a simple storage service. We would like to emphasize that we are aware that the results presented in this chapter capture merely a snapshot (June 2009) of the current state-of-the-art in utility computing. Given the success of AWS and the recent offerings by Google, it is likely that the utility computing market and its offerings will evolve quickly. Nevertheless, we believe that the techniques and trade-offs discussed in this and the subsequent chapters are fundamental and are going to

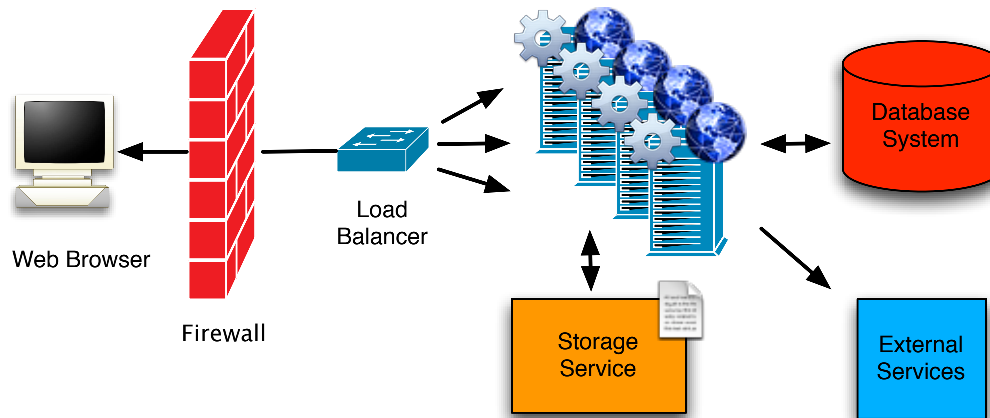

Figure 2.3: Cloud Web Architecture(modified from [Con02, CBF03])

continue to remain relevant.

2.3.1 Foundations of Cloud Storage Systems

The section explains the importance of the CAP theorem for developing cloud solutions before presenting some of the basic tools used to build cloud services.

2.3.1.1 The Importance of the CAP Theorem

To achieve high scalability at low cost, cloud services are typically highly distributed systems running on commodity hardware. Here scaling just requires adding a new off-the-shelf server. Unfortunately, the CAP theorem states that it is not possible to achieve *C*onsistency, *A*vailability and tolerance against network *P*artitioning at the same time [Bre00, GL]. In order to completely avoid network partitioning, or at least to make it extremely unlikely, single servers or servers on the same rack can be used. Both solutions do not scale and, hence, are not suited for cloud systems. Furthermore, these solutions also decrease the tolerance against other failures (e.g., power outages or over-heating). Also, to use more reliable links between the networks does not eliminate the chance of partitioning, and increases the cost significantly. Thus, network partitions are unavoidable and either consistency or availability can be achieved. As a result, a cloud service needs to position itself somewhere in the design space between consistency and availability.

2.3.1.2 Consistency Guarantees: ACID vs. BASE

Strong consistency in the context of database systems is typically defined by means of the *ACID* properties of transactions [RG02, WV02]. ACID requires that for every transaction the following attributes hold:

- *A*tomicity: Either all of the tasks of a transaction are performed or none.
- *C*onsistency: The data remains in a consistent state before the start of the transaction and after the transaction.
- *I*solation: Concurrent transactions result in a serializable order.
- *D*urability: After reporting success, the modifications of the transaction will persist.

If ACID is chosen for consistency, it emphasizes consistency while at the same time diminishing the importance of availability. Requiring ACID also implies that a pessimistic view is taken, where inconsistencies should be avoided at any price. As a consequence, to achieve ACID properties, complex protocols such as 2-phase-commit or consensus protocols like Paxos are required.

On the other extreme, where availability is more important than consistency, BASE [Bre00] is proposed as the counter-part for ACID. BASE stands for: *B*asically *A*vailable, *S*oft state, *E*ventual consistent. Where ACID is pessimistic and forces consistency at the end of every operation, BASE is optimistic and accepts inconsistency. Eventual consistency only guarantees that updates will eventually become visible to all clients and that the changes persist if the system comes to a quiescent state [SS05]. In contrast to ACID, eventual consistency is easy to achieve and makes the system highly available.

Between the two extremes BASE and ACID, a range of consistency models from the database community (e.g. the ISO isolation levels) [WV02] as well as from the distributed computing community [TS06] can be found. Most of the proposed models can lead to inconsistencies but at the same time lower the likelihood of those.

2.3.1.3 Techniques

To achieve fault tolerance and ease of maintenance, cloud services make heavy use of distributed algorithms such as Paxos and distributed hash-tables. This section presents the basic tool box to build highly reliable and scalable cloud services and thus, the

basic techniques to compare different cloud services. However, the focus here is on distributed algorithms. Standard database techniques (e.g. 2-phase-commit, 3-phase-commit etc.) are assumed to be known.

Master-Slave/Multi-Master: The most fundamental question when designing a system is the decision for a master-slave or a multi-master architecture [TS06]. In the master-slave model one device or process has the control over a resource. Every change to the resource has to be approved by the master. The master is typically elected from a group of eligible devices/processes. In the multi-master model the control of the resource is not owned by a single process; instead, every process/device can modify the resource. A protocol is responsible for propagating the data modifications to the rest of the group and resolve possible conflicts.

Distributed hash-table (DHT): A distributed hash-table provides a decentralized look-up service [SMK+01, RD01]. Within a DHT the mappings from keys to values are distributed across nodes often including some redundancy to ensure fault tolerance. The key properties of a DHT are that the disruptions caused by node joins or leaves are minimized, typically by using consistent hashing [KLL+97], and that no node requires the complete information. DHT implementations normally differ in the hash method they apply (e.g. order preserving vs. random), the load-balancing mechanism and the routing to the final mapping [UPvS09]. The common use case for DHTs is to load-balance and route data across several nodes.

Quorums: To update replicas, a quorum protocol is often used. A quorum system has three parameters: a replication factor *N*, a read quorum *R* and a write quorum *W*. A read/write request is sent to all replicas N, and each replica is typically on a separate physical machine. The read quorum R (respectively the write quorum W) determines the number of replicas that must successfully participate in a read (write) operation. That is, to successfully read (write) a value, the value has to be read (written) by R (W) numbers of replicas. Setting R + W $>$ N ensures that always the latest update is read. In this model, the latency of read/write is dictated by the slowest of the read/write replicas. For this reason, R and W are normally set to be lower than the number of replicas. Furthermore, by setting R and W accordingly the system is balanced between read and write performance. The quorums also determine the availability and durability of the system. For example, a small W increases the chance of loosing data and if fewer nodes than W or R are available, reads or writes are not possible anymore. For systems where availability and durability are more important than consistency, the concept of sloppy quorums was introduced. In the presence of failures, sloppy quorums can use

any node to persist data and these are reconsolidated later. As a consequence, sloppy quorums also do not guarantee to read the latest value if R + W is set to be bigger than N.

Vector Clocks: A vector clock is a list (i.e, vector) of (client, counter) pairs created to capture causality between different versions of the same object [Lam78, DHJ+07]. Thus, a vector clock is associated with every version of every object. Each time a client updates an object, it increments its (client, counter) pair (e.g., the own logical clock) in the vector by one. One can determine whether two versions of an object are conflicting or have a causal ordering, by examining their vector clocks. Causality of two versions is given, if every counter for every client is higher or equal to the counter of every client of the other version. Else, a branch (i.e., conflict) exists. Vector clocks are typically used to detect conflicts of concurrent updates without requiring consistency control or a centralized service [DHJ+07].

Paxos: Paxos is a consensus protocol for a network of unreliable processors [Lam98, CGR07]. At its core, Paxos requires a majority to vote on a current state - similar to the quorums explained above. However, Paxos goes further and can ensure strong consistency as it is able to reject conflicting updates. Hence, Paxos is often applied in multi-master architectures to ensure strong consistency - in contrast to simple quorum protocols, which are typically used in eventually consistent scenarios.

Gossiping protocols: Gossiping protocols, also referred to as epidemic protocols, are used to multi-cast information inside a system [DGH+87, RMH96]. They work similar to gossiping in social networks where a rumor (i.e., information) is spread from one person to another in an asynchronous fashion. Gossip protocols are especially suited for scenarios where maintaining an up-to-date view is expensive or impossible.

Merkle Trees: A Merkle tree or hash tree is a summarizing data structure, where leaves are hashes of the data blocks (e.g., pages) [Mer88]. Nodes further up in the tree are the hashes of their respective children. Hash trees allow to quickly identify if data blocks have changed and allow further to locate the changed data. Thus, hash trees are typically used to determine if replicas diverge from each other.

2.3.2 Cloud Storage Services

This section gives an overview of the available cloud storage services including open-source projects, that help to create private cloud solutions.

2.3.2.1 Commercial Storage Services

Amazon's Storage Services: The most prominent storage service is Amazon's S3 [Ama09d]. S3 is a simple key-value store. The system guarantees that data gets replicated across several data centers, allows key-range scans, but only offers *eventual consistency* guarantees. Thus, the services only promise that updates will *eventually* become visible to all clients and that changes persist. More advanced concurrency control mechanisms such as transactions are not supported. Not much is known about the implementation of Amazon's S3. Section 3.2 gives more details about the API and cost infrastructure of S3 because we use it in parts of our experiments.

Next to S3, Amazon offers the *Elastic Block Store (EBS)* [Ama09c]. In EBS, data is divided into storage volumes, and one volume can be mounted and accessed by exactly one EC2 instance at a time. In contrast to S3, EBS is only replicated within a single data center and provides session consistency [Rig08].

Internally, Amazon uses another system called Dynamo [DHJ+07]. Dynamo supports high update rates for small objects and is therefore well-suited for storing shopping carts etc. The functionality is similar to S3 but does not support range scans. Dynamo applies a multi-master architecture where every node is organized in a ring. Distributed hash tables are used to facilitate efficient look-ups and the replication and consistency protocol is based on quorums. The failure of nodes is detected by using gossiping and Merkle trees help to bring diverged replicas up-to-date. Dynamo applies sloppy quorums to achieve high availability for writes. The only consistency guarantee given by the system is eventual consistency, although the quorum configuration allows optimizing the typical behavior (e.g., read-your-write monotonicity). The architecture of Dynamo inspired many open-source projects such as Cassandra or Voldemort.

Google's Storage Service: Two Google-internal projects are known: BigTable [CDG+06] and Megastore [FKL+08]. The latter, Megastore, is most likely the system behind Google's AppEngine storage service.

Google's BigTable [CDG+06] is a distributed storage system for structured data. BigTable can be regarded as a sparse, distributed, persistent, multi-dimensional sorted map. The map is indexed by a row key, a column key, and a timestamp. No schema is imposed and no higher query interface exists. BigTable uses a single-master architecture. To reduce the load on the master, data is divided into so-called *tablet*s and one tablet is exclusively handled by one slave (called tablet server). The master is responsible for (re-)assigning the tablets to tablet servers, for monitoring, load-balancing, and certain maintenance tasks. Because BigTable clients do not rely on the master

for tablet location information, and read/write request are handled by the tablet server, most clients never communicate with the master.

Chubby [Bur06], Google's distributed lock service, is used to elect the master, to determine the group-membership of servers, and to ensure one tablet server per tablet. As its heart, Chubby uses Paxos to achieve strong consistency among the Chubby servers.

To store data persistently, BigTable relies on Google's File System (GFS) [GGL03]. GFS is a distributed file system for large data files, that are a normally only appended and rarely overwritten. Thus, BigTable stores the data on GFS in an append-only mode and does not overwrite data. GFS only provides relaxed consistency but as BigTable guarantees a single tablet server per tablet, no concurrent writes can appear, and at row-level atomicity and monotonicity are achieved. BigTable and Chubby are designed as a single data center solution. To make BigTable available across data centers an additional replication protocol exists, providing eventual consistency guarantees [Dea09].

Google's Megastore [FKL+08] is built on top of BigTable and allows to impose schemas and a simple query interface. Similar to SimpleDB, the query language is restricted and more advanced queries (e.g.,a join) are not possible. Furthermore, Megastore supports transactions with serializable guarantees inside an entity group.[1] Entity groups are entirely user-defined and have no size restriction. Still, every transaction inside an entity group is executed serially. Thus, if the number of concurrent transactions is high, the entity group becomes a bottle-neck. Again, no consistency guarantees are made between entity groups.

Yahoo's Storage Service: Two systems are known: PNUTS [CRS+08] and a scalable data platform for small applications [YSY09]. The first is similar to Google's BigTable. PNUTS applies a similar data model and also splits data horizontally into tablets. In contrast to BigTable, PNUTS is designed to be distributed across several data centers. Thus, PNUTS assigns tablets to several servers across data center boundaries. Every tablet server is the master for a set of records from the tablets. All updates to a record are redirected to the record master and are afterwards propagated to the other replicas using Yahoo's message broker (YMB). The mastership of a record can migrate between replicas depending on the usage and thus, increases the locality for writes. Furthermore, PNUTS offers an API which allows the implementation of different levels of consistency, such as eventual consistency or monotonicity.

Yahoo's second system [YSY09] consists of several smaller databases and provides

[1] Also, transactions are executed in a serial order; serializability can be violated because of the lazy updates to indexes [Ros09].

strong consistency. Nodes/machines are organized into so-called *colos* with one colo controller each. Nodes in one colo are located in the same data center, often even in the same rack. Nodes in a colo run MySQL and host one or more user databases. The colo controller is responsible for mapping the database to nodes. Furthermore, every user database is replicated inside the colo. The client only interacts with the colo controller and the controller forwards the request to the MySQL instance by using a read-once-write-all replication protocol. Thus, ACID guarantees can be provided. Additionally, user databases are replicated across colos using an asynchronous replication protocol for disaster recovery. As a consequence, major failures can lead to data loss. Another restriction imposed by the architecture is, that neither data nor transactions inside a database can span more than one machine, implying a scalability limit on the database size and usage.

Microsoft's Storage Service: Microsoft offers two services: Azure Storage Service [Cal08] and SQL Azure Database [Lee08]. Windows Azure storage consists of three sub-services: blob service, queue service, table service. The blob service is best compared to a key-value store for binary objects. The queue service provides a message service, similar to SQS, and also does not guarantee first in/first out (FIFO) behavior. The tablet service can be seen as an extension to the blob service. It allows to define tables and even supports a simple query language. Within Azure Storage Service data is replicated inside a single data center and monotonicity guarantees are provided per record but here exists no notion of transactions for several records. Little is known about the implementation, although the imposed data categorization and limitations look similar to the architecture of BigTable.

The second service offered by Microsoft is SQL Azure Database. This service is structured similar to Yahoo's platform for small applications but instead of running MySQL, Microsoft SQL Server is used [Sen08]. The service offers full transaction support, a simplified SQL-like query language (so far not all SQL Server features are exposed), but restricts the consistency to a smaller set of records (i.e., 1 or 10 GB).

2.3.2.2 Open-Source Storage Systems

This section provides an overview about existing open-source storage systems. The list is not exhaustive and many other systems such as Tokyo Cabinet [Hir09], MongoDB [10g09], and Ringo[Tuu09] exist. However, those systems were chosen as they are already more stable and/or provide some interesting features.

Cassandra: Cassandra [Fac09] was designed by Facebook to provide a scalable reverse index per user-mailbox. Cassandra tries to combine the flexible data model of BigTable with the decentralized administration and always-writable approach of Dynamo. To efficiently support the reverse index, Cassandra supports an additional three-dimensional data structure which allows the user to store and query index like structures. For replication and load-balancing, Cassandra makes use of a quorum system and a DHT similar to Dynamo.

CouchDB: CouchDB [The09b] is a JSON-based document database written in Erlang. CouchDB can do full text indexing of the stored documents and supports expressing views over the data in JavaScript. CouchDB uses peer-based asynchronous replication, which allows updating documents on any peer. When distributed edit conflicts occur, a deterministic method is used to decide on a winning revision. All other revisions are marked as conflicting. The winning revision participates in views and further provides the consistent view. However, every replica can still see the conflicting revisions and has the opportunity to resolve the conflict. The transaction mechanism of CouchDB can best be compared with snapshot isolation, where every transaction sees a consistent snapshot. But in contrast to the traditional snapshot isolation, conflicts do not result in aborts. Instead, the changes are accepted in different versions/branches which are marked as conflicting.

HBase: HBase [The09a] is a column-oriented, distributed store modeled after Google's BigTable and is part of the of the Hadoop project [The09c], an open-source MapReduce framework [DG08]. Like BigTable, HBase also relies on a distributed file system and a locking service. The file system provided together with Hadoop is called HDFS and is similar to Google's FileSystem architecture. The locking service is called ZooKeeper [Apa08] and makes use of the TCP connections to ensure consistency (in contrast to Chubby from Google which uses Paxos). However, the whole architecture of HBase, HDFS and ZooKeeper looks quite similar to Google's software stack.

Redis: Redis [Red09] is a disk-backed, in-memory key-value store written in C. The most interesting feature is the data model. Redis not only supports binary strings, integers etc., but also lists, queues and sets, as well as higher level atomic operations on them (e.g., push/pop/replacement of values in list). Redis does a master-slave replication for redundancy but no sharding; thus, all data must fit in a single system's RAM.[2] Redis maintains the whole dataset in memory and checkpoints the data from time to

[2] It is noteworthy that with an appropriate client library sharding is possible but it is not automatically provided by the system.

time to disk. Thus, the last updates may be lost in the case of bigger failures. The master-slave replication is synchronous and every change done in the master is replicated as soon as possible to the slaves. Every write has to go through the master and no further concurrency or locking mechanism is provided.

Scalaris: Scalaris [BosG09] is an in-memory key-value store written in Erlang. It uses a modified version of the Chord [SMK+01] algorithm to form a DHT, and stores the keys in lexicographical order, thus enabling range queries. Data is replicated using a quorum system. Furthermore, Scalaris supports transactions across multiple keys with ACID guarantees by using an extended version of Paxos. In a way, Scalaris combines the concept of Dynamo with Paxos and thus, offers strong consistency.

Project-Voldemort: Project-Voldemort [The09d] is yet another key-value store designed along the lines of Dynamo written in Java. However, it applies a layered architecture, which makes it possible to exchange the different components of the system. For example, it is easily possible to exchange the storage engine or the serialization method. The system seems to be in a reliable state and is in production at LinkedIn [Cor09].

2.4 XQuery as the Programming Model

This section provides an overview of existing programming models for database applications. As cloud applications are typically accessed over the internet, the standard user frontend is web-based. Thus, for the following we will concentrate on web-based database applications.

2.4.1 Programming Models Overview

The standard model for web-based applications is still a three-tier architecture, where the user interface, functional process logic and data access are developed and maintained as independent modules as demonstrated in Figure 2.4 [Con02, CBF03]. On the different layers, different languages and data representations are used. On the client-tier the standard format is HTML or XML which are then interpreted by the browser. To "program" the client the standard languages are JavaScript or Microsoft's ActionScript. The middle-tier renders the HTML or XML documents on request of the client. The

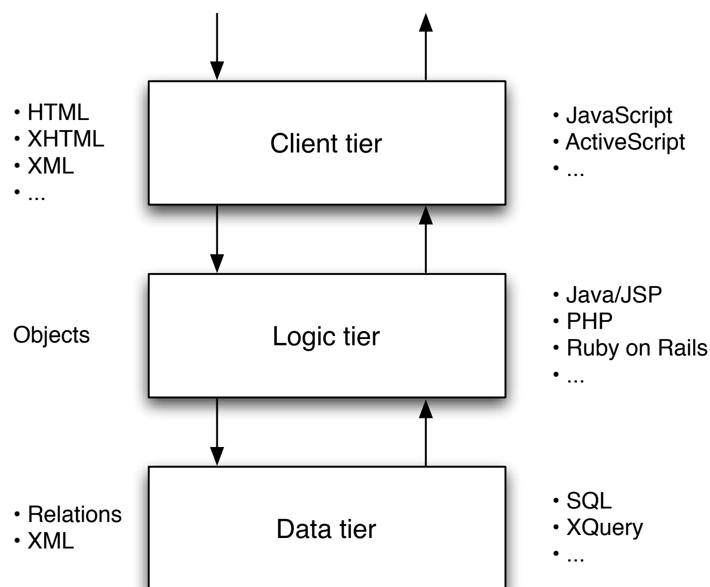

Figure 2.4: Web application layers

prominent languages for the middle-tier are Java/JSP, PHP, Ruby on Rails and Python. If the application is running inside the cloud, the middle-tier might be already a cloud service (e.g., a Platform as a Service) or hosted on a virtualized machine. The client typically interacts with the middle-tier by means of REST calls containing JSON or XML data. This data is then transformed to data types of the host language such as objects. Furthermore, the middle-tier is often a layer consisting of several services which communicate over XML/JSON as well. To persist and access data, the data tier is responsible for using a structured (e.g., relational) or a semi-structured (e.g., XML) data model. To access the data declarative languages such as SQL or XQuery are used. The communication between the middle-tier and the data tier is normally done either with remote procedure calls (RPC) or by means of XML/JSON messages.

One of the biggest problems with the traditional architecture is the need to cope with several languages and different data types (e.g., XML Schema vs. SQL types). Furthermore, transforming data from one environment to another impacts performance and adds unnecessary complexity. The approaches that have been proposed for eliminating the mismatch between the middle and data-tier fall into two general categories: (1) Enhance a host language with declarative query operators so that it no longer needs to call the data-tier; or (2) Enhance data-tier language with extensions to support application logic without relying on a middle-tier language. The approach of enhancing a programming language with a declarative language for data acces is exemplified by

languages such as Pascal/R [JS82], Microsoft LINQ [MBB06], or to a certain extent also Ruby on Rails [Gee06]. Although these extensions reduce the languages the programmer has to deal with and make the data access more integrated, the approach tends to limit opportunities for optimization, and does not address the basic mismatch between the primitive types of XML Schema and the host language.

Extending the data-tier language to make it independent of a host language as done with PL/SQL [Feu05] and IBM's SQL PL [JLB+04] has been quite a successful approach. Today, programming extensions are supported by most SQL implementations. SQL as a programming language has been used extensively in building many commercial applications including salesforce.com [Sal09] and the Oracle application suite. In general, industry experience suggests that it is easier to add a few carefully selected control flow operations to a database query language than to embed a foreign type system and persistence model into a procedural programming language.

Given the proliferation of XML data, XQuery [CR08] has recently been proposed as an alternative language which is able to run on all layers [CCF+06, FPF+09]. XQuery is a declarative language particularly developed for XML, although not restricted to XML. In the following sub-section, XQuery is explained in more detail as a general programming model for web-based database applications.

2.4.2 What is XQuery?

XQuery is a declarative programming language and builds on top of the XML Schema data types. Hence, XQuery is well-suited for parallelization and avoids the impedance mismatch between XML data types and the types of the hosting programming language. Since 2007, XQuery 1.0 is recommended by the W3C [CR08]. So far, almost fifty XQuery implementations are advertised on the W3C web pages, including implementations from all major database vendors and several open source offerings.

XQuery itself relies on several standards like XML Schema and XPath and has its own data model, which is also shared with XPath and XSLT. In the XPath and XQuery Data Model (XDM) [FMM+06], every value is an ordered sequence of zero or more items, which can be either an atomic value or a node. An atomic value is one of the atomic types defined by XML Schema or is derived from one of those. A node can be a document, an element, an attribute, a text, a comment, a processing instruction or a namespace node. So an instance of the data model may contain one or more XML documents or fragments of documents, each represented by its own tree of nodes. XQuery has several predefined functions and language constructs to define what and

how to transform one instance to another. The most commonly known feature is the FLWOR (pronounced ”flower”) expressions, which stands for the keywords ”For Let Where Order Return” and is the equivalent to ”select from where order by” in SQL.

XQuery itself is defined as a transformation from one instance of the data model to another instance, similar to a functional programming language. This also allows connecting several XQueries to each other, as every result is also a valid input. Input data from outside the XQuery engine can be inserted applying functions such as document or collection, or by referencing to the external context (prebound variables). Each of these methods then returns an XDM instance that can be processed by the XQueries.

The main XQuery specification [CR08] defines the query language. The ”XQuery Update Facilities” [CDF+09] and ”XQuery 1.0 and XPath 2.0 Full-Text” [AYBB+09] specifications add functionality for updates and full-text support to the language. Although XQuery (as the query language) is turing complete [Kep04], the pure declarative concept makes it hard to develop complete web-applications in XQuery. In [CCF+06, GRS06, BCE+08], XQuery is extended to a programming language with imperative concepts. With these extensions it becomes feasible to develop complete applications with XQuery. As XQuery is well-suited to run in the application layer as well as in the database layer, even the impedance mismatch between layers can be avoided. The different proposals to extend XQuery are now under revision of the W3C XQuery Working Group. A public working draft is already available under the name XQuery Scripting Extension 1.0 [CEF+08].

2.4.3 XQuery for Web-Based Applications

By now, XQuery is already present in all layers of a web-application. At the data-tier, XQuery is supported by all mayor database vendors and several open-source implementations exist [eXi09, Dat09, Zor09]. Thus, web-applications are already able to store data directly as XML and retrieve it using XQuery. In the middle-tier, XQuery serves several purposes. One prominent example is the transformation and routing of XML messages [DAF+03] between services. Another example is enterprise information integration [BCLW06]. A third example involves the manipulation and processing of configuration data represented in XML. At the client-tier XQuery is not so established yet, but some initial projects are available to make XQuery also useable inside the browser. For example, by default Firefox allows to execute XPath (a subset of XQuery) inside JavaScript [Moz09] or the XQuery USE ME plug-in [Zam09] enables to execute user defined XQueries to customize web-pages. However, in the middle-tier as well as

the client-tier XQuery is used inside a hosting language and therefore the impedance mismatch still exists.

XQuery as a complete programming language for the middle-tier has first been investigated inside the XL-platform in the context of web services [FGK03]. The XL platform provides a virtualized machine for XQuery code, several language extensions and a framework responsible for triggering XQuery programs based on events (i.e., a web-service message). Today, the most successful XQuery-all solution is the MarkLogic Server [Mar09]. MarkLogic Server combines an XML database with an XQuery application server. Thus, the data- and middle-tier are combined in one server. MarkLogic Server is particular well suited for document-centric applications because of its full-text search and text analyzing capabilities but not restricted to those scenarios. Finally, the XQuery in the browser project at ETH [FPF+09], investigates XQuery as an alternative to JavaScript making it possible to use XQuery at all layers and completely avoid the impedance mismatch.

Chapter 3

Building a Database on Top of Cloud Infrastructure

3.1 Motivation

Cloud computing has lead to novel solutions for storing and processing data in the cloud (as shown in Chapter 2.1). Examples are Amazon's S3, Google's BigTable, or Yahoo's PNUTS. Compared to traditional transactional database systems, the major advantage of these novel storage services is their elasticity in the face of changing conditions. For example, in order to adapt dynamically to the load, cloud providers automatically allocate and deallocate resources (i.e., computing nodes) on the fly while offering a pay-as-you-go computing model for billing. However, in terms of functionality these novel storage services are far away from what database systems offer.

One of the most important differences between cloud storage services compared to traditional transactional database systems are the provided consistency guarantees. In order to offer fault tolerance and high availability, most providers replicate the data within one or even across several data centers. Following the CAP theorem (see Section 2.3.1.1), it is not possible to jointly provide availability and strong consistency (as defined by the ACID properties) in the presence of network failures. Consequently, most cloud providers sacrifice strong consistency for availability and offer only some weaker forms of consistency (e.g., Amazons's S3 guarantees only eventual consistency). If a higher level of consistency is required it has to be built on top. Furthermore, the query processing capabilities are far from a full SQL support. Most systems offer key and key-range lookups (see Section 2.3.2). Few systems support more advanced filters and even less support a simplified query language with the ability to order and/or count.

Except for Azure SQL Data Services, no public available service offers joins or more advanced aggregates. Again, any additional functionality has to be built on top.

Although it is possible to install traditional transactional database systems in the cloud (e.g., MySQL, Oracle, IBM DB2), these solutions do not scale and adapt themselves to the load the way storage services like S3 do (see also Section 2.2.2). Furthermore, traditional database systems are not provided as a service and need to be installed, configured and monitored, thus precluding the advantages of outsourcing the database (see also Section 2.2.2). The purpose of this chapter is to explore how web-based database applications (at any scale) can be implemented on top of storage services like S3 similar to a shared-disk architecture. The chapter presents various protocols in order to store, read, and update objects and indexes using a storage service. The ultimate goal is to preserve the scalability and availability of a distributed system like S3 and achieve the same level of consistency as a database system (i.e., ACID transactions).

3.1.1 Contributions

The first contribution of this chapter is to define an (abstract) API for utility services. This API can be implemented effectively using AWS and other utility service offerings. As today all these offerings vary significantly and standardization is just at the beginning [CCI09], such a reference API is important as a basis for all further developments in this area.

The second contribution of this chapter is the development of alternative protocols in order to ensure the consistency of data stored using utility services such as S3. The most common level of consistency that the services provide is *eventual consistency* and has no support to coordinate and synchronize parallel access to the same data. To rectify the poor level of consistency, this chapter presents a number of protocols that orchestrate concurrent updates to an eventual consistent storage.

The third contribution is to study alternative client-server architectures in order to build web-based applications on top of utility services. One big question is whether the application logic should run on the client or on a server provided by the utility services (e.g., EC2). A second question concerns the implementation and use of indexes; for instance, is it beneficial to push query processing into the cloud and use services such as SimpleDB or MS Azure as opposed to implementing indexing on the client-side. Obviously, the experimental results depend strongly on the current prices and utility service offerings. Nevertheless, there are fundamental trade-offs which are not likely to change.

Finally, a fourth contribution of this chapter is to discuss alternative ways to implement the reference API for utility services based on Amazon Web Servces (AWS). Ideally, a utility service provider would directly implement the API, but as mentioned above, such standardization is not likely to happen in the foreseeable future.

3.1.2 Outline

This chapter is organized as follows: Section 3.2 describes AWS (i.e., S3, SQS, EC2 and SimpleDB) as the most prominent cloud provider in more detail and Section 3.3 defines an (abstract) API for utility services which forms the basis for the remaining chapter. Even though there are many utility service offerings today (see Section 2.3.2), AWS has the most complete and mature suite of services and was thus used for the experiments. Section 3.4 presents the proposed architecture to build web-based database applications on top of utility services. Sections 3.5 and 3.6 present the protocols to implement reads and writes on top of S3 at different levels of consistency. Section 3.7 gives implementation details of these protocols using AWS. Section 3.8 summarizes the results of experiments conducted using the TPC-W benchmark. Section 3.9 discusses related work. Section 3.10 contains conclusions and suggests possible avenues for future work.

3.2 Amazon Web Services

Among the various providers appearing on the cloud computing market place (see Chapter 2.1), Amazon with its Amazon Web Services (AWS) not only offers the most complete stack of services, but makes it especially easy to integrate different services. This section describes this most prominent representative for the wide range of today's offerings in more detail. The focus of this short survey is on Amazon's infrastructure services (e.g., storage and CPU cycles) because those form the foundation for building web-based applications. More specialized services like Amazon's payment service are beyond the scope of this work.

3.2.1 Storage Service

Amazon's storage service is named S3, which stands for **S**imple **S**torage **S**ystem. Conceptually, it is an infinite store for objects of variable size (minimum 1 Byte, maximum 5 GB). An object is a byte container which is identified by a URI. Clients can read and update S3 objects remotely using a SOAP- or REST-based interface, e.g., *get(uri)* returns an object and *put(uri, bytestream)* writes a new version of the object. A special *getIfModifiedSince(uri, timestamp)* method allows retrieving the new version of an object only if the object has changed since the specified timestamp. Furthermore, user-defined metadata (maximum 2 KB) can be associated to an object and can be read and updated independent of the rest of the object.

In S3, each object is associated to a bucket. When a user creates a new object, he specifies the bucket into which the new object should be placed. S3 provides several ways to *scan* through objects of a bucket. For instance, a user can retrieve all objects of a bucket or only those objects whose URIs match a specified prefix. Furthermore, the bucket can be the unit of security; users can grant read and write authorization to other users for entire buckets. Alternatively, access privileges can be given for an individual objects.

S3 is not for free. It costs USD 0.15 to store 1 GB of data for one month if the data is stored in the USA (see Table 3.1). Storing 1 GB in a data center in Europe costs USD 0.18. In comparison, a 160 GB disk drive from Seagate costs USD 70 today. Assuming a two-year life time of a disk drive, the cost is about USD 0.02 per GB per month (power consumption not included). Given that disk drives are never operated at 100 percent capacity and considering mirroring, the storage cost of S3 is in the same ballpark as that for regular disk drives. Therefore, using S3 as a backup device is natural. Users, however, need to be more careful when using S3 for live data because every read and write access to S3 comes with an additional cost of USD 0.01 (USD 0.012 for Europe) per 10,000 *get* requests, USD 0.01 (USD 0.012 for Europe) per 1,000 *put* requests, and USD 0.10 to USD 0.17 per GB of network bandwidth consumed (the exact rate depends on the total monthly volume of a user). For this reason, services like SmugMug use S3 as a persistent store, yet operate their own servers in order to cache the data and avoid interacting with S3 as much as possible [Mac07].

Another reason for making excessive use of caching is latency. Table 3.2 shows the response time of *get* requests and the overall bandwidth of *get* requests depending on the *page size* (defined below). These experiments were executed using a Mac (2.16 GHz Intel Core Duo with 2 GB of RAM) connected to the Internet and S3 via

	Elastic Computing Cloud	Elastic Block Store	Simple Storage Service	Simple Queuing Service
Usage US	*Standard Instance:* $0.10 per hour (small) $0.40 per hour (large) $0.80 per hour (extra large) *High CPU Instance:* $0.20 per hour (medium) $0.80 per hour (extra large)	$0.10 per 1,000,000 I/O request	$0.01 per 1,000 PUT/LIST/COPY/POST requests $0.01 per 10,000 GET and other requests No charge for delete requests	$0.01 per 10,000 req. (since WSDL 2008-01-01) $0.01 per 100 send req. No charges forall other req. (before WSDL 2008-01-01)
Storage US		$0.10 per GB/month (provisioned storage)	$0.15 per GB/month first 50TB $0.14 per GB/month next 50TB $0.13 per GB/month next 400TB $0.12 per GB/month over 500TB	
Usage EU	*Standard Instance:* $0.11 per hour (small) $0.44 per hour (large) $0.88 per hour (extra large) *High CPU Instance:* $0.22 per hour (medium) $0.88 per hour (extra large)	$0.11 per 1,000,000 I/O request	$0.012 per 1,000 PUT/LIST/COPY/POST requests $0.012 per 10,000 GET and other requests No charge for delete requests	same as US
Storage EU		$0.11 per GB/month (provisioned storage)	$0.18 per GB/month first 50TB $0.17 per GB/month next 50TB $0.16 per GB/month next 400TB $0.15 per GB/month over 500TB	
In-Transfer	$0.10 per GB			
Out-Transfer	$0.17 per GB first 10TB $0.13 per GB next 40TB $0.11 per GB next 100TB $0.10 per GB over 150TB			

Table 3.1: Amazon Web Service Infrastructure Prices (as of 20 Oct 2009)

Page Size [KB]	*Resp. Time [secs]*	*Bandwidth [KB/secs]*
10	0.14	71
100	0.45	222
1,000	2.87	348

Table 3.2: Read Response Time, Bandwidth of S3, Varying Page Size
External Client in Europe

a fast Internet connection. The results in Table 3.2 support the need for aggressive caching of S3 data; reading data from S3 takes at least 100 msecs (Column 2 of Table 3.2), which is two to three orders of magnitude longer than reading data from a local disk. Writing data to S3 (not shown in Table 3.2) takes not much more time as reading data. The results of a more comprehensive performance study of S3 are reported in [Gar07]. While latency is an issue, S3 is clearly superior to ordinary disk drives in terms of *throughput*: Virtually an infinite number of clients can use S3 concurrently and the response times shown in Table 3.2 are practically independent of the number of concurrent clients.

Column 3 of Table 3.2 shows the bandwidth that a European client gets when reading data from S3 (S3 servers located in the USA). It becomes clear that an acceptable bandwidth can only be achieved if data are read in relatively large chunks of 100 KB or more. Therefore, small objects should be clustered into *pages* with a whole page of small objects being the unit of transfer. The same technique to cluster *records* into *pages* on disk is common practice in all state-of-the-art database systems [GR94] and we adopt this technique for this study.

Amazon's service level agreement (SLA) defines an availability of the service of 99.9%. In the event of Amazon S3 not meeting the SLA, the user is eligible to receive a service credit. Amazon S3 ensures object durability by initially storing objects multiple times across data centers and additional replications in the event of device unavailability or detected bit-rot [Ama09b]. Each replica can be read and updated at any time and updates are propagated to replicas asynchronously. If a data center fails, the data can nevertheless be read and updated using a replica at a different data center; reconciliation happens later on a *last update wins* basis. This approach guarantees full read and write availability which is a crucial property for most web-based applications: No client is blocked by system failures (as long as one machine is reachable and working) or other concurrent clients and updates. To achieve this level of availability, S3 relaxes the consistency guarantees. That is, it is possible to read stale data, in case of two concurrent updates the latest wins and one might be lost.

The purpose of this work is to show how additional consistency guarantees can be provided on top of a storage service, which only guarantees eventual consistency such as Amazon S3.

3.2.2 Servers

Amazon's offering for renting computing hardware is named **E**lastic **C**omputing **C**loud, short EC2. Technically, the client gets a VM which is hosted on one of the Amazon servers. Like S3, EC2 is not for free. The cost varies from USD 0.10 to 0.80 per hour depending on the configuration. For example, the cheapest machine costs USD 0.10, the second cheapest machine costs USD 0.2 per hour, but this machine comes with 5 times the computing power of the cheapest machine. Independent of the configuration, the fee must be paid for the time the machine is leased regardless of its usage.

One interesting aspect of EC2 is that the network bandwidth from an EC2 machine to other Amazon Services is free. Nevertheless, the "per request" charges are applicable in any case. For example, if a user makes a request from an EC2 machine to S3, then the user must pay USD 0.01 per 10,000 *get* requests. As a result, it seems advantageous to have large block sizes for transfers between EC2 and S3.

EC2 is built on top of XEN, an open source hypervisor [Xen08]. It allows operating a variety of linux/unix images. Recently, Amazon also started to support Windows as operating system (no details are known about the implementation). To the best of our knowledge, EC2 does not support any kind of VM migrations in case of failures. If a machine goes down, the state is lost unless it was stored somewhere else, e.g., on S3. From a performance perspective, it is attractive to run applications on EC2 if the data is hosted on S3 because the latency of the communication between EC2 and S3 is much faster than between an external client and EC2. Table 3.3 shows the response times of EC2/S3 communication, again varying the page size. Comparing Tables 3.3 and 3.2, the performance difference becomes apparent. Nevertheless, even when used from an EC2 machine S3 is still much slower than a local disk. In addition to the reduction of latency, EC2 is also attractive to reduce the cost as internal network traffic inside the same data center is for free.

Furthermore, EC2 allows to implement missing infrastructure services such as a global transaction counter. The use of EC2 for this purpose is discussed in more detail in Section 3.7.

Page Size [KB]	*Resp. Time [secs]*	*Bandwidth [KB/secs]*
10	0.033	303
100	0.036	2778
1,000	0.111	9080

Table 3.3: Read Response Time, Bandwidth of S3, Varying Page Size
EC2 Client

3.2.3 Server Storage

Amazon designed an additional service in particular for persisting data from EC2 instances, the Elastic Block Store (EBS). EBS requires to partition the data into so-called *volumes*. A volume can then be mounted and accessed by exactly one EC2 instance at a time. Other access methods than mounting it to EBS do not exist. In contrast to S3, EBS is only replicated inside a single data center and provides session consistency. Hence, Amazon recommends to periodically checkpoint/save the data to S3. The pricing of EBS is significantly cheaper than S3: USD 0.10 per GB month and 0.10 per 1 million I/O requests. However, EBS requires provisioning the required storage per volume upfront and does not automatically scale on demand.

EBS is particularly suited for installing a traditional database on top of EC2. With EBS, the data is not lost in the event of a VM failure. However, as only one machine can access EC2 at a given time, it makes it unusable for many other scenarios. All the data access has to go through one EC2 instance, which might become a bottleneck. Furthermore, the volume storage has to be provisioned, preventing easy scale-outs.

3.2.4 Queues

Amazon also provides a queue service named **S**imple **Q**ueueing **S**ervice, short SQS. SQS allows users to manage a (virtually) infinite number of queues with (virtually) infinite capacity. Each queue is referenced by a URI and supports sending and receiving messages via an HTTP- or REST-based interface. The maximum size of a message is 8 KB. Any bytestream can be put into a message; there is no pre-defined schema. Each message is identified by a unique ID. Messages can only be deleted by a client if the client first receives the message. Another important property of SQS is, that it supports the locking of a message for up to 2 hours. Amazon changed the prices for SQS recently to USD 0.01 per 10,000 requests. Furthermore, the network bandwidth costs at least USD 0.10 per GB of data transferred, unless the requests originated from

an EC2 machine. As for S3, the cost for the consumed network bandwidth decreases the more data is transferred. USD 0.10 per GB is the minimum for heavy users.

Again, Amazon has not published any details on the implementation of SQS. It seems, however, that SQS was designed along the lines of S3. The messages of a queue are stored in a distributed and replicated way, possibly on many machines in different data centers. Clients can initiate requests at any time; they are never blocked by failures or other clients and will receive an answer (or acknowledgment) in nearly constant time. For instance, if one client has locked all messages of a queue as part of a *receive* call, then a concurrent client who initiates another *receive* call will simply get an empty set of messages as a result. Since the queues are stored in a distributed way, SQS only makes a best-effort when returning messages in a FIFO manner. So there is no guarantee that SQS returns the first message of a queue as part of a *receive* call or that SQS returns the messages in the right order. Although SQS is designed to be extremely reliable, Amazon enforces deletions of messages after 4 days in the queue and retains the right to delete unused queues.

3.2.5 Indexing Service

Amazon SimpleDB (short SDB) is a recent extension to the AWS family of services. At the time of writing this chapter, SDB was still in beta and only available for a restricted set of users. With SimpleDB, Amazon offers a service to run simple queries on structured data. In SimpleDB, each item is associated to a domain which has associated attributes. SimpleDB automatically indexes an item as it is added to a domain. Each item can have up to 256 attribute values and each attribute value can range from 1 to 1,024 bytes. SimpleDB does not require pre-defined schemas so that any item with any kinds of attributes can be inserted and indexed.

Unfortunately, SimpleDB has a number of limitations. First, SimpleDB only supports primitive bulk-loading: 25 items can be inserted at a time. Second, SimpleDB only supports *text* values (e.g., *strings*). If an application requires integers, dates, or floating point numbers, those values must be encoded properly. Third, the size limit of 1,024 bytes per attribute turns out to be too restrictive for many applications. Amazon recommends the use of S3 for anything larger. Another restriction is the expressiveness of the SimpleDB query language. The language allows for simple comparisons, negation, range expressions and queries on multiple attributes, but does not support joins or any other complex operation. Compared to a traditional database system, SimpleDB

only provides *eventual consistency* guarantees in the same way as S3: There are no guarantees *when* committed updates are propagated to all copies of the data.

As all services of AWS, SimpleDB is not free of charge. Unlike S3 and EC2, the cost is difficult to predict as it depends on the machine utilization of a request. Amazon charges USD 0.14 per machine hour consumed, USD 1.5 per GB of structured data stored per month, plus network bandwidth in the range of USD 0.10 to USD 0.17 per GB for communication with external (non-EC2) clients, depending on the consumed volume per month.

3.3 Reference Cloud API for Database Applications

This section describes an API that abstracts from the details of cloud services such as those offered by Amazon, Google, Microsoft and the other service providers presented in Chapter 2.1. All protocols and techniques presented for the development of web-based database applications in the remainder of this chapter are based on calls to this API. This API specifies not only the interfaces but also the guarantees which the different services should provide. Section 3.7 illustrates by using AWS how this API can be implemented.

3.3.1 Storage Services

The most important building block is a reliable store. The following is a list of methods that can easily be implemented using today's cloud services (e.g., AWS and Google), thereby abstracting from vendor specifics. This list can be seen as the greatest common divisor for all current cloud service providers and the minimum required in order to build stateful applications:

- ***put****(uri as string, payload as binary, metaData as key-value-pairs) as void:* Stores an item (or object) identified by the given URI. An item consists of the payload, the meta-data, and a timestamp of the last update. If the URI already exists, the item gets overwritten without warning; otherwise, a new item is created.
- ***get****(uri as string) as item:* Retrieves the item (payload, meta-data, and timestamp) associated to the URI.

- ***get-metadata****(uri as string) as key-value-pairs:* Returns the meta-data of an item.
- ***getIfModifiedSince****(uri as string, timestamp as time) as item:* Returns the item associated to the URI only if the timestamp of the item is greater than the timestamp passed as a parameter of the call.
- ***listItems****(uriPrefix as string) as items:* Lists all items whose URIs match the given URI prefix.
- ***delete****(uri as string) as void:* Deletes an item.

For brevity, we do not specify all error codes. Furthermore, we do not specify the operations for granting and removing access rights, because implementing security on cloud services is beyond the scope of this thesis.

In terms of consistency, we expect the cloud storage service to support *eventual consistency* [TS06] with time-based consolidation. That is, every update has an attached time-stamp to it and if two updates need to be consolidated, the highest time-stamp wins. However, we do not assume a global clock and several writes within a very short time-frame might be affected by clock-screws inside the system. Again, eventual consistency (with latest-update-wins conflict resolution) seems to be the standard offered by most cloud services today because it allows for scaling out and provides 100 percent availability at low cost.

3.3.2 Machine Reservation

Deploying user applications in the cloud requires deploying and running custom application code in the cloud. The current trend is to provide either a hosting platform for a specific language like Google's AppEngine [Goo08], or a facility to start a virtualized machine as done in Amazon's EC2. Hosting platforms are often restricted to a certain programming language and hide details about the way the code gets executed. Since virtualized machines are more general, we propose the API to allow to start and shutdown machines as follows:

- ***start****(machineImage as string) as machineDescription:* Starts a given image and returns the virtual machine information such as its id and a public DNS name.
- ***stop****(machineID as string) as void:* Stops the VM with the machineID

For brevity, methods for creating own images are not described.

3.3.3 Simple Queues

Queues are an important building block for creating web-based database applications in the cloud (see Section 3.5) and offered by several cloud providers (e.g., Amazon and Google) as part of their infrastructure. A queue should support the following operations (again, error codes are not specified for brevity):

- ***createQueue****(uri as string) as void:* Creates a new queue with the given URI.
- ***deleteQueue****(uri as string) as void:* Deletes a queue.
- ***listQueues****() as strings:* Returns the URIs as string of all queues.
- ***sendMessage****(uri as string, payload as binary) as integer:* Sends a message with the payload as content to the queue and returns the MessageID. The MessageID is an integer (not necessarily ordered by time).
- ***receiveMessage****(uri as string, N as integer) as message:* Retrieves N messages from the queue. If less than N messages are available, as many messages as possible are returned. A message is returned with its MessageID and payload, of course.
- ***deleteMessage****(uri as string, messageID as integer) as void:* Deletes a message (identified by MessageID) from a queue (identified by its URI).

Queues should be reliable and never lose a message.[1] Simple queues do not make any FIFO guarantees. Hence, a *receiveMessage* call may return the second message without the first message. Furthermore, Simple Queues do not make any guarantees that all messages are available at all times; that is, if there are N messages in the queue, then it is possible that a *receiveMessage* call asking for N messages returns less than N messages. Thus, Simple Queues can even operate in the presence of network partitioning and hence, can provide the highest availability (i.e., they sacrifice the consistency property for availability). All protocols that are built on top of these Simple Queues must respect possible inconsistencies. As will become clear in Sections 3.5 and 3.6, the performance of a protocol improves the better the queue conforms to the FIFO principle and the more messages the queue returns as part of a *receiveMessage* call. Some protocols require stricter guarantees; these protocols must be implemented on top of Advanced Queues.

[1] Unfortunately, SQS does not fulfil this requirement because it deletes messages after four days. Workarounds are described in Section 3.7.

3.3.4 Advanced Queues

Compared to Simple Queues, Advanced Queues provide stronger guarantees. Specifically, Advanced Queues are able to provide all successfully committed messages to a user at any moment and always return the messages in the same order.[2] That is, in the moment a send request to the queue returns successfully, the message persists and is retrievable by following receive requests. All messages are brought into a total order but an increasing order between two messages m_1 and m_2 is only guaranteed if m_2 is sent after the successful return of the send request for m_1. As a consequence of the additional guarantees, requests to these queues are expected to be more expensive and Advanced Queues may temporarily not be available because of the CAP theorem. An additional difference between Advanced Queues and Simple Queues is the availability of operators in the Advanced Queue which allows the filtering of messages. Advanced Queues provide a way to attach a user-defined key (in addition to the MessageID) to each message. This key allows further filtering of messages. Such a service is not only required for more advanced protocols, but also useful in many other scenarios. For example, Advanced Queues can be used to connect producers and consumers where the order of messages can not be ignored (e.g., in event processing, monitoring, etc.) or it can be used as reliable message framework between components. Again, the details of our implementation of Advanced Queues on top of AWS are given in Section 3.7. The API of Advanced Queues is as follows:

- ***createAdvancedQueue****(uri as string) as void:* Creates a new Advanced Queue with the given URI.
- ***deleteAdvancedQueue****(uri as string) as void:* Deletes a queue.
- ***sendMessage****(uri as string, payload as binary, key as integer) as integer:* Sends a message with the payload as the content to the queue and returns the MessageID. The MessageID is an integer that is ordered according to the arrival time of the message in the queue (messages received earlier have lower MessageID). Furthermore, Advanced Queues support the attachment of user-defined keys to messages (represented as integers) in order to carry out further filtering.
- ***receiveMessage****(uri as string, N as integer, messageIdGreaterThan as integer) as message:* Returns the top N messages whose MessageID is higher than the "messageIdGreaterThan." If less than N such messages exist, the Advanced

[2] The Advanced Queues allow to receive several messages without removing them from the queue. Thus, the queues provide even more functionality than just simple FIFO queues.

Queue returns *all* matching messages. Unlike Simple Queues, the *receiveMessage* method of Advanced Queues respects the FIFO principle.

- ***receiveMessage****(uri as string, N as integer, keyGreaterThan as integer, keyLessThan as integer) as messages:* Retrieves the top N messages whose key matches the specified key value range. In terms of FIFO and completeness guarantees, this version of the *receiveMessage* operation behaves exactly like the *receiveMessage* operation which filters on MessageID.

- ***receiveMessage****(uri as string, N as integer, olderThan in seconds) as message:* Retrieves the top N messages in the queue which are older than *olderThan* seconds. This feature is often required to recover from failure scenarios (e.g., messages were not processed in time).

- ***deleteMessage****(uri as string, messageID as integer) as void:* Deletes a message (identified by MessageID) from a queue (identified by its URI).

- ***deleteMessages****(uri as string, keyLessThan as integer, keyGreaterThan as integer) as void:* Deletes all messages whose key is in the specified range.

3.3.5 Locking Service

The locking service implements a centralized service to keep track of read- and write-locks. A client (identified by a ID) is able to acquire a *shared lock* on a resource specified by a URI. Furthermore, users can acquire *exclusive locks*. Following conventional wisdom, several different clients can hold shared locks at the same time whereas exclusive locks can only be held by a single client and preclude the granting of shared locks. Locks are only granted for a certain timeframe. Expiring locks after a specified timeout, ensures the liveliness of the system in case a client holding a lock crashes. If a lock needs to be held longer, it can be re-acquired before the timeframe expires by using the same ID. Fortunately, exclusive locks can easily be implemented on top of SQS as shown in Section 3.7, but implementing it as a dedicated cloud service, as part of the cloud infrastructure, may be more efficient than implementing it on top of basic cloud services.

The Locking Service API has the following operations:

- ***setTimeOut****(prefix as string, timeout as integer) as void:* Sets the timeout of locks for resources identified by a certain URI prefix.

- ***acquireXLock****(uri as string, Id as string, timeout as integer) as boolean:* Acquires an exclusive lock. Returns true if the lock was granted, and false otherwise.
- ***acquireSLock****(uri as string, Id as string, timeout as integer) as boolean:* Acquires a shared lock. Returns true if the lock was granted, and false otherwise.
- ***releaseLock****(uri as string, Id as string) as boolean:* Releases a lock.

In terms of reliability, it is possible that the locking service fails. It is also possible that the locking service looses its state as a consequence of such a failure. If a locking service recovers after such a failure, it refuses all *acquireXLock* and *acquireSLock* requests for the maximum timeout period, in order to guarantee that all clients who hold locks can finish their work as long as they are holding the locks.

Thus, the availability of the locking service is reduced: it is neither accessible during network partitioning and even not during fail-overs. Whereas the latter can be avoided, the service must always provide strong consistency and thus cannot always be available if network partitions are possible (Section 2.3.1.1). However, most of the protocols presented in Sections 3.5 and 3.6 assume that unavailability of the locking service corresponds to not being able to require the lock. This allows continuing the operation depending on the protocol without further drawbacks.

3.3.6 Advanced Counters

The advanced counter service is a special service designed for implementing the Generalized Snapshot Isolation protocol (Section 3.6). As its name implies, the advanced counter service implements counters which are incremented with every *increment* call. Each counter is identified by a URI. As a special feature, the advanced counter service allows *validating* counter values as well as getting the highest *validated counter value*. If not explicitly validated, counter values are automatically validated after a specified timeout period. As shown in Section 3.6.4, this validation feature is important in order to implement the commit protocol of snapshot isolation. In summary, the API contains the following operations:

- ***setTimeOut****(prefix as string, timeout as integer) as void:* Sets the timeout of all counters for a certain URI prefix.
- ***increment****(uri as string) as integer:* Increments the counter, identified by the *uri* parameter, and returns the current value of the counter (including a epic number). If no counter with that URI exists, a new counter is created and initialized to 0.

- ***validate****(uri as string, value as integer) as void:* Validates a counter value. If not called explicitly, the counter value is validated automatically considering the timeout interval after the *increment* call that created that counter value.
- ***getHighestValidatedValue****(uri as string) as integer:* Returns the highest validated counter value.

Like the locking service, all protocols must be designed keeping in mind that the advanced counter service can fail at any time. When the advanced counter service recovers, it resets all counters to 0 with a new higher epic number. However, after restart the advanced counter service refuses all requests (returns error) for the maximum timeout period of all counters. The latest epic number and the maximum timeout period of all counters must, therefore, be stored persistently, and in a recoverable way (e.g., in services like S3).

Similar to the locking service, this service also has reduced availability. However, in contrast to the locking service, unavailability can typically not be ignored and thus also reduces the availability of all systems depending on it. Hence, implementations using this service have to be aware of its availability.

3.3.7 Indexing

An indexing service is useful in situations where self-managed indexes stored on the storage service are too expensive in terms of cost and performance. This is often the case when the whole application stack should be hosted at the client level instead of on an application server, or for rarely used applications with huge amounts of data (Section 3.8 analyzes this trade-off).

Many cloud providers offer services which can be used as an index. Amazon's SimpleDB is designed as an index and supports point and range queries. Other providers often offer restricted database services (Section 2.3.2.1). For example, Microsoft's SQL Azure provides a database service in the cloud, but only up to 10GB. Google's DataStore has no data-size restrictions but synchronizes every write request inside one entity group. Although, due to their restrictions, it is often not desirable to use those services directly as a database, those services can easily be used to provide an indexing service where the restrictions do not play an important role (see Section 3.7.7).

The requirements of the indexing service are similar to the one of a storage service of subsection 3.3.1. Keys should be stored together with a payload and be retrievable by

a key. The biggest difference is, that a key does not have to be unique, that the payload is rather small and that get requests with a key and key-range are possible.

- ***insert****(key as string, payload as string) as void:* Stores a payload identified by the given key. The key is not required to be unique.
- ***get****(key as string) as set(pair(key as string,payload as string)):* Retrieves all key/value pairs associated to the key.
- ***get****(startkey as string, endkey as string) as set(pair(key as string, payload as string)):* Retrieves all key/value pairs for all keys in the range between startkey and endkey (inclusive).
- ***delete****(key as string, payload as string) as void:* Deletes the first key/value pair with matching key and payload.

In terms of consistency, we expect either *eventual consistency* or read committed, depending on the desired overall consistency guarantee and thus, the applied protocol. This service is optional for all protocols presented in this thesis. If the cloud provider does not offer an indexing service or provides a lower level of consistency than required, it is always possible to use the self-managed indexes described in Section 3.4.4.1.

3.4 Database Architecture Revisited

As mentioned in Section 3.2.1, cloud computing promises infinite scalability, availability, and throughput - from the point of view of small and medium-size companies. This section shows that many textbook techniques to implement tables, pages, B-Trees, and logging can be applied to implement a database on top of the Cloud API of Section 3.3. The purpose of this section is to highlight the commonalities between a disk-based and cloud-based database system. Sections 3.5 and 3.6 highlight the also existing differences.

3.4.1 Client-Server Architecture

Figure 3.1 shows the proposed architecture of a database implemented on top of the Cloud Storage API described in Section 3.3.1. This architecture has a great deal of

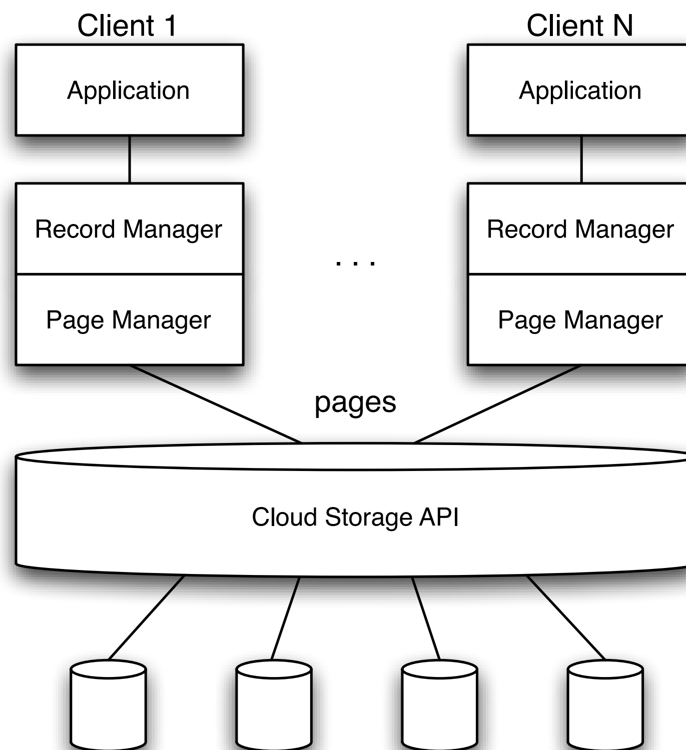

Figure 3.1: Shared-Disk Architecture

commonalities with a distributed shared-disk database system [Sto86]. The unit of transfer and buffering is a *page*. The difference is that pages are persistently stored in the cloud rather than on a disk that is directly controlled by the database system. In this architecture, pages are implemented as *items* in the Storage Service API of Section 3.3.1. Consequently, pages are identified by a URI.

As in traditional database systems, a page contains a set of records or index entries. Following the general DB terminology, we refer to records as a bytestream of variable size which can not be bigger than the page size. Records can be relational tuples or XML elements and documents. Blobs (for records bigger than the page size) can be stored directly on the storage service or using the techniques devised in [Bil92]; all these techniques are applicable in a straightforward way and that is why Blobs are not further discusses in this chapter.

Within a client, there is a stack of components supporting the application. This work focuses on the two lowest layers; i.e., the record and page managers. All other layers (e.g., the query processor) are not affected by the use of cloud services and are, for that reason, considered to be part of the application. The *page manager* coordinates

read and write requests to the Storage Service and buffers pages in local main memory or disk. The *record manager* provides a record-oriented interface, organizes records on pages, and carries out free-space management for the creation of new records. Applications interact with the record manager only, thereby using the interface described in the next subsection. Again, for the purpose of this study, a query processor (optimizer and run-time) is part of the application.[3]

Throughout this work, we use the term *client* to refer to software artifacts that retrieve pages from and write pages back to the Cloud Storage Service. It is an interesting question which parts of the *client* application stack should run on machines provided by the cloud service provider (e.g., EC2 machines), and which parts of the application stack should run on machines provided by end users (e.g., PCs, laptops, mobile phones, etc.). Indeed, it is possible to implement a web-based database architecture using this architecture without using any machines from the cloud service provider. Section 3.8 explores the performance and cost trade-offs of different client-server configurations in more detail.

A related question concerns the implementation of indexes: One option is to use SimpleDB (or related services) in order to implement indexing. An alternative is to implement B-Trees on top of the cloud services in the same way as traditional database systems implement B-Trees on top of disks. Again, the trade-offs are studied as part of performance experiments presented in Section 3.8.

Independent of whether the client application stack runs on machines of users or on, say, EC2 machines, the architecture of Figure 3.1 is designed to support thousands, if not millions of clients. As a result, all protocols must be designed in such a way that any client can fail at any time and possibly never recovers from the failure. As a result, clients are *stateless*. They may cache data from the cloud, but the worst thing that can happen if a client fails is that all the work of that client is lost.

In the remainder of this section, the record manager, page manager, implementation of (B-Tree) indexes, and logging are described in more detail. Meta-data management such as the management of a catalogue which registers all collections and indexes is not discussed in this thesis. It is straightforward to implement as all the information is stored in the database itself in the same way as traditional (relational) databases store the catalogue inside some hidden tables.

[3] Obviously, the query processor's cost model is affected by the use of a storage service, but addressing query optimization issues is beyond the scope of this thesis.

3.4.2 Record Manager

The record manager manages records (e.g., relational tuples). Each record is associated to a collection (see below). A record is composed of a key and payload data. The key uniquely identifies the record within its collection. Both key and payload data are bytestreams of arbitrary length; the only constraint is that the size of the whole record must be smaller than the page size. (As mentioned previously, the implementation of Blobs is not addressed in this thesis.)

Physically, each record is stored in exactly one page which in turn is stored as a single *item* using the Cloud Store API. Logically, each record is part of a collection (e.g., a table). In our implementation, a collection is identified by a URI. All pages in the collection use the collection's URI as a prefix. The record manager provides functions to create new records, read records, update records, and scan collections.

Create(key, payload, uri): Creates a new record into the collection identified by *uri*. If the key is not unique, then *create* returns an error (if the error can be detected immediately) or ignores the request (if the error cannot be detected within the boundaries of the transaction, see Section 3.5.3). In order to implement keys which are guaranteed to be unique in a distributed system, we use *uuids* generated by the client's hardware in our implementation. A uuid is a 128bit identifier composed of the user's MAC address, the time and some random number making a clash almost impossible [LMS05].

There are many alternative ways to implement free-space management [MCS96], and they are all applicable in this context. In our current implementation, we use a free space table which is stored together with the index information of the primary key index on the storage service (see Section 3.4.4.1). This allows using the same type of free space management for client-side B-Trees as well as for indexing services. With the help of the free space table the best fitting page for a record is estimated. It is just an estimation, as the information of the page size might be outdated by record updates (see the checkpointing and commit protocol of Section 3.5). Ideally, an assignment for a record to a page is only done once. However, as records can grow, due to variable length fields, it might be necessary to move records between pages. To avoid these moves, we reserve a certain percentage of the space for updates, similar to the *PCTFREE* parameter of Oracle's DB. In contrast to a traditional database disk layout, pages on a storage service are not required to be aligned on disk and thus, pages are not required to all have the same size. Pages are not only of flexible size but even allowed to grow beyond the page size limit. Hence, as long records grow insignificantly over time records never need to be moved.

In scenarios where records grow significantly over time, it is unavoidable to move records. Splitting a data page has a lot in common with splitting pages for a client-side B-Tree. Thus, an indexed organized table can be used, where the leaf pages are the data pages, and all the algorithms of [Lom96] can be directly adapted. With indexed organized tables, however, it is no longer possible to use an indexing service for the primary index. For the rest of the thesis we assume that records do not grow significantly as it allows to use the same presentation for a B-Tree index as well as an indexing service. However, all algorithms also work, with small adjustments, with an indexed organized table.[4]

Read*(key as uuid, uri as string):* Reads the payload data of a record given the key of the record and the URI of the collection.

Update*(key as uuid, payload as binary, uri as string):* Updates the payload information of a record. In this study, all keys are immutable. The only way to change a key of a record is to *delete* and re-*create* the record.

Delete*(key as uuid, uri as string):* Deletes a record.

Scan*(uri as string):* Scans through all records of a collection. To support scans, the record manager returns an *iterator* to the application.

In addition to the *create, read, update,* and *scan* methods, the API of the record manager supports *commit* and *abort* methods. These two methods are implemented by the page manager as described in the next section. Furthermore, the record manager exposes an interface to probe indexes (e.g., range queries): Such requests are either handled by services like SimpleDB (straightforward to implement) or by B-Tree indexes which are also implemented on top of the cloud infrastructure (Section 3.4.4).

3.4.3 Page Manager

The page manager implements a buffer pool directly on top of the Cloud Storage API. It supports reading pages from the service, pinning the pages in the buffer pool, updating the pages in the buffer pool, and marking the pages as updated. The page manager also provides a way to create new pages. All this functionality is straightforward and can be implemented just as in any other database system. Furthermore, the page manager implements the *commit* and *abort* methods. We use the term *transaction*

[4] In addition, it is also possible to adapt the tombstone technique of [Lom96] just for data pages. Thus, it is still possible to use an indexing service. For simplicity, this solution is not presented in more detail in this thesis.

for a sequence of read, update, and create requests between two commit or abort calls. It is assumed that the write set of a transaction (i.e., the set of updated and newly created pages) fits into the client's main memory or secondary storage (e.g., flash or disk). If an application commits, all the updates are propagated to the cloud via the *put* method of the Cloud Storage Service (Section 3.3.1) and all the affected pages are marked as *unmodified* in the buffer pool. How this propagation works is described in Section 3.5. If the application aborts a transaction, all pages marked *modified* or *new* are simply discarded from the buffer pool, without any interaction with the cloud service. We use the term transaction liberally in this work: Not all the protocols presented in this thesis give ACID guarantees in the DB sense. The assumption that the write set of a transaction must fit in the client's buffer pool can be relaxed by allocating additional overflow pages for this purpose using the Cloud Storage Service; discussing such protocols, however, is beyond the scope of this thesis.

The page manager keeps copies of pages from the Cloud Storage Service in the buffer pool across transactions. That is, no pages are evicted from the buffer pool as part of a *commit*. An *abort* only evicts modified and new pages.[5] Pages are refreshed in the buffer pool using a *time to live* (TTL) protocol: If an unmodified page is requested from the buffer pool after its time to live has expired, the page manager issues a *get-if-modified-since* request to the Cloud Storage API in order to get an up-to-date version, if necessary (Section 3.3.1). Furthermore, the page manager supports to force get-if-modified-since requests when retrieving a page which is useful for checkpointing the pages.

3.4.4 Indexes

As already mentioned, there are two fundamentally different ways to implement indexes. First, cloud services for indexing such as SimpleDB can be leveraged. Second, indexes can be implemented on top of the page manager. The trade-offs of the two approaches are studied in Section 3.8. This section describes the ways how both approaches can be implemented.

[5] Optionally, undo-logging (Section 3.4.5) can be used to rollback modifications instead of evicting modified pages.

3.4.4.1 B-Tree Index

B-Trees can be implemented on top of the page manager in a fairly straightforward manner. Again, the idea is to adopt existing textbook database technology as far as possible. The root and intermediate nodes of the B-Tree are stored as pages on the storage service (via the page manager) and contain *(key, uri)* pairs: *uri* refers to the appropriate page at the next lower level. The leaf pages of a primary index contain entries of the form *(key, PageURI)* and thus, refer to the pages of the according records (Section 3.4.2).[6] The leaf pages of a secondary index contain entries of the form *(search key, record key)*. So probing a secondary index involves navigating through the secondary index in order to retrieve the *keys* of the matching records and then navigating through the primary index in order to retrieve the records with the payload data.

As mentioned in Section 3.4.1, holding locks must be avoided as long as possible in a scalable distributed architecture. Therefore, we propose to use B-link trees [LY81] and their use in a distributed system as proposed by [Lom96] in order to allow concurrent reads and writes (in particular splits), rather than the more mainstream *lock-coupling protocol* [BS77]. That is, each node of the B-Tree contains a pointer (i.e., URI) to its right sibling at the same level. At the leaf level, this chaining can naturally be exploited in order to implement scans through the whole collection or through large key ranges.

Every B-Tree is identified by the URI of its index information, which contains the URI of the root page of the index. This guarantees that the root page is always accessible through the same URI, even when the root page splits. A collection in our implementation is identified by the URI of its primary index information. All URIs to the index information (primary or secondary) are stored persistently as meta-data in the system's catalogue on the cloud service (Section 3.4.1).

3.4.4.2 Cloud Index

Using the cloud API defined in Section 3.3.7 implementing an secondary index is straightforward. Entries are the same as for the B-Tree implementation and have the form of *(search key, record key)*. However, as the API defines the search key as well as the record key as string, numerical values have to be encoded. Possibilities on how to encode numerical search keys are given in [Ama09e]. As the index is provided as

[6] In order to save space, we store only a page ID and not the complete page URI. The page ID together with the collection URI forms the page URI. This optimization is done at several places, and not further mentioned.

a service, the index information is not required to find the root page of the tree. However, for primary indexes the index information is still required to store the free-space table. Except for that, the entries have the same form as for the B-Tree and can be implemented in the same manner as the secondary indexes.

3.4.5 Logging

The protocols described in Sections 3.5 and 3.6 make extensive use of *redo* log records. In all these protocols, it is assumed that the log records are *idempotent*; that is, applying a log record twice or more often has the same effect as applying the log record only once. Again, there is no need to reinvent the wheel and textbook log records as well as logging techniques are appropriate [GR94]. If not stated otherwise, we use the following (simple) redo log records in our implementation:

- *(insert, key, payload):* An *insert* log record describes the creation of a new record; such a log record is always associated to a collection (more precisely to the primary index which organizes the collection) or to a secondary index. If such an *insert* log record is associated to a collection, then the *key* represents the key value of the new record and the *payload* contains the other data of the record. If the *insert* log record is associated to a secondary index, then the *key* is the value of the search key of that secondary index (possibly composite) and the payload is the primary key value of the referenced record.
- *(delete, key):* A *delete* log record is also associated either to a collection (i.e., primary index) or to a secondary index.
- *(update, key, afterimage):* An *update* log record must be associated to a data page; i.e., a leaf node of a primary index of a collection. An update log record contains the new state (i.e., after image) of the referenced record. Logical logging is studied in the context of Consistency Rationing in Chapter 4.5.2. Other optimized logging techniques are not studied in this work for simplicity; they can be applied to cloud databases in the same way as to any other database system. Entries in a secondary index are updated by deleting and re-inserting these entries.

By nature, all these log records are idempotent: In all three cases, it can be deduced from the database whether the updates described by the log record have already been applied. However, with such simple *update* log records, it is possible that the same

update is applied twice if another update overwrote the first update before the second update. This property can result in indeterminisms as shown in Section 3.5.3. In order to avoid these indeterminisms, more sophisticated logging can be used such as the log records used in Section 3.6.2.

If an operation involves updates to a record and updates to one or several secondary indexes, then separate log records are created by the record manager to log the updates in the collection and at the secondary indexes. Again, implementing this functionality in the record manager is straightforward and not different to any textbook database system.

Most protocols studied in this work involve *redo* logging only. Only the protocol sketched in Section 3.6.4 requires *undo* logging. *Undo* logging is also straightforward to implement by keeping the *before image* in addition to the *after image* in *update* log records, and by keeping the last version of the record in *delete* log records.

3.4.6 Security

Several security concerns exist in the open architecture shown in Figure 3.1. The most important ones are the access control to the data and the trust issue with the cloud provider. In the following, we briefly describe possible solutions for both security concerns. However, to cover the security aspect in more detail is beyond of the scope of this thesis and remains future work.

Basic access control can be implemented using the access control functionality most cloud services have already in place. For example, inside Amazon S3 users can give other users read and/or write privileges on the level of buckets or individual objects inside the bucket. This coarse-grained access control can be used for the fundamental protection of the data. More fine-grained security and flexible authorization using, e.g., SQL views have to be implemented on top of the storage provider. Depending on the trust to the user and the application stack, several security scenarios are applicable to implement more fine-grained control. If the protocols cannot be compromised, one solution is to assign a curator for each collection. In this scenario, all updates must be approved by the curator before they become visible to other clients. While the updates wait for approval, clients continue to see the old version of the data. This allows for a more central way of security control and is well-suited for the checkpointing protocol presented in the next section. Additionally, the curator can ensure the integrity constraints inside the database. However, a curator only works for the simple protocols which do not provide serializability guarantees. For more advanced protocols or in

case protocols can be corrupted, the P2P literature already proposes various solutions [PETCR06, BT04, BLP05], which are also adaptable to the here presented architecture.

The trust issue between the user or application and the service provider can be solved by encrypting all the data. Furthermore, in order to give different sets of clients access to different sets of pages, different encryption keys can be used. In this case, the header of the page indicates which key must be used to decrypt and re-encrypt the page in the event of updates, and this key must be shared by all the clients or application servers who may access that page.

3.5 Basic Commit Protocols

The previous section showed that a database implemented on top of cloud computing services can have a great deal of commonalities with a traditional textbook database system implemented on top of disks. This section addresses one particular issue which arises when concurrent clients commit updates to records stored on the same page. If no care is taken, then the updates of one client are overwritten by the other client, even if the two clients update different records. The reason is that the unit of transfer between clients and the Cloud Storage Service in the architecture of Figure 3.1 is a page rather than an individual record. This issue does not arise in a (shared-disk) database system because the database system coordinates updates to the disk(s); however, this coordination limits the scalability (number of nodes/clients) of a shared-disk database. It also does not arise in today's conventional use of cloud storage services: When storing large objects (e.g., multi-media objects), the unit of transfer can be the object without a problem; for small records, clustering several records into pages is mandatory in order to get acceptable performance (Section 3.2.1). Obviously, if two concurrent clients update the same record, then the last updater wins. Table 3.4 provides on overview of the different protocols described in this and the next section together with the levels of consistency and availability they provide.

The protocols devised in this section are applicable to all architectural variants described in Section 3.4; i.e., independent of which parts of the client application stack are executed on end-users' machines (e.g., laptops) and which parts are executed on cloud machines (e.g., EC2 machines). Again, the term *client* is used in order to abstract from these different architectures. The protocols are also applicable to the two different ways of implementing indexes (SimpleDB vs. client-side B-Trees). The protocols designed in this section preserve the main features of cloud computing: clients can fail anytime,

Protocol	Guarantees[1]	Required Services[2]	Availability
3.5 Basic	•Eventual consistent on record level •Possible loss of updates in the case of failures during the commit	•Storage Service •Simple Queues	High
3.6.1 Atomicity	•Atomicity[3] (either all or none of the updates will persist) •No guarantee on when a transaction becomes fully visible	•Storage Service •Simple Queues	High
3.6.2 Monotonicity	Various monotonicity guarantees (Monotonic Reads, Monotonic Writes, Read your Writes, Write follows Read, Session Consistency)	•Storage Service •Simple Queues	For single clients reduced
3.6.3 Advanced Atomicity	•Atomicity[3] •Guaranteed time until a transaction becomes fully visible	•Storage Service •Advanced Queues •Locking Service	System wide reduced
3.6.4 Generalized Snapshot Isolation	•Snapshot Isolation Guarantees •Depends on the advanced atomicity protocol	•Storage Service •Advanced Queues •Locking Service •Advanced Counters	System wide reduced
3.6.5 2-Phase-Locking	•Serializable •Depends on the advanced atomicity protocol	•Storage Service •Advanced Queues •Locking Service	System wide reduced

[1]All transactions provide *read-committed* guarantees. That is, only the changes of committed transactions are visible to other clients. Still, the guarantee does not correspond to the definition of the ISO isolation levels, because of the missing or dissent atomicity guarantees. [2]Lists only the services which are required to execute a transaction, not to checkpoint updates. Further, the indexing service and the machine reservation service are orthogonal to the protocol and also not listed. [3]The guarantee does not apply for the visibility of updates. Hence, it is possible for a concurrent transaction to observe partially carried out transactions.

Table 3.4: Consistency Protocols

clients can read and write data at constant time, clients are never blocked by concurrent clients, and distributed web-based applications can be built on top of the cloud without the need to build or administrate any additional infrastructure. Again, the price to pay for these features is reduced consistency: In theory, it might take an undetermined amount of time before the updates of one client become visible to other clients. In practice, the time can be controlled, thereby increasing the cost (in $) of running an application for increased consistency (i.e., a reduced propagation time).

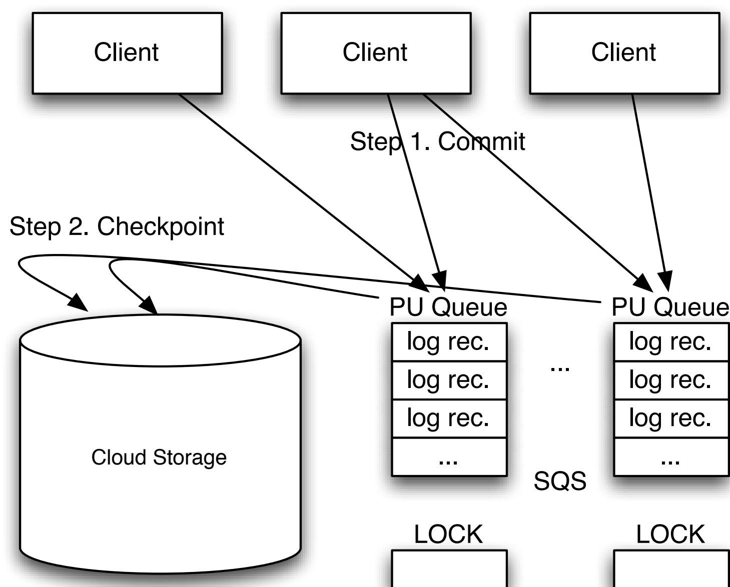

Figure 3.2: Basic Commit Protocol

3.5.1 Overview

Figure 3.2 demonstrates the basic idea of how clients commit updates. The protocol is carried out in two steps:

- In the first step, the client generates log records for all the updates that are committed as part of the transaction and sends them to the queues.
- In the second step, the log records are applied to the pages using our Store API. We call this step checkpointing.[7]

This protocol is extremely simple, but it serves the purpose. Assuming that the queue service is virtually always available and that sending messages to the queues never blocks, the first step can be carried out in constant time (assuming a constant or bounded number of messages which must be sent per commit). The second step, checkpointing, involves synchronization (Section 3.5.3), but this step can be carried out asynchronously and outside the execution of a client application. That is, end users are never blocked by the checkpointing process. As a result, virtually 100 percent

[7] We use the word *checkpointing* for this activity because it applies updates from one storage media (queues) to the persistent storage. There are, however, important differences to traditional DBMS checkpointing. Most importantly, checkpointing in traditional DBMSes is carried out in order to reduce the recovery time after failure. Here, checkpointing is carried out in order to make updates visible.

read, write, and commit availability is achieved, independent of the activity of concurrent clients and failures of other clients.

The protocol of Figure 3.2 is also resilient to failures. If a client crashes during *commit*, then the client resends all log records when it restarts. In this case, it is possible that the client sends some log records twice and as a result these log records may be applied twice. However, applying log records twice is not a problem because the log records are idempotent (Section 3.4.5). If a client crashes during a commit, it is also possible that the client never comes back or loses uncommitted log records. In this case, some log records of the commit have been applied (before the failure) and some log records of the commit will never be applied, thereby violating atomicity. Indeed, the basic commit protocol of Figure 3.2 does not guarantee atomicity. Atomicity, however, can be implemented on top of this protocol as shown in Section 3.6.1.

In summary, the protocol of Figure 3.2 preserves all the features of cloud computing. Unfortunately, it does not help with regard to consistency. That is, the time before an update of one client becomes visible to other clients is unbounded in theory. The only guarantee that can be given is that *eventually* all updates will become visible to everybody and that all updates are durable. This property is known as *eventual consistency* [TS06]. Thus, no serializability guarantees are provided for transactions and the consistency level is best comparable to read committed without atomicity guarantees (e.g., a transaction might be not completely applied due to failures). Protocols providing more guarantees are presented in the subsequent sections (see also Table 3.4). In practice, the freshness of data seen by clients can be controlled by setting the checkpoint interval (Section 3.5.7) and the TTL value at each client's cache (Section 3.4.1). Setting the checkpoint interval and TTL values to lower values will increase the freshness of data, but it will also increase the ($) cost per transaction (see experiments in [BFG$^+$08]). Another way to increase the freshness of data (at increased cost) is to allow clients to receive log records directly from the queue, before they have been applied to the persistent storage as part of a checkpoint.

The remainder of this section describes the details of the basic commit protocol of Figure 3.2; i.e., committing log records to queues (Step 1) and checkpointing (Step 2).

3.5.2 PU Queues

Figure 3.2 shows that clients propagate their log records to so-called *PU queues* (i.e., **P**ending **U**pdate queues). These PU queues are implemented as Simple Queues using the API defined in Section 3.3.3. In theory, it would be sufficient to have a single PU

queue for the whole system. However, it is better to have several PU queues because that allows multiple clients to carry out checkpoints concurrently: As shown in Section 3.5.3, a PU queue can only be checkpointed by a single client at the same time. Specifically, we propose to establish PU queues for the following structures:

- Each index (primary and secondary) has one PU queue associated to it. The PU queue of a index is created when the index is created and its URI is derived from the URI of the index. The URI of the primary index is the same as the collection URI. All *insert* and *delete* log records are submitted to the PU queues of index.
- One PU queue is associated to each data page containing the records of a collection. *Update* and *delete* log records are submitted to the PU queues of data pages. The URIs of these PU queues are derived from the corresponding URIs of the data pages.

The delete records are send to both as they require to update the primary index as well as the page. Inserts cannot be sent to both queues immediately, as first a free page needs to be determined for the insert. Hence, inserts are first send to the collection queue. During the checkpointing of a collection, those new records get a page assigned and are afterwards forwarded to the according page. Finally, the page checkpointing applies then the records to the according pages (see Section 3.5.4).

Algorithm 1 Basic Commit Protocol

1: **for all** modified records R **do**
2:    $C \leftarrow$ collection of R
3:    $P \leftarrow$ page of R
4:    **if** R.P $\neq$ null **then**
5:       sendMessage(P.Uri, R.Log)
6:    **end if**
7:    **if** R.P $=$ null $\vee$ R is marked as deleted **then**
8:       sendMessage(C.Uri, R.Log);
9:    **end if**
10:    **for all** C.SecondaryIndexes S **do**
11:       **if** R has a modified value for S **then**
12:          sendMessage(S.Uri, pair(R.Key, V));
13:       **end if**
14:    **end for**
15: **end for**

The pseudo code of the commit routine is given in Algorithm 1. In that algorithm, $R.Log$ refers to the log record generated to capture all updates on Record R. $C.Uri$ refers to the URI of the collection to which Record R belongs to; this URI coincides with the URI of the root page of the collections primary index. $P.Uri$ is the URI of the page in which a record resides. $R.key$ is the key value of Record R. If a record comes without any page assigned, it is a new record and hence an insert. For efficiency, it is possible for the client to pack several log records into a single message, thereby exploiting the maximum message size of the Cloud Queue Service. This trick is also applicable in all the following algorithms and not further shown to facilitate the presentation.

3.5.3 Checkpoint Protocol for Data Pages

Checkpoints can be carried out at any time and by any node (or client) of the system. A checkpoint strategy determines when and by whom a checkpoint is carried out (Section 3.5.7). This section describes how a checkpoint of log records is executed on data pages. This applies updates to records which already have an assigned data page. The next section describes how checkpoints of *insert* and *delete* log records on the collection are carried out.

The input of a checkpoint is a PU queue. The most important challenge when carrying out a checkpoint is to make sure that nobody else is concurrently carrying out a checkpoint on the same PU queue. For instance, if two clients carry out a checkpoint concurrently using the same PU queue, some updates (i.e., log records) might be lost because it is unlikely that both clients will read exactly the same set of log records from the PU queue (Section 3.3.3). In order to synchronize checkpoints, the Locking Service (Section 3.3.5) is used. When a client (or any other authority) attempts to do a checkpoint on a PU queue, it tries first to acquire an exclusive lock for the PU queue URI. If that lock request is granted, then the client knows that nobody else is concurrently applying a checkpoint on that PU queue and proceeds to carry out the checkpoint. If it is not granted (or if the service is unavailable), then the client assumes that a concurrent client is carrying out a checkpoint and simply terminates the routine (no action required for this client). It has to be noted, that the lock does not prevent concurrent transactions to attach messages to the queue (see Algorithm 1). Thus, carrying out a checkpoint does not block any transaction.

Per definition, the exclusive lock is only granted for a specific timeframe. During this time period the client must have completed the checkpoint; if the client is not finished within that timeout period, the client aborts checkpoint processing and propagates no

changes. Setting the timeout for holding the exclusive lock during checkpointing a data page is critical. Setting the value too low and the maximum number of log records per checkpoint to high might result in starvation because no checkpoint will ever be completed. Furthermore, the timeout must be set long enough to give the Cloud Storage Service enough time to propagate all updates to a data page to all replicas of that data page. On the other hand, a short timeout enables frequent checkpoints and, thus, fresher data. For the experiments reported in Section 3.8, a timeout of 20 seconds was used.

Algorithm 2 Page Checkpoint Protocol

Require: page P, $PropPeriod$, $X \leftarrow$ maximum number of log records per checkpoint

1: **if** acquireXLock(P.Uri) **then**
2: $StartTime \leftarrow$ CurrentTime()
3: $V \leftarrow$ get-if-modified-since(P.Uri, P.Timestamp)
4: **if** $V \neq Null$ **then**
5: $P \leftarrow V$
6: **end if**
7: $M \leftarrow$ receiveMessage(P.Uri, X)
8: **for all** messages m in M **do**
9: apply m to P
10: **end for**
11: **if** CurrentTime() - $StartTime < LockTimeOut - PropPeriod$ **then**
12: put(P.Uri, P.Data)
13: **if** CurrentTime() - $StartTime < LockTimeOut - PropPeriod$ **then**
14: **for all** messages m in M **do**
15: deleteMessage(P.Uri, m.Id)
16: **end for**
17: **end if**
18: **end if**
19: **end if**

The complete checkpoint algorithm is given in Algorithm 2. The algorithm first gets an exclusive lock (Line 1) and then re-reads the data page, if necessary (Lines 2-6). In order to find out whether the current version of the data page in the client's cache is fresh, each data page contains the timestamp of the last checkpoint as part of its page header. After that, X update log records are read from the PU Queue (Line 7). X is a parameter of this algorithm and depends on the timeout period set for holding the exclusive lock (the longer, the more log records can be applied) and the checkpoint interval (Section

3.5.7, the longer, the more log records are available). In our experiments, X was set to 1000 messages at the beginning. Afterwards X is automatically adjusted depending on the queue performance. After reading the log records from the PU queue, the log records are applied to the local copy of the page in the client's cache (Lines 8-10) and the modified page is written back to the Cloud Storage Service (Line 12). Writing back the page to the Cloud Storage Service involves propagating the new version of the data page to all replicas of the data page inside the cloud. This propagation must happen within the timeout period of the exclusive lock in order to avoid inconsistencies created by concurrent checkpointing clients. Unfortunately, Cloud Storage providers do not give any guarantees how long this propagation takes, but, for instance, Amazon claims that five seconds is a safe value for S3 (less than one second is the norm). To simplify the presentation we assume here, that this *PropPeriod* is always sufficient. Accordingly, Line 11 considers a *PropPeriod* parameter which is set to five seconds in all experiments reported in Section 3.8. Section 3.5.6 explains, how the protocols can be extended if this assumption does not hold. The time is checked again after writing the page before deleting the messages (Lines 13) to ensure that no second checkpoint started before the page was written and propagated to S3. Finally, at the end of Algorithm 2, the log records are deleted from the PU queue (Lines 14-16).

In Line 9 of Algorithm 2, it is possible that the data page must be split because the records have grown. In the implementation with index-organized tables, splitting data pages is the same as splitting index nodes and carried out along the lines of [Lom96] so that clients which read a page are not blocked while the page is split. In Line 12, the *put* method to the Cloud Storage Service is considered to be atomic. The exclusive lock obtained in Line 1 need not be released explicitly because it becomes available automatically after the timeout period.

This protocol to propagate updates from a PU queue to the Cloud Storage Service is safe because the client can fail at any point in time without causing any damage. If the client fails before Line 12, then no damage is done because neither the PU queue nor the data page have changed. If the client fails after Line 12 and before the deletion of all log records from the PU queue, then it is possible that some log records are applied twice. Again, no damage is caused in this case because the log records are idempotent (Section 3.4.5). In this case, indeterminisms can appear if the PU queue contains several *update* log records that affect the same *key*. As part of a subsequent checkpoint, these log records may be applied in a different order (i.e., Simple Queues do not guarantee the order or availability of messages) so that two different versions of the page may become visible to clients, even though no other updates have been initiated in the meantime. These indeterminisms can be avoided by using the extended

logging mechanism for *monotonic writes* described in Section 3.6.2 as opposed to the simple log records described in Section 3.4.5.

3.5.4 Checkpoint Protocol for Collections

The previous section has shown, how updates to a data page are checkpointed. This section describes how inserts and deletes to a collection are handled. The basic idea of a collection checkpoint is to update the primary index with the deletion and inserts to the collection and forward the request to the according pages. As mentioned earlier, our implementation supports free-space management by using a simple table as well as an index-organized table. Here, we only present the free-space table approach because it does not restrict the primary index to a client-side index (see also Section 3.4.2).

Algorithm 3 sketches the collection checkpoint protocol. The algorithm first gets an exclusive lock (Line 1) to prevent concurrent checkpoints. A collection checkpoint often requires a longer and more variable time than a page checkpoint due to the updates of the primary index. Thus, we use longer timeout periods (i.e., 60s in our experiments) and actively release the locks because typically the required time is much shorter. After acquiring the lock, the index information containing the free-space table is re-read from the storage service (Line 3) and the primary index is outdated (Line 4). Outdating the index pages forces get-if-modified-since requests for every index page and thus, brings the index up-to-date. Afterwards, X update log records are read from the PU Queue (Line 5). X again depends on the timeout period of the exclusive lock and can even be adjusted during run-time. Line 7-19 iterates through all received messages and uses the free-space table (Line 11) to find a page for inserts or informs the free-space table about deletions (Line 17). All pages receiving inserts are stored in the Set P (Line 15). Again, note that the free-space table just estimates free pages and is not 100% exact as updates on records are not considered. Hence, the table has to be corrected from time-to-time by scanning through all data tables (not shown here). The biggest advantage of using a free-space table instead of an index-organized table is, that the likelihood increases that all records are inserted into the same page (recall, RIDs are random uuids and not continuously increasing) and thus, less data pages need to be updated.

If the lock has not expired yet, including same padding to propagate the changes (Line 20), all changes are applied to the index (Line 21). In case an indexing service is used, the request is simply forwarded. If a client-side index is used, updating the index involves navigating through the tree and possibly merging and splitting pages. Again, the *PropPeriod* is used to guarantee that there is enough time to update the index and

Algorithm 3 Collection Checkpoint Protocol

Require: $CollectionURI$, $PropPeriod$, $X \leftarrow$ maximum number of log records per checkpoint, $ClientID$

1: **if** acquireXLock($CollectionURI$, $ClientID$) **then**
2: $StartTime \leftarrow$ currentTime()
3: $IndexInfo \leftarrow$ get($CollectionURI$)
4: PageManager.outdate($IndexInfo$, X)
5: $M \leftarrow$ receiveMessage($CollectionURI$, X)
6: UniqueMap $P \leftarrow \emptyset$
7: **for all** messages m in M **do**
8: **if** m.Type = INSERT **then**
9: m.PageURI $\leftarrow$ findKey($IndexInfo$, m.RID)
10: **if** m.PageURI = $Null$ **then**
11: m.PageURI $\leftarrow$ $IndexInfo$.getFreePage(size(m.log))
12: **else**
13: $IndexInfo$.correctFreePage(m.PageURI, size(m.log)
14: **end if**
15: P.add(m.PageURI)
16: **else**
17: $IndexInfo$.freePage(m.PageURI, size(m.log))
18: **end if**
19: **end for**
20: **if** CurrentTime() - $StartTime < LockTimeOut - PropPeriod$ **then**
21: updateIndex($IndexInfo$, M)
22: put($IndexInfo$)
23: $CommitTime \leftarrow$ CurrentTime()
24: **if** CurrentTime() - $StartTime < LockTimeOut - PropPeriod$ **then**
25: **for all** messages m in M **do**
26: **if** m.Type = INSERT **then**
27: sendMessage(m.PageURI, M)
28: **end if**
29: deleteMessage($IndexURI$, m.MessageID)
30: **end for**
31: **for all** Pages p in P **do**
32: do page-checkpoint for p
33: **end for**
34: wait($CommitTime$ + PropPeriod - CurrentTime())
35: releaseLock($CollectionURI$, $ClientID$)
36: **end if**
37: **end if**
38: **end if**

propagate all changes (including the changes to the index). As several pages are involved, this typically requires more time than a simple page propagation and was set depending on the location of the client (e.g., EC2 or end-user machine). Afterwards, the inserts are forwarded to the according pages (Line 25-27) and all messages are deleted from the collection queue (Line 28). Finally, a checkpoint is tried on all affected pages (Line 30-32) and the lock is released (Line 34) after the propagation time has expired (Line 33).

The protocol is safe because the client can fail at any point in time without causing any damage. If the client fails before Line 21 nothing has changed inside the queues or on the storage service and the failure is safe. If the client fails after Line 21, no damage is caused as the index has already been updated in Line 21 and the same inserts and deletes will be re-read from the queue by another checkpoint. As every insert is checked against the index (Line 9) before the free-space table is used, all inserts will be placed on exactly the same page. If the client crashes during Line 21, the behaviour is similar as again, all the updates which have already been made can be found. Failures after Line 22 can however increase the coherence between the actual available free-space and the information in the free-space table (i.e., deletes are applied more than once). That does not harm, as the free-space table is only an estimation and corrected from time-to-time.

Algorithm 2 is just a sketch of the protocol, and many optimizations are possible to increase the performance and robustness. Most importantly, *updateIndex* can be a time-consuming operation and is hard to predict. Thus, it is often useful to sort the messages by their key, and process the checkpoint in several batches. It is even possible to use the index information to form the batches (similar to the algorithm sketch of Section 3.5.5). Furthermore, the divergence of the free-space information can be decreased by additional free-space correction messages to the collection queue. Again, batching of messages is an easy and effective optimization. Finally, checkpointing the collection queue can be done in parallel. The simplest solution is, to have one or more collection queues partitioned by the record key, each with its own set of pages and a corresponding free-space table. Because keys are constructed by uuids it already results in an almost uniform distribution. The index can still be shared or also be partitioned. The simple parallelization scheme introduces additional overhead when higher levels of consistency is required (e.g., for snapshot isolation, several queues have to be checked simultaneously and messages additionally need to be sorted). Thus, index-organized tables or Advanced Queues can also be explored to increase the level of parallelism for inserts and deletes. To discuss those in detail is beyond the scope of this thesis.

3.5.5 Checkpoint Protocol for Client-Side B-Trees

As mentioned in Section 3.5.2, checkpointing is only needed for client-side (B-Tree) indexes. No checkpointing is required if indexing is implemented using services like SimpleDB. For client-side B-Tree indexes, there is one PU queue associated to each B-Tree: This PU queue contains log records for the respective B-Tree. Primary and secondary indexes are checkpointed in the same way. Only if the index is used for an index-organized table, the leaf nodes (i.e., data pages) have to be treated separately. Checkpointing a client-side B-Tree is more complicated than checkpointing a data page because several (B-Tree) pages are involved in a checkpoint and because splitting and deleting pages is a frequent procedure. To allow concurrent reads during updates we adopted the techniques of [Lom96]. In contrast to [Lom96], the navigation to a leaf-page always needs to be done by the client and cannot be forwarded to the server holding the data page. Furthermore, as checkpointing enables increased batch-processing, we try to update as many items as possible at once to the index. Hence, the basic protocol for checkpointing B-Tree pages can be summarized in the following protocol sketch:

1. Obtain the token from the LOCK queue (same as Step 1, Section 3.5.3).
2. Receive log records from the PU queue (Step 2, Section 3.5.3).
3. Sort the log records by key.
4. Take the first (unprocessed) log record and navigate through the B-Tree to the leaf node which is affected by this log record. Reread that leaf node from S3 using S3's *get-if-modified* method.
5. Apply all log records that are relevant to that leaf node.
6. If the timeout of the token received in Step 1 has not expired (with some padding for the *put*), *put* the new version of the node to S3; otherwise, terminate (same as Step 5, Section 3.5.3).
7. If the timeout has not expired, delete the log records which were applied in Step 5, from the PU queue.
8. If not all log records have been processed yet, goto Step 4; otherwise, terminate.

The details on how to navigate to a page, range scans, inserts, updates and deletes can be found in [Lom96]. The only required modifications to [Lom96] concern the atomicity of splits and deletions of nodes. The paper proposes to use a 2-phase commit protocol

to coordinate the node split, which is not applicable in the here presented architecture. Instead, we can achieve atomicity by either applying a recovery protocol similar to the one presented in Section 3.6.1 or by allowing dead-pages and, thus, garbage collections. The interested reader is referred to Appendix A.1 for more details.

3.5.6 Unsafe Propagation Time

All presented checkpoint protocols require setting a *PropPeriod*. The *PropPeriod* ensures that there is enough time to propagate changes inside the storage service. However, for many systems it is not possible to set this value as it might take an arbitrary time until an update becomes visible. For example, a system like Dynamo [DHJ+07] continues to accept updates even in the presence of network partitioning. In this situation, updates inside one partition become quickly visible but it might take an arbitrary time until the updates become visible in the whole system.

To overcome this problem, it needs to be ensured that the latest version is received before a checkpoint is applied.[8] The easiest way to ensure this property is by making use of the ordering of messages of the Advanced Queue Services (Section 3.3.4). That is, every checkpoint to a page contains in the header the latest applied MessageID. In addition, the latest MessageID is not deleted from the queue when applying a checkpoint. It follows that, if a new checkpoint needs to be done, the latest applied message has to be higher or equal to the first message in the queue. If this is the case, the page has been propagated successfully and the next checkpoint can be done. Otherwise, the checkpoint has to be postponed.

It is more complicated to ensure that the latest version of a client-side B-Tree is received because it consists of several pages. Several possible solutions exist: One is to introduce one queue per node inside the tree and to coordinate the node splitting, inserts etc. over the queues. Hence, the same technique as for the data pages could be used. Another possibility is to use an additional summarizing data structure similar to Merkle-trees (Section 2.3.1.3) stored inside the index information. Thus, whenever a change happens, this structure requires an update. This information, together with the latest MessageID can then be used to determine if the index corresponds to the latest version. Furthermore, this information can also be used to determine which data pages have to be updated for a given set of inserts/updates/deletes. If required, this

[8] We assume that the system always propagates the latest written version. This assumption holds in almost all systems, because even when using simple system timestamps, the time inside different nodes normally does not diverge more than the time between two consecutive checkpoints (e.g., 30s).

summarizing structure can even be split and stored in different files for space reasons or to allow for concurrent changes to the tree.

However, Advanced Queues have the drawback that they are not as highly available as Simple Queues and probably more expensive. Luckily, it is also possible to avoid the *PropPeriod* with Simple Queues. The simplest solution is to use the counter service. Every data page or index information stores an additional counter value, which is increased before writing it. After writing the page, the counter service is called with the URI of the page to increase the counter of the service to the same value. Only afterwards, the applied messages can be deleted safely. Before applying a checkpoint, the counter service is read to ensure that the counter of the page is equal to or higher than the one of the counter service. Otherwise, the last checkpoint has not propagated and the checkpoint has to be postponed. For the B-Tree itself the same summarizing structure can be used.

In order to completely avoid additional services, client counter IDs can be used as will be introduced for monotonicity guarantees in Section 3.6.2. These, together with postponing the deletion of records, always allow for reconstructing the page, even when a checkpoint has been done before the previous checkpoint was propagated. More details on this technique are given in Section 3.6.2.

Unfortunately, all methods introduce different levels of overhead and influence the availability. The last method ensures the highest availability, as a Simple Queue Service can even operate in the presence of network partitioning, but at the same time creates the highest overhead, as it requires to keep on to *(client, counter)* pairs and hold on to messages longer than otherwise required. In contrast, the use of Advanced Queues reduces the overhead but decrease the availability because Advanced Queues are typically not resilient against network partitioning (else, the strong order of messages could not be provided).

3.5.7 Checkpoint Strategies

The purpose of the previous two sections was to show *how* checkpoints are implemented. The protocols were designed in such a way that anybody could apply a checkpoint at any time. This section discusses alternative *checkpoint strategies*. A checkpoint strategy determines *when* and *by whom* a checkpoint is carried out. Along both dimensions, there are several alternatives.

A checkpoint on a page (or an index) X can be carried out by the following authorities:

- *Reader:* A reader of X.
- *Writer:* A client who just committed updates to X.
- *Watchdog:* A process which periodically checks PU queues.
- *Owner:* X is assigned to a specific client which periodically checks the PU queue of X.

In this work, we propose to have checkpoints carried out by *readers* and *writers* while they work on the page (or index). Establishing *watchdogs* to periodically check PU queues is a waste of resources and requires an additional infrastructure to run the *watchdogs*. Likewise, assigning *owners* to PU queues involves wasting resources because the *owners* must poll the state of their PU queues. Furthermore, *owners* may be offline for an undetermined amount of time in which case the updates might never be propagated from the PU queue to S3. The advantage of using *watchdogs* and assigning *owners* to PU queues is that the protocols of Sections 3.5.3, 3.5.4 and 3.5.5 are simplified (no LOCK queues are needed) because no synchronization between potentially concurrent clients is required. Nevertheless, we believe that the disadvantages outweigh this advantage.

The discussion of whether checkpoints should be carried out by *readers* or *writers* is more subtle and depends on the second question of *when* checkpoints should be carried out. In this work, we propose to use *writers* in general and *readers* only in exceptional cases (see below). A *writer* initiates a checkpoint using the following condition:

- Each data page records the timestamp of the last checkpoint in its header. For B-Trees, the timestamp is recorded in the meta-data (Section 3.2.1) associated to the root page of the B-Tree. For B-Trees, the timestamp is not stored in the root page because checkpointing a B-Tree typically does not involve modifying the root and rewriting the whole root in this event would be wasteful. The timestamp is taken from the machine that carries out the checkpoint. It is not important to have synchronized clocks on all machines; out-of-sync clocks will result in more or less frequent checkpoints, but they will not affect the correctness of the protocol (i.e., eventual consistency at full availability).
- When a client commits a log record to a data page or B-Tree, the client computes the difference between its current wallclock time and the timestamp recorded for

the last checkpoint in the data page / B-Tree. If the absolute value of this difference is bigger than a certain threshold (*checkpoint interval*), then the *writer* carries out a checkpoint asynchronously (not blocking any other activity at the client). The *absolute* value of the difference is used because out-of-sync clocks might return outrageous timestamps that lie in the future; in this case, the difference is negative.

The *checkpoint interval* is an application-dependent configuration parameter; the lower it is set, the faster updates become visible, yet the higher the cost (in $) for carrying out many checkpoints. The trade-offs of this parameter have been studied in [BFG$^+$08]. Obviously, the checkpoint interval should be set to a significantly larger value than the *timeout* on the LOCK for checkpoint processing used in the protocols. For a typical web-based application, a checkpoint interval of 30 seconds with timeouts on locks of 20 second is typical sufficient. Clearly, none of the protocols devised in this work are appropriate to execute transactions on hot-spot objects which are updated thousands of times per second.

Unfortunately, the *writer-only* strategy has a flaw. It is possible that a page which is updated only once is never checkpointed. As a result, the update never becomes visible. In order to remedy this situation, it is important that *readers* also initiate checkpoints if they see a page whose last checkpoint was a long time ago: A *reader* initiates a checkpoint randomly with a probability proportional to $1/x$ if x is the time period since the last checkpoint; x must be larger than the checkpoint interval. Thus, the longer the page has not been checkpointed after the checkpoint interval expired, the less likely a checkpoint is required in this approach. Initiating a checkpoint does not block the *reader*; again, all checkpoints are carried out asynchronously outside any transaction. Of course, it is still possible that an update from a PU queue is never checkpointed if the data page or index is neither read nor updated; however, we do not need to worry about this, because in this case the page or index is garbage. Further, the checkpoint interval only influences the freshness of the data but it is no guarantee for it. Freshness is a form of consistency and thus, can only be guaranteed if stronger consistency protocols (as shown in 3.6.4 and 3.6.5) are used.[9]

The proposed checkpointing strategy makes decisions for each data page and each index individually. There are no concerted checkpointing decisions. This design simplifies the implementation, but it can be the source of additional inconsistencies. If a

[9]With the simple queuing service it is not possible to give any freshness guarantee. Simple queues work even in the presence of network partitioning. Thus, per definition it might take an arbitrary time that an update becomes visible.

new record is inserted, for instance, the new record may become visible in a secondary index on S3 before it becomes visible in the primary index. Likewise, the query *select count(*) from collection* can return different results, depending on the index used to process this query. How to avoid such phantoms and achieve serializability is discussed in Sections 3.6.4 and 3.6.5. Unfortunately, serializability cannot be achieved without sacrificing scalability and full availability of the system.

3.6 Transactional Properties

The previous section has shown how *durability* can be implemented on top of an eventual consistent store with the help of a queuing service. No update is ever lost, updates are guaranteed to become visible to other clients (*eventual consistency* [TS06]), and the states of records and indexes persist until they are overwritten by other transactions. This section describes how additional transactional properties can be implemented (see also Table 3.4). Again, the goal is to provide these added properties at the lowest possible cost (monetary and latency) without sacrificing the basic principles of utility computing: scalability and availability. It is shown that atomicity and all client-side consistency levels described in [TS06] can be achieved under these constraints whereas *isolation* and *strict consistency* can only be achieved if we tolerate reduced availability. All protocols described in this section are layered on top of the basic protocols described in the previous section.

3.6.1 Atomicity

Atomicity implies that *all* or *none* of the updates of a transaction become visible. Atomicity is not guaranteed using the basic commit protocol depicted in Figure 3.2. If a client fails while processing a commit of a transaction, it is possible that the client already submitted some updates to the corresponding PU queues whereas other updates of the transaction would be lost due to the failure.

Fortunately, atomicity can be implemented using additional ATOMIC queues. An ATOMIC queue is associated to each client and implemented using the Simple Queue Service (Section 3.3.3). In fact, it is possible that a client maintains several ATOMIC queues in order to execute several transactions concurrently. For ease of presentation, we assume that each client has a single ATOMIC Queue and that a client executes transactions one after the other (of course, several operations can be executed by a client as part of the same transaction).

The commit protocol that ensures atomicity with the help of ATOMIC queues is shown in Algorithm 4. The idea is simple. First, all log records are sent to the ATOMIC queue, thereby remembering the message IDs of all messages sent to the ATOMIC queue in the set of LogMessageIDs (lines 1-3). Again, it is possible to exploit the maximum message size of the Queue Service by packing several log records into a single message in order to reduce the cost and response time. Once the client has written all log records of the transaction to its ATOMIC queue, the client sends a special *commitLog* record to the ATOMIC queue (line 5). At this point, the transaction is successfully committed and recoverable. Then, the client executes the basic commit protocol of Algorithm 4 (line 6); that is, the client sends the log records to the corresponding PU queues so that they can be propagated to the data and index pages in subsequent checkpoints. If the basic commit protocol was carried out successfully, the client removes all messages from its ATOMIC queue using the set of LogMessageIDs (lines 7-9).

Algorithm 4 Atomicity - Commit

Require: $AtomicQueueUri \leftarrow$ Atomic Queue URI for the Client
1: $LogMessageIDs \leftarrow \emptyset$
2: **for all** modified records R **do**
3: $LogMessageIDs$.add(sendMessage($AtomicQueueUri$, R.Log))
4: **end for**
5: $LogMessageIDs$.add(sendMessage($AtomicQueueUri$, CommitLog));
6: execute basic commit protocol (Algorithm 1)
7: **for all** i in $LogMessageIDs$ **do**
8: deleteMessage($AtomicQueueUri$, i)
9: **end for**

When a client fails, the client executes the recovery protocol of Algorithm 5 after the restart. First, the client reads all log messages from its ATOMIC queue (lines 1-6). Since Simple Queues are used and these queues do not give any completeness guarantees, probing the ATOMIC queue is carried out several times until the client is guaranteed to have received all messages from its ATOMIC queue.[10] If an Advanced Queue is used,

[10]This probing is a simplification. In the case the service does not provide any guarantee on the availability of messages, it is possible that log records are not propagated although the transaction submitted the *CommitLog*. However, there are many ways to overcome that problem. The simplest is, to submit a single message, the *CommitLog*, containing all log records. Thus, whenever the *CommitLog* message is received, all messages are available. If the message size is not sufficient, all log records can be stored inside an object on the storage service and the *CommitLog* contains a simple reference to the object. As a consequence, unsuccessful transactions have to be garbage collected in the moment it is safe to assume that the transaction failed (see also Section 3.5.6).

Algorithm 5 Atomicitiy - Recovery

Require: $AtomicQueueUri \leftarrow$ Atomic Queue URI for the Client
1: $LogMessages \leftarrow \emptyset$
2: $M \leftarrow$ receiveMessage($AtomicQueueUri$, ∞)
3: **while** $M.size() \neq 0$ **do**
4: $LogMessages$.add(M)
5: $M \leftarrow$ receiveMessage($AtomicQueueUri$, ∞)
6: **end while**
7: **if** CommitLog $\in LogMesssages$ **then**
8: execute basic commit protocol (Algorithm 1)
9: **end if**
10: **for all** l in $LogMessages$ **do**
11: deleteMessage($AtomicQueueUri$, l.MessageID)
12: **end for**

this iterative probing process is simplified. Once the client has read all log records from its ATOMIC queue, the client checks whether its last transaction was a winner; i.e., whether a *commitLog* record was written before the crash (Line 7 of Algorithm 5). If it was a winner, then the client re-applies all the log records according to the basic commit protocol and deletes all log records from its ATOMIC queue (Lines 10-12). For loser transactions, the client simply deletes all log records from its ATOMIC queue.

Naturally, clients can fail after restart and while scanning the ATOMIC queue. Such failures cause no damage. If in such a situation log records are propagated to PU queues twice or even more often, this is not an issue because applying log records is idempotent. The protocol assumes that no client fails permanently. If this assumptions does not hold, then a different client (or some other service) must periodically execute the recovery protocol of Algorithm 5 for inactive ATOMIC queues. If the *commitLog* record is deleted last in Lines 10-12 of Algorithm 5, then several recovery processes on the same ATOMIC queue can be carried out concurrently (and fail at any moment) without causing any damage. More details for concurrent recovery may be found in Section 3.6.3.

The guarantee given by the protocol is, that either all or none of the updates eventually become visible. Thus, no update is ever lost because of a failure. Still, the protocol provides relaxed atomicity guarantees according to some text-book definition as transactions might become partially visible.[11] The here-proposed protocol implies the

[11] For example, Raghu and Gehrke [RG02] define the atomicity property of transactions as "Either all actions are carried out or none are. Users should not have to worry about the effect of incomplete

existence of a transition phase because of the deferred and not synchronized check-pointing. During this transition phase, some of the updates from a transaction might be visible while some others might not. In this work, we see atomicity as a property which guarantees that all or no updates are applied even in the presence of failures and consider visibility as a problem of consistency and isolation. Thus, the consistency is still relaxed and almost 100% availability can be guaranteed as the protocol, as well as all the services it uses, can operate even during network partitioning.

3.6.2 Consistency Levels

Tanenbaum and van Steen describe different levels of consistency [TS06]. The highest level of consistency is *strict consistency*, also referred to as *strong consistency*. Strict consistency mandates that "every read on a data item x returns a value corresponding to the result of the most recent write on x" [TS06]. Strict consistency can only be achieved by synchronizing the operations of concurrent clients; isolation protocols are discussed subsequently in Sections 3.6.4 and 3.6.5). This section discusses how the other (weaker) levels of consistency described in [TS06] can be achieved. The focus is on so-called client-side consistency models [TS06, TDP$^+$94] because they are the basis for the design of most web-based services [Vog07]:

Monotonic Reads: *"If a client [process]*[12] *reads the value of a data item x, any successive read operation on x by that client will always return the same value or a more recent value"* [TS06]. This property can be enforced by keeping a record of the highest commit timestamp for each page which a client has cached in the past. If a client receives an old version of a page from the storage service (older than a version the client has seen before), the client can detect this and re-read the page from the storage service. As a consequence the availability might be reduced for single clients in certain failure scenarios (e.g., network partitioning) because re-reading the page from the storage service, until the correct version is found, might take arbitrarily long.

Monotonic Writes: *"A write operation by a client on data item x is completed before any successive write operation on x by the same client"* [TS06]. This level of consistency can be implemented by establishing a counter for each page (or index) at a client

transactions (say, when a system crash occurs)." Whereas Conolly and Begg [CB04] define atomicity of transactions as: "A transaction is an indivisible unit that is either performed in its entirety or is not performed at all."

[12][TS06] uses the term *process*. We interpret that term as *client* in our architecture. If we would interpret this term as *transaction*, then all the consistency levels would be fulfilled trivially.

(as for monotonic reads) and incrementing the counter whenever the client commits an update to that page (or index). The pairs *(client id, counter value)* of the latest updates of each client are stored in the header of each page and in the log records. As a result, the log records can be ordered during checkpointing and out-of-order log records can be detected in the event that SQS does not return all relevant records of a PU queue (Section 3.3.3). If an out-of-order log record is found during checkpointing, this log record is not applied and its application is deferred to the next checkpoint.

Furthermore, using the counter values also allows to use storage services which have no guarantees on the propagation time together with Simple Queues (see discussion in Section 3.5.6). To ensure that, the checkpoint protocol of the previous section copies successful applied messages to an additional history queue before deleting them from the PU queue. In the event that a new checkpoint is written before the old checkpoint has propagated, the history queue can always be used to reconstruct the overwritten checkpoint. Theoretically, the messages inside the history queue have to be kept forever because the time that the storage service propagates the new change can take infinitely long. In practice, long propagation times (e.g., more than 1-2 seconds) only happen in the case of failures and those are usually quickly detected (e.g., by gossiping) and reported online. Hence, the history only needs to be kept longer than a few minutes if failures are encountered.

As a final problem of this protocol the header information can grow as more users update the page. Thus, the counter values have to be garbage collected. This problem is already known from vector-clocks (see Section 2.3.1.3) and can for example be solved by using a last-recently-used truncation [DHJ+07].

Read your Writes: *"The effect of a write operation by a client on data item x will always be seen by a successive read operation on x by the same client"* [TS06]. This property is automatically fulfilled in the architecture of Figure 3.1 if *monotonic reads* are supported.

Write follows Read: *"A write operation by a client on data item x following a previous read operation on x by the same client, is guaranteed to take place on the same or a more recent value of x that was read"* [TS06]. This property is fulfilled because writes are not directly applied to data items; in particular, the *posting a response* problem described in [TS06] cannot occur using the protocols of Section 3.5.

Session Consistency: A special and in practice quite popular form of monotonicity is session consistency. Session consistency guarantees monotonicity as long the session remains intact. For example, session consistency allows for easily implementing

monotonicity guarantees in a server-centric setup by using the monotonicity protocols from above. When a user accesses the system, it implicitly creates a session with one of the servers. All preceding requests from the same user are than forwarded to the same server (e.g., by using a session-id in every request). The server uses the protocol as described above and acts as one single client with monotonicity guarantees. The biggest benefit of this setup is, that the server can cache the changes from the users and thus, guarantees monotonicity almost without any additional overhead. However, if the server crashes the session is lost and the monotonicity is no longer guaranteed. If this is not acceptable and sessions are required to survive server crashes, the protocols of [TDP+94] are applicable.

Similar to the client-centric protocols, several data-centric consistency levels defined in [TS06] can be implemented on eventually consistent storage services (e.g., FIFO consistency). Going through the details is beyond the scope of this thesis.

All client protocols increase the level of consistency. Those protocols ensure an order of updates per single client which is a form of consistency. According to the CAP theorem failures can decrease the availability of the system. However, the availability only decreases for single clients at a time and not for the whole system. This is due to the fact that only those clients which are not able to receive a missing message have to wait until the message becomes available. All other clients can operate normally.

3.6.3 Atomicity for Strong Consistency

In Section 3.6.1 we presented how atomicity can be achieved. This protocol guarantees that either all or none of the updates become visible as long as the client recovers at some point. Unfortunately, this protocol gives no guarantees on how long such a recovery will take and therefore how long an inconsistent state can exist. For building protocols with stronger consistency guarantees we need to minimize the inconsistency timeframe. As a solution clients could be allowed to recover for failures from other clients. Another problem encountered in a highly distributed system with thousands of clients is, that in the worst case a single client is able to block the whole system (Sections 3.6.4 and 3.6.5 will demonstrate this issue in more detail). Transaction timeouts are one solution to this problem and are applied here. As the higher consistency levels require Advanced Queues anyway, the atomicity protocol presented here also makes use of Advanced Queue features.

The revisited code for Atomicity is shown in Algorithm 6 and Algorithm 7. The main differences are:

Algorithm 6 Advanced Atomicitiy - Commit

Require: $AtomicQueueUri$, $AtomicCommitQueueUri$, $ClientID$, $TransactionStartTime$, $TransactionTimeout$

1: $LogMessageIDs \leftarrow \emptyset$
2: **for all** modified records R **do**
3: $LogMessageIDs$.add(sendMessage($AtomicQueueUri$, R.Log, $ClientID$))
4: **end for**
5: **if** CurrentTime() $-TransactionStartTime < TransactionTimeout$ **then**
6: $C \leftarrow$ sendMessage($AtomicCommitQueueUri$, CommitLog, $ClientID$);
7: execute basic commit protocol (Algorithm 1)
8: deleteMessage($AtomicCommitQueueUri$, C)
9: **end if**
10: **for all** i in $LogMessageIDs$ **do**
11: deleteMessage($AtomicQueueUri$, i)
12: **end for**

- The protocol only requires two ATOMIC queues for all clients: the ATOMIC COMMIT Queue and the ATOMIC queue. PU messages are sent to the ATOMIC queue whereas commit messages are sent to the ATOMIC COMMIT Queue. We separate those two queues for simplicity and performance reasons.

- The commit message is only sent if the transaction is faster than the transaction timeout. Therefore, long-running transactions are aborted.

- All messages carry the ClientID as additional key.

The revised recovery algorithm is presented in Algorithm 7. In order to enable clients to recover for failures from other clients, clients are required to check on commit messages from others. This is typically done in the beginning of every strongly consistent transaction (e.g., described in the next subsection).[13] If we find an old message, meaning older than a timeframe called AtomicTimeout, it is assumed that recovery is required (Line 1). Setting the AtomicTimeout in a sensible way is crucial. Considering too young messages as old might result in false positives, setting the value too high holds the database in an inconsistent state for too long. If a client determines it has to do recovery, it tries to receive the recovery lock. If the client is able to get the lock (Line 3), the client is allowed to perform the recovery steps (Lines 4-11). If not, the transaction

[13] It is also possible to perform the check and recovery by using a watchdog which periodically checks the ATOMIC queues. However, discussing such watchdogs is beyond the scope of this thesis.

Algorithm 7 Advanced Atomicitiy - Recovery

Require: $AtomicQueueUri$, $AtomicCommitQueueUri$, $AtomicTimeout$

```
1:  M ← receiveMessage(AtomicCommitQueueUri, ∞, AtomicTimeout)
2:  if M.size() > 0 then
3:     if acquireXLock(AtomicCommitQueueUri, ClientID) then
4:        for all m in M do
5:           LogMessages ← receiveMessage(AtomicQueueUri, ∞, m.Key-1, m.Key+1)
6:           execute basic commit protocol for LogMessages (Algorithm 1)
7:           for all l in LogMessages do
8:              deleteMessage(AtomicQueueUri, L.MessageID))
9:           end for
10:          deleteMessage(AtomicCommitQueueUri, m.MessageID))
11:       end for
12:       releaseXLock(AtomicQueueUri, ClientID)
13:    else
14:       abort() //Database in recovery
15:    end if
16: end if
```

is aborted as the database is in recovery (Line 15). To allow just one client to recover is the easiest way of recovery but might also result in longer unavailability times of the system. If those outages are unacceptable or/and clients are assumed to crash more often, solutions include parallel recovery from several clients up to partitioning the ATOMIC queues to consistency entities. For simplicity, we do not further discuss those strategies. The actual recovery is similar to the recovery mechanism described before. Unlike before, we now need to make sure to recover for a certain commit message only. To do this kind of group by we use the feature of the Advanced Queues that allows filtering the messages according to the user-defined key (Line 6).

Again, crashes during the recovery do not harm, as the logs are idempotent. Also, if the lock-timeout for the recovery expires or a transaction simply takes longer for the atomic commit, no damage is done for the same reason.

An optimization we apply in our implementation is to piggy back the log messages with the commit message if the size of the log messages is not too big. Thus, only one additional message has to be inserted and deleted for every atomic commit.

The use of the Advanced Queues decreases the availability of the system (see Section 3.3.6) for all clients. However, the advanced commit protocol is only required to-

gether with strong consistency protocols as snapshot isolation or two-phase-locking presented in the next two subsection. As those protocols want to provide strong consistency, it follows from the CAP theorem that the availability is reduced if network partitions are possible. Hence, this protocol as well as all the following ones sacrifice availability for consistency.

3.6.4 Generalized Snapshot Isolation

The idea of snapshot isolation is to serialize transactions in the order of the time they started [BHG87]. When a transaction reads a record, it initiates a time travel and retrieves the version of the object as of the moment when the transaction started. For this purpose, all log records must support undo-logging (Section 3.4.5) and log records must be archived even beyond checkpointing. Hence, the checkpointing protocols of the previous section have to be adapted to keep the applied messages (i.e., Line 15 in Algorithm 2 and Line 29 in Algorithm 3). Generalized snapshot isolation relaxes snapshot isolation in the sense that it is not required to use the latest snapshot [EZP05]. This allows higher concurrency in the system as several transactions are able to validate at the same time. Also, in the case of read-only transactions a snapshot is more likely to be handled by the cache.

We therefore propose a protocol applying generalized snapshot isolation (GSI). GSI requires that every transaction is started by a call to BeginTransaction, presented in Algorithm 8. To ensure that a consistent state exists, BeginTransaction first checks if recovery is required. Afterwards, the protocol retrieves a valid SnapshotID from the Advanced Counter Service. This ID represents the snapshot that the transaction will be working on.

Algorithm 8 GSI - BeginTransaction

Require: $DomainUri$

1: execute advanced atomic recovery protocol
2: $SnapshotID \leftarrow$ getHighestValidatedValue($DomainUri$)

To ensure that every read/write is performed on the correct snapshot it is also required to apply or rollback logs from the PU queues. Algorithm 9 shows the necessary modifications to the BufferManager for pages. If the BufferManager receives a get request for a page, it also requires the SnapshotID. Depending on the fetched page being older or younger than the SnapshotID, log records are rolled back or applied in the normal way. The correctness of the protocol is only guaranteed if all messages inside a queue can

be received and related to a Snapshot. Hence, we again assume Advanced Queues and use the SnapshotID as the MessageID. The interested reader might have noticed, that this also requires a slight modification of the atomicity protocol: Log messages sent to the ATOMIC queue are also required to hold on to the SnapshotID to be used in the case of a recovery.[14]

Algorithm 9 GSI - BufferManager - Get Page

Require: $SnapshotID$, $PageUri$

1: $P \leftarrow$ fetch(PageUri)
2: **if** $SnapshotID < P$.SnaptshotId **then**
3: $\quad M \leftarrow$ receiveMessage(P.Uri, ∞, $SnapshotID$, P.SnaptshotId + 1)
4: $\quad$ rollback M from P
5: **else if** $SnapshotID > P$.SnaptshotId **then**
6: $\quad M \leftarrow$ receiveMessage(P.Uri, ∞, P.SnaptshotId, $SnapshotID$ + 1)
7: $\quad$ apply M to P
8: **end if**
9: P.SnaptshotId $\leftarrow SnapshotID$

The commit shown in Algorithm 10 forms the last step of the GSI. It consists of two steps, the validation phase and the actual commit phase. The validation phase ensures that we do not have two conflicting writes for our snapshot whereas the commit phase is the actual commit. To be able to successfully validate, the algorithm first checks if the database needs to be recovered (Line 1). Furthermore, the validation requires that no other transaction validates the same record at the same time. The protocol ensures this by acquiring a lock for every record in the write set (Lines 5-10). If it is not possible to acquire a lock for one of the records, the transaction aborts.[15] Deadlocks are avoided by sorting the records according to the page Uri and RecordID. Once all locks are acquired, the protocol checks the PU queues for conflicting updates. Therefore, a CommitID is received by incrementing the Advanced Counter. Then, every PU queue

[14] The presented protocol does not consider inserts which have been sent to the collection queue but are not yet forwarded to the page queues. However, it is questionable if those inserts have to be considered. Snapshot isolation (no matter if general or traditional) does not guarantee serializability. One prime reason is, that inserts can never cause a transaction to fail during validation. Hence, considering inserts can only reduce the likelihood of inconsistency caused by inserts but never completely prevent it. Nevertheless, it is straightforward to extend the protocol to include all inserts from the collection PU queues.

[15] This might lead to livelocks if after an abort the same transaction immediately tries to modify the same data again. However, to avoid this situation, waiting for locks and other methods from traditional database systems can be applied. In favor of simplicity those additional techniques are not further discussed.

which might contain conflicting updates is checked by retrieving all messages from the used SnapshotID up to the CommitID. If one of the received log records conflicts with one of the records in the write set the transaction aborts. Else, the validation phase has been successful and the actual commit phase starts. The commit is similar to the one of the advanced atomicity protocol - as only difference, the message key is set to the CommitID. Once the atomic commit finishes, some cleanup is performed: Locks are released and the CommitID gets validated to become a new valid snapshot (Lines 20 - 26). The TransactionTimeout starts with the commit and not with the BeginTransaction. This is an optimization applied to allow for long-running read requests.

Finally, to fully enable GSI the counter and lock timeout have to be chosen carefully. If they are too small, to many write-transactions might be aborted (read-only transaction are never aborted and always work on a consistent snapshot). The time spanned by *TransactionTimeout + AtomicityTimeout* is the maximum time required to detect that the database requires recovery and thus, effects the availability of the system in case of failures. Assuming that one transaction starts shortly before the recovery would be detected, then all locks and GSI are not allowed to timeout before the validation of the transaction finished. This corresponds at most to the time spanned by another *TransactionTimeout* period. As a consequence, the timeouts for the counter and lock service should be set to a value bigger than *2 * TransactionTimeout + AtomicityTimeout*.[16] As this might be quite a long period, higher concurrency in snapshot isolation is achieved by explicitly releasing locks and validating the counter. Again, the TransactionTimeout starts with the commit and not with the BeginTransaction. Thus, the protocols only restricts the time required to commit a change, not the running time of the whole transaction.

Again, the algorithm can fail at any time and guarantees the consistency as defined by snapshot isolation. If Algorithm 10 fails before Line 21, no updates have been sent to the queues, all locks will time-out and the SnapshotID will automatically validate. If the client fails during the atomic commit in Algorithm 6 before the commit message has been written to the queue (Line 6), the same situation exists and no harm is done. If the transaction fails after the CommitLog has been written but before all updates are forwarded to the pages, two situations are possible: First, the AtomicityTimeout has expired and the crash is detected in Line 1 of Algorithm 10 by concurrent clients and corrected and hence, any possible conflict will be detected during the validation phase. Second, the AtomicityTimeout has not expired and the crash cannot be detected at this

[16] For simplicity, we do not consider network latency at his point. To ensure correctness in all cases, the network latency for sending the CommitLog in Algorithm 6 has to be included for the LockTimeouts and the sendMessage request itself requires another timeout.

Algorithm 10 GSI - Commit

Require: $DomainUri$, $SnapshotID$, $ClientID$, $WriteSet$

1: execute advanced atomic recovery protocol
2: $TransactionStartTime \leftarrow$ CurrentTime()
3: *//Validation Phase*
4: sort $WriteSet$ according to the PageUri and RecordId
5: $P \leftarrow \emptyset$
6: **for all** records r in $WriteSet$ **do**
7: P.add(r.Page)
8: $RecordUri \leftarrow$ concat(P.Uri, R.Id)
9: **if** NOT acquireXLock($RecordUri$, $ClientID$) **then**
10: abort()
11: **end if**
12: **end for**
13: $CommitId \leftarrow$ increment($DomainUri$)
14: **for all** pages p in P **do**
15: $M \leftarrow$ receiveMessage(p.Uri, ∞, $SnapshotID$, $CommitId$)
16: **if** M contains a log for a item in WriteSet **then**
17: abort()
18: **end if**
19: **end for**
20: *//Commit Phase*
21: execute advanced atomic commit protocol
22: validate($DomainUri$, $CommitId$)
23: **for all** record r in $WriteSet$ **do**
24: $RecordUri \leftarrow$ concat(P.Uri, R.Id)
25: releaseXLock($RecordUri$, $ClientID$)
26: **end for**

stage (Algorithm 10, Line 1). In this case, the elapsed time after the commit of the first transaction is not more than *TransactionTimeout + AtomicityTimeout*. The locks and counters do not time out before ***2*** ** TransactionTimeout + AtomicityTimeout*. Thus, the validation phase is correct because possible conflicts will result in an unsuccessful lock request. Further, if the commit requires more time than *TransactionTimeout* no harm is caused, as the transaction will fail before the commit-message is send. As a result, it is always guaranteed that all messages for a snapshot received from the counter service are accessible for the time-travel.

For the purpose of snapshot isolation, log records can be garbage collected from the archive of log records if all the transactions using the SnapshotID of the log record or a younger id have either committed or aborted. Keeping track of those transactions can be achieved by using an additional counter. Alternatively, garbage collection can be done by enforcing the read and write time to be part of the transaction time. Log records older than the TransactionTimeout can then automatically be garbage collected. The garbage collection itself can in turn be integrated into the checkpointing protocol.

The presented protocol aborts if the database is in recovery. This is not really necessary, as it is possible to include the atomic queues during time-travel and validation. The system can continue to operate even in the presence of recovery. The required changes to the protocol are rather simple. In the beginning of an transaction, the atomic queue has to be read. If messages older than the AtomicityTimeout are found, they have to be buffered and used with all the other messages to create the snapshot. During validation, the ATOMIC queues have to be read again for timed-out messages. This is required, as it is not guaranteed that locks hold until the validation phase of the transaction starts. As the recovery protocol first forwards the messages to the corresponding queues, and afterwards deletes them from the ATOMIC queues, it is guaranteed that no message can be missed which has committed successfully.

Furthermore, our algorithms can be extended to reuse an existing snapshot if it is determinable that a transaction is read-only. Doing so allows answering read-transaction from the cache, hence reducing transaction time and cost.

3.6.5 2-Phase-Locking

Next to snapshot isolation we also implement 2-Phase-Locking (2PL) which allows serializability. Our 2PL protocol is rather traditional: It uses the Lock Service to acquire read- and write-locks and to propagate read- to write-locks. For simplicity, we abort transactions if they are not able to receive a lock. This is only possible because we do fine-grained locking on record-level and do not expect a lot of conflicts. For other scenarios, waiting might be advisable together with deadlock detection on the Locking Service. Atomicity for the 2PL works along the same lines as the GSI except that the TransactionTimeout starts with the begin of the transaction and not with the commit. So read heavy transactions also underlie a timeout, which can be an important drawback for certain scenarios. Again, similar to GSI it is required to consider the messages from the PU queues, this time to ensure that the latest state is read. As the protocol design closely follows the text-book description, we do not discuss it any further.

3.7 Implementation on Amazon Web Services

This section describes the implementation of the API of Section 3.3 using services available from Amazon. Although we restrict it to Amazon, other providers offer similar features (see also Section 2.3.2) and allow for adopting the discussed methods.

3.7.1 Storage Service

The API for the reliable storage service has already been designed in the lines of storage services from Google (BigTable), Amazon (S3) and others. It just requires a persistent store with eventual consistency properties with latest time-stamp conflict resolution or better for storing large objects. The mapping of the API to Amazon S3 is straightforward (compare Section 3.2.1) and not discussed in detail. However, note that Amazon makes use of a concept called Bucket. Buckets can be seen as big folders for a collection of objects identified by the URI. We do not require buckets and assume the bucket name to be some fixed value.

3.7.2 Machine Reservation

The Machine Reservation API consists of just two methods, *start* and *stop* of a virtualized machine. Consequently, Amazon Elastic Compute Cloud (EC2) is a direct fit to implement the API and additionally offers much more functionality. Furthermore, EC2 also enables us to experiment with alternative cloud service implementations and to build services which are not offered directly in the cloud. This not only saves a lot of money as transfer cost between Amazon machines is for free, but also makes intermediate service requests faster due to reduced network latency.

3.7.3 Simple Queues

Simple Queues do not make any FIFO guarantees and do not guarantee that all messages are available at all times. One way to implement Simple Queues is using Amazon SQS although SQS is not reliable because it deletes messages after 4 days. If we assume that for any page a checkpoint happens in less than 4 days, SQS is suitable for implementing Simple Queues. Unfortunately, another overhead was introduced with

the SQS version from 2008-01-01; that is, it is not possible to delete messages directly with the MessageID. Instead, it is required to receive the message before deleting it. Especially for the atomicity this change of SQS made the protocol to cause more overhead and consequently to be more expensive. Alternatively, Simple Queues can be implemented by S3 itself. Every message is written to S3 using the page URI, a timestamp and ClientID to guarantee a unique MessageID. By doing a prefix scan per page URI, all messages can be retrieved. S3 guarantees reliability and has ”no” restriction on the message size, which allows big chunks (especially useful for atomicity). It is therefore also a good candidate for implementing Simple Queues. Last but not least, Simple Queues can be achieved by the same mechanism as the Advanced Queues, see next section.

3.7.4 Advanced Queues

Advanced Queues are most suited to be implemented on top of EC2. The range of possible implementations is huge. A simple implementation is to build a simple in-memory queue system with a recovery log stored on EBS (Section 3.2.3) and time-to-time checkpointing to S3. Amazon claims an annual failure rate (AFR) for a 20 GB EBS drive of 0.1% - 0.5%, which is 10 times more reliable than a typical commodity disk drive. For log recovery with regular snapshots typically not more than 1 GB is required, resulting in an even better AFR. If higher reliability and availability is desired (EBS is a single data center solution), the queue service has to be replicated itself. Possibilities include replication over data-center boundaries using Paxos protocols [Lam98, CGR07] or multi-phase protocols [OV99].

Next to the reliability of the queuing service, load balancing is the biggest concern. Usage frequencies of queues tend to differ substantially. We avoid complex load balancing by randomly assigning queues to servers. This simple load balancing strategy works the best if none of the queues is a bottleneck and is better the bigger the overall number of queues is. Unfortunately, the collection queues for inserts are natural bottlenecks. However, small changes to the protocols allow for splitting those queues to several queues and hence, avoid the bottleneck. Still, more advanced load balancing strategies for the queuing service might be required. Solutions include queue handover with Paxos or multi-phase protocols. However, building more reliable and autonome queues is research for itself and we restrict our implementation to a simple in-memory solution with EBS logging.

3.7.5 Locking Service

AWS does not directly provide a locking service. Instead, it refers to SQS to synchronize processes. Indeed, SQS can be used as a locking service, as it allows to retrieve a message exclusively. Thus, locks can be implemented on top of SQS by holding one queue per lock with exactly one message. The timeout of the message is set to the timeout of the lock. If a client is able to receive the lock message, the client was able to receive the lock. Unfortunately, Amazon states that it deletes messages older than 4 days and even queues that have not been used for 30 consecutive days. Therefore, it is required to renew lock messages within 4 days. This is especially problematic as creating such a message is a critical task. A crash during deletion/creation can either result in an empty queue or in a queue with two lock messages. We therefore propose to use a locking service on EC2. Implementations for such a locking service are wide-ranging: The easiest way is to have a simple fail-over lock service. Thus, one server holds the state for all locks. If the server fails it gets replaced by another server with a clean empty state. However, this server has to reject lock requests, until all the lock timeouts since the failure have expired. Hence, all locks would have been automatically returned anyway. The lock manager service does not guarantee 100 percent availability, but it guarantees failure resilience. Other possibilities include more complex synchronization mechanisms like the ones implemented in Chubby [Bur06] or ZooKeeper [Apa08]. Again, we implement the simplest solution and build our services on top of EC2 using the described fail-over protocol. Developing our own locking service on EC2 gives the additional advantage of easily implementing shared, exclusive locks and the propagation from shared to exclusive without using an additional protocol. It seems quite likely that such a locking service will be offered by many cloud providers in the near future. Literature already states that Google, Yahoo already use such a service internally.

3.7.6 Advanced Counters

As for the locking service, using EC2 is the best way to implement advanced counters. If a server which hosts one (or several) counter(s) fails, then new counter(s) must be established on a new server. To always ensure increasing counter values, counters work with epoch numbers. This implies that if the counter fails, the epoch number is increased. Every counter value is prefixed with this epoch number. If a machine instance fails, a new machine replaces the service with a clean state, but with a higher epoch number. In other words, the counter service is not reliable and when it fails it

loses its state. Again, like for the lock service, the fail-over time has to be longer than the counter-validation time. This ensures for GSI that requesting the highest validated value does not reveal an inconsistent state.

3.7.7 Indexing

For hosted indexes we make use of Amazon's SimpleDB service. Every primary and secondary index is stored in one table inside SimpleDB. The translation of key searches and range queries to SimpleDBs query language is straightforward and not further discussed. Results from SimpleDB are cached on the client-side to improve performance and reduce the cost. SimpleDB is only eventual consistent. As a consequence, the *Atomicity* and *Snapshot* protocol together with SimpleDB do not provide exactly the specified consistency level. In the case of the primary index, the eventual consistency property can be compensated by considering the messages in the collection queues (assuming that a maximum propagation time exist). However, for secondary indexes it is not possible without introducing additional overhead. Thus, the *Atomicity* and *Snapshot* protocol do not provide the specified level of consistency if an secondary index access is made.[17] This problem does not exists with client-side indexes, which are implemented similar to [Lom96] as discussed several times before.

[17] This behavior is similar to Google's MegaStore implementation [Ros09].

3.8 Performance Experiments and Results

3.8.1 Software and Hardware Used

We have implemented the Cloud API of Section 3.3 on AWS (S3 and EC2) as discussed in the previous section. Furthermore, we have implemented the protocols presented in Sections 3.5 and 3.6 on top of this Cloud API and the alternative client-server architecture variants described in Section 3.4. This section presents the results of experiments conducted with this implementation and the TPC-W benchmark.

More specifically, we have implemented the following consistency protocols:

- *Naïve:* As in [BFG+08], this approach is used as a baseline. With this protocol, a client writes all dirty pages back to S3 at commit time, without using queues. This protocol is subject to lost updates because two records located on the same page may be updated by two concurrent clients. As a result, this protocol does not even fulfil eventual consistency. It is used as a baseline because it corresponds to the way that cloud services like S3 are used today.
- *Basic:* The basic protocol depicted in Figure 3.2. As explained in Section 3.5, this protocol only supports *eventual consistency*.
- *Atomicity:* The atomicity protocol of Section 3.6.3 in addition to the *monotonicity* protocols which are specified in detail in [BFG+08], all on top of the *Basic* protocol.
- *Locking:* The locking protocol as described in Section 3.6.5. This protocol implements strong consistency.
- *Snapshot Isolation:* The snapshot isolation protocol as described in Section 3.6.4. This protocol fulfils the snapshot isolation level as defined in [EZP05].

3.8.1.1 TPC-W Benchmark

To study the trade-offs of the alternative consistency protocols and architecture variants, we use a TPC-W-like benchmark [Cou02]. The TPC-W benchmark models an online bookstore and a mix of different so-called *Web Interactions (WI)*. Each WI corresponds to a *click* of an online user; e.g., searching for products, browsing in the product catalog, or shopping cart transactions. The TPC-W benchmark specifies three kinds of workload mixes: (a) browsing, (b) shopping, and (c) ordering. A workload mix specifies the

probability for each kind of request. In all the experiments, we use the Ordering Mix as the most update-intensive mix: About one third of the requests involve an update of the database.

The TPC-W benchmark applies two metrics. The first metric is a throughput metric and measures the maximum number of valid WIs per second (i.e., requests per second). The second metric defined by the TPC-W benchmark is Cost/WIPS with WIPS standing for Web Interactions Per Second at the performance peak. This metric tries to relate the performance (i.e., WIPS) with the total cost of ownership for a computer system. To this end, the TPC-W benchmark gives exact guidance concerning the computation of a system's cost.

For our experiments, both metrics have been relaxed in order to apply to cloud systems. Ideally, there exists no maximum throughput and the cost is not a fixed value but depends on the load. Therefore, we measure the *average response time* in secs for each WI and the *average cost* in milli-$ per WI. In contrast to the WIPS measures of the TPC-W benchmark these metrics allow for comparing the latency and cost trade-offs of the different consistency protocols and architecture variants. For the scalability experiments, however, we report WIPS to demonstrate the scalability. In all experiments, response time refers to the end-to-end wallclock time from the moment a request has been initiated at the end-user's machine until the request has been fully processed and the answer has been received by the end-user's machine.

Throughout this section, we do not present error bars and the variance of response times and cost. Since the TPC-W benchmark contains a mix of operations with varying profile, such error bars would merely represent the artefacts of this mix. Instead, we present separate results for read WIs (e.g., searching and browsing) and update WIs (e.g., shopping cart transactions) whenever necessary. As shown in [BFG+08], the variance of response times and cost for the same type of TPC-W operation is not high using cloud computing. In this performance study, we have made the same observation.

As already mentioned, the TPC-W benchmark is not directly applicable to a cloud environment (see also [BKKL09]). Thus, next to the metrics we apply the following changes:

- Benchmark Database: The TPC-W benchmark specifies that the size of the benchmark database grows linearly with the number of clients. Since we are specifically interested in the scalability of the services with regard to the transactional workload and elasticity with changing workloads, all experiments are carried out with a fixed benchmark database size and without pre-filling the order history.
- Consistency: The TPC-W benchmark requires strong consistency with ACID trans-

actions. As shown before, not all protocols support this level of consistency.

- Queries: We change the bestseller query to randomly select products. Else, this query is best solved by a materialized view which is currently not supported in our implementation.

- We concentrate on measuring the transaction part and do not include the cost for picture downloads or web-page generation. Pictures can be stored on S3, so that they just add an additional fixed dollar cost for the transfer.

- Emulated Browsers: The TPC-W benchmark specifies that WIs are issued by emulated browsers (EB). According to the TPC-W benchmark, emulated browsers use a wait-time between requests. Here, we do not use a wait-time for the browser emulation for simplicity and in order to scale more quickly to higher loads. Again, this does not influence the relations between the results for the different configurations.

- HTTP: The TPC-W benchmark requires the use of the HTTPS protocol for secure client / server communication. As a simplification, we use the HTTP protocol (no encryption).

3.8.2 Architectures and Benchmark Configuration

As discussed in Section 3.4, there are several alternative client-server architectures and ways to implement indexing on top of cloud services like AWS. In this study, the following configurations are applied:

- *EU-BTree:* The whole client application stack of Figure 3.1 is executed on "end-user" machines; i.e., outside of the cloud. For the purpose of these experiments, we use Linux boxes with two AMD 2.2 GHz processors located in Europe. In this configuration, client-side B-Tree indexes are used for indexing; that is B-Tree index pages are shipped between the EU machines and S3 as specified in Section 3.5.

- *EU-SDB:* The client application stack is executed on "end-user" machines. Indexing is done using SimpleDB (Section 3.2.5). Again, the consistency levels for *Atomicity* and *Snapshot* are not directly comparable to the EC2-BTree configuration as secondary indexes suffer from the eventual consistency property of SimpleDB.

- *EC2-BTree:* The client application stack is installed on EC2 servers. On the "end-user" side, only TPC-W requests are initiated. Indexing is done using a "client-side" B-Tree.
- *EC2-SDB:* The client application stack is installed on EC2 servers. Indexing is carried out using SimpleDB.

All five studied consistency protocols (Naïve, Basic, Atomicity, Locking, and Snapshot Isolation) support the same interface at the record manager as described in Section 3.4.2. Furthermore, all four client-server and indexing configurations support the same interfaces. As a result, the benchmark application code is identical for all variants.

All experiments on the EU side are initiated from a single machine located in Europe. This machine, which simulats "end-user" clients, is a Linux box with two AMD processors and 6 GB RAM. For all EC2 variants, we use Medium-High-CPU EC2 instances. A TPC-W application installed on EC2 acts as web server and application server at the same time. Clients can connect to the server which in turn communicates with the cloud services. For the scale-out experiment, we have experienced problems to generate the high load from our cluster in Europe.[18] Hence, we have decided to simulate the user requests, by running the client directly on the web server and application server and add a fixed latency (0.1s) and cost (0.020752 milli-$) for each generated page to simulate transfer cost ($) and latency (ms). These average latency times and network cost have been calculated upfront in a separate experiment. As the network latency and cost per page is independent of the used protocol or index, adding a fixed latency and cost value per transaction does not impact the significance of the experiments.

The data-size for all basic and tuning experiments is set to 10,000 products and accessed by 10 concurrent emulated browser issuing permanent requests without any wait-time. Unless stated otherwise, the page size in all experiments is set to 100 kb, the checkpoint interval to 45 seconds and the time-to-live of data pages to 30 seconds. For all our experiments we use a US Amazon data center and thus, cost refers to the charges of Amazon for using EC2 and S3 inside a US location. Furthermore, for the sake of uniformity, we use for all experiments our Advanced Queue Service implementation. Otherwise the response times for the different protocols would not have been comparable. Requests to advanced services such as Advanced queues, counters, and locks are priced at USD 0.01 per 1000 requests. The alternative would have been a dynamic pricing based on the utilization of the system using the EC2 instance prices.

[18] Partly, the network latency has caused a problem. We received several mails from ETH system administrators assuming intrusions inside the network. Furthermore, our cluster has not been exposed to the internet and we would have been required to change the complete infrastructure.

We decided for a fixed price to make the cost comparable to implementations on top of SQS and to avoid effects of different concurrency degrees. However, the price is rather high and compares to a not fully-utilized EC2 instance. A fully-utilized queue implementation running on EC2 using EBS logging is able to lower the cost to less than USD 0.01 per 10,000 requests.

If not stated otherwise, we run all experiments 10 times for 300 seconds. In two cases, we have encounter outages from Amazon's internal network. In such cases, the results have been excluded and the experiments repeated.

3.8.3 Response Time Experiments

3.8.3.1 EU-BTree and EU-SDB

Figure 3.3 shows the average response times per WI of the TPC-W benchmark for the EU-BTree and EU-SDB configurations.

Comparing the various protocols, the results are not surprising. The *Basic*, *Atomic* and *Snapshot Isolation* protocols all make use of extensive caching. Only *Locking* always requires the latest snapshot resulting in more service requests to bring the data up-to-date. These differences become more obvious in Figure 3.4 which shows the average response time for update WIs only (e.g., putting items into the shopping cart). Now all protocols behave differently. Our baseline *Naïve* shows the worst performance because it writes all updated pages directly to S3, which is expensive. The *Atomic* protocol is

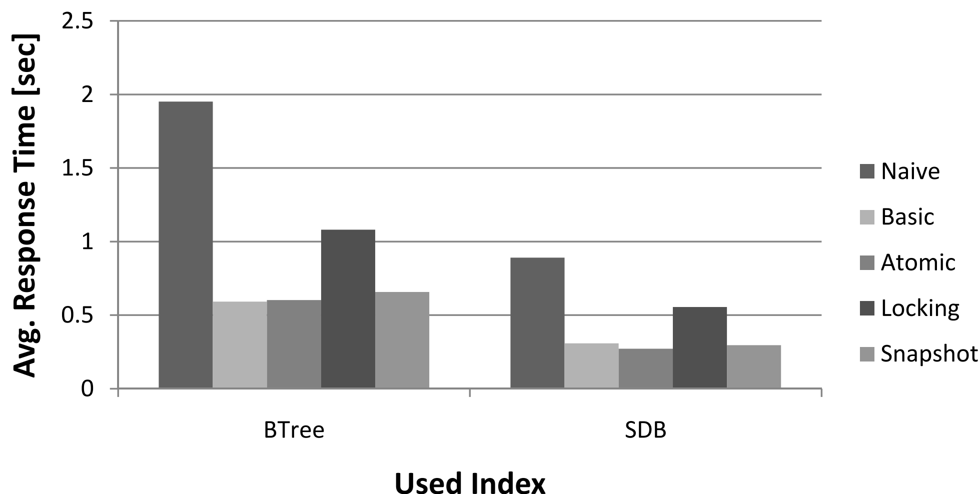

Figure 3.3: Avg. Response Time (secs), EU-BTree and EU-SDB

faster than the *Basic* protocol. This is due to the fact that it is faster to send log records to the ATOMIC queue using batching than to send each individual log record to the various PU queues (Section 3.6.1). Furthermore, in the atomic protocol of Algorithm 4, sending the log records to the PU queues (Lines 6-9) is done in an asynchronous way so that that is not part of the response time experienced by the end user, but only adds to the *cost* as will be seen in Section 3.8.4. Obviously, *Snapshot* and *Locking* are more expensive than the *Basic* and *Atomic* protocols. However, the difference between *Locking* and *Snapshot* is not as significant as one might expect. That is mainly due to the low concurrency. Both protocols require a lock for each record (either during the transaction or during the validation) and almost the same amount of interactions with the queues to update to a consistent snapshot. However, through better caching generalized snapshot isolation is still slightly faster.

Comparing the client-side B-Tree and the SDB configuration in Figure 3.3, it can be seen that for most protocols the ”SDB” configuration is about twice as fast as the ”BTree” configuration. Using an indexing service saves the time to receive the pages of the B-Tree because the search is done on the server. Furthermore, our implementation packs as many search requests as possible into one request to ”SDB” and caches the returned value to improve the performance. As a matter of course, the difference depends heavily on the data size. The larger the data the more efficient the ”SDB” solution, the smaller the more efficient the ”BTree” solution.

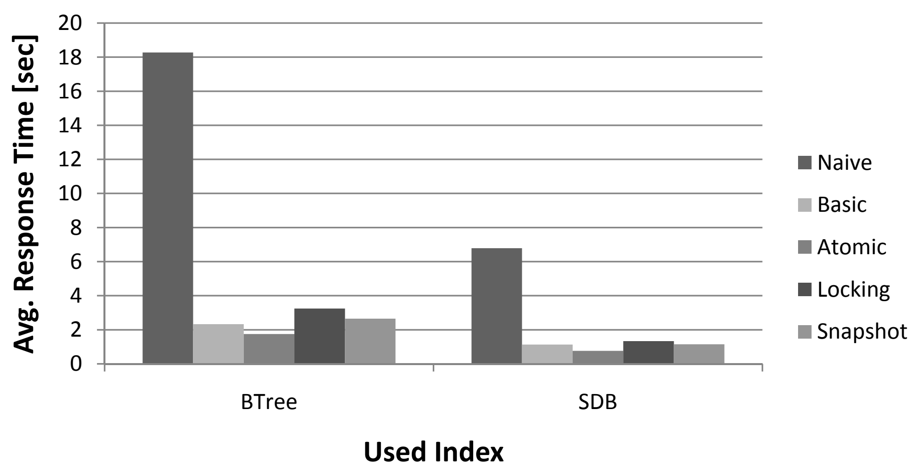

Figure 3.4: Avg. Write Response Time (secs), EU-BTree and EU-SDB, Update WIs Only

3.8.3.2 EC2-BTree and EC2-SDB

Figure 3.5 shows the average response time per WI for the different consistency protocols on the EC2-BTree and EC2-SDB configurations. Among the protocols, the same trends can be observed for the EC2 as for the EU configurations. For the same reason as before the *Naïve* protocol shows the worst performance and the *Atomic* one the best. Here, *Snapshot* almost reaches the performance of *Atomic*. Generalized snapshot isolation uses cached snapshots for read-only transactions and hence, has almost no additional overhead for read transactions; whereas *Locking* always requires bringing the data to the current view. Again, the difference is bigger for read-only transactions than for update interactions. Figure 3.6 again shows the average response times considering update interactions only. There is hardly any difference between *Snapshot* and *Locking* because of the fine-grained locking and the low concurrency. As described before, for update transactions these two require a similar amount of work to ensure the consistency requirement.

In general, the differences between the *Basic*, *Atomic*, *Locking*, and *Snapshot* protocols are smaller for the EC2 configurations than for the EU ones. A major chunk of the response times is caused by the communication between the end-user's machine and the EC2 server; the Amazon-internal communication between an EC2 server and S3 is comparatively fast.

Comparing the B-Tree and SDB configurations in Figure 3.5, the B-Tree one is slightly faster than SDB (except for *Naïve*). Since the communication between EC2 and S3 is fast compared to end-user machine to S3 communication, reading the B-Tree is generally faster. In addition, it provides better caching behavior. By reading the B-Tree, keys

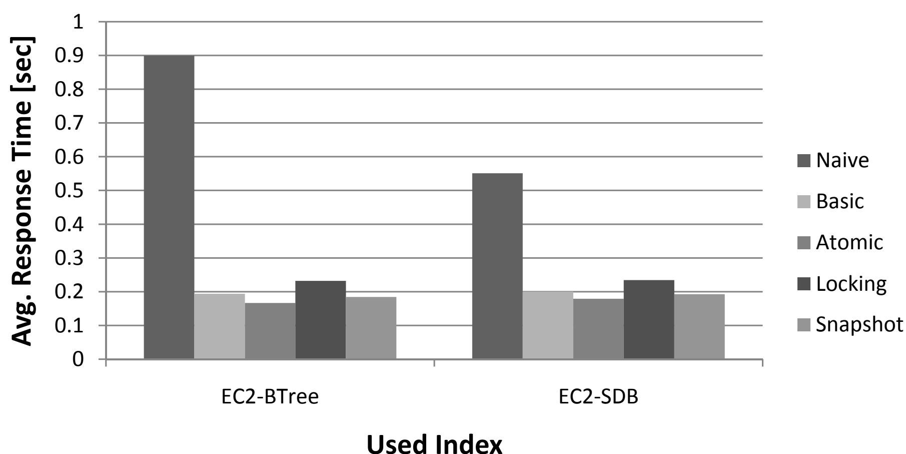

Figure 3.5: Avg. Response Time (secs), EC2-BTree and EC2-SDB

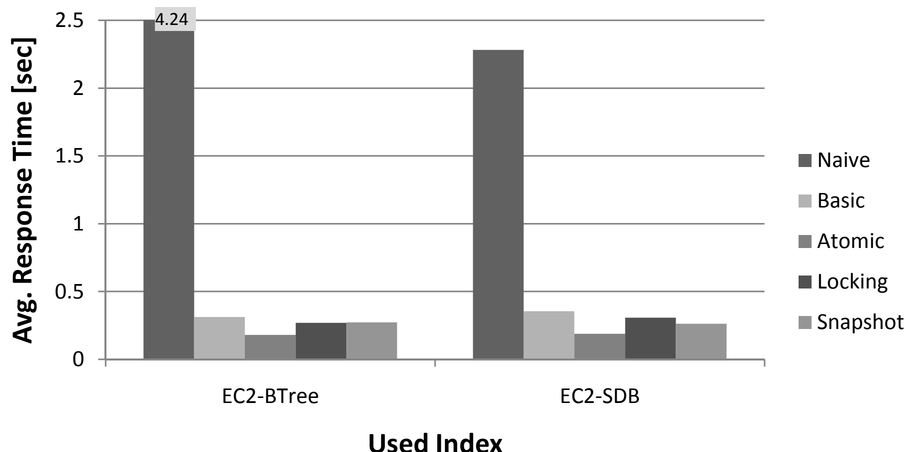

Figure 3.6: Avg. Response Time (secs), EC2-BTree and EC2-SDB, Update WIs Only

other than the search-keys are cached at once when loading a page.

Comparing Figures 3.3 and 3.5, we see that significantly better response times can be achieved with the EC2 than with the EU configurations. Again, this result is not surprising and can easily be explained. Since the EC2 machines are much closer to the data, the EC2 configurations achieve lower response times in the same way as *stored procedures* are more efficient than regular client-server interactions in a traditional DB application scenario. In other words, running the client application stack of Figure 3.1 on EC2 machines is a form of *query shipping* (i.e., moving the application logic to the data).

3.8.4 Cost of Consistency Protocols

3.8.4.1 EU-BTree and EU-SDB

Figure 3.7 shows the average cost per WI for the various protocols in the two EU configurations. Independent of the index, *Basic* is the cheapest protocol, because it requires only a few requests for coordinating commits using cloud services such as Simple queues, Advanced queues, counters, or locks. The *Naïve* protocol is slower, as it requires transferring entire pages between Amazon and the client leading to higher network costs.

The other protocols are more expensive because of their extensive interaction with cloud services in order to achieve higher levels of consistency. Furthermore, in terms of cost, the SDB configuration outperforms the B-Tree configuration. The SDB config-

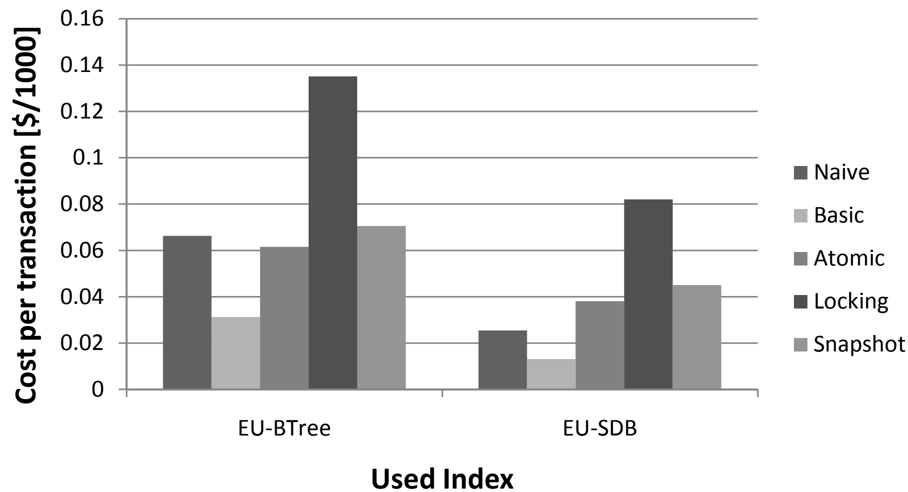

Figure 3.7: Cost per WI (milli-$), EU-BTree and EU-SDB

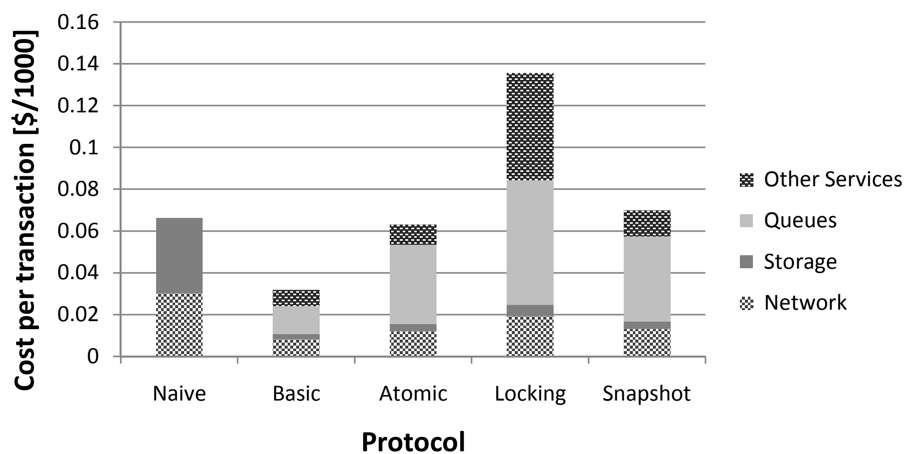

Figure 3.8: Cost per WI (milli-$), EU-BTree Cost factors

uration requires fewer interactions with queues, locks and S3 compared to the B-Tree configuration. This not only reduces the cost for requests but also significantly lowers the cost for the network traffic.

One of the most important findings of this work is that *response time* and *cost* are not necessarily correlated when running a web-based database application in the cloud. This becomes evident when comparing Figure 3.7 with Figure 3.3. The *Atomic* protocol does very well in the response time experiments because most of its work (flushing ATOMIC queues to PU queues and checkpointing) is carried out asynchronously so that the extra work has no impact on the response times of WIs as experienced by users. Nevertheless, this work must be paid for (independently of whether it is executed asynchronously or not). Thus, it is in general not possible to predict cost (in USD) from other performance metrics such as response time or throughput.

Figure 3.8 shows the different cost factors for the client-side B-Tree. Not surprisingly, the only significant cost factors for the *Naïve* protocol are the storage service and the network traffic. For the other protocols, the queue service and the locking service are the main cost drivers. The high share of the locking service for the *Basic* protocol can again be explained by the checkpointing protocol. Because of the low concurrency, few transactions have to share the cost for one checkpoint. The freshness of the data is again the reason for the high cost for *Locking* as compared to *Snapshot* and *Atomic*. Guaranteeing serializability (instead of working mainly on cached data) requires more interactions with the queues and the locking service. Obviously, the results depend heavily on the pricing model. For example, lowering the cost per request for the queues could significantly lower the transaction cost.

For brevity, the breakdown for read and write transactions is not shown. In terms of overall cost, write transactions clearly dominate because of the high fees for all the requests to cloud servers such as queues, counters, and locks, needed in order to orchestrate the updates. Again, this observation contrasts the findings of Section 3.8.3: The overall response times are dominated by the (much more frequent) read requests.

3.8.4.2 EC2-BTree and EC2-SDB

Figure 3.9 shows the cost per WI in milli-dollars for the various consistency protocols in the EC2 configurations. Comparing the consistency protocols we observe a similar pattern as in Figure 3.7. A more interesting observation can be made when comparing

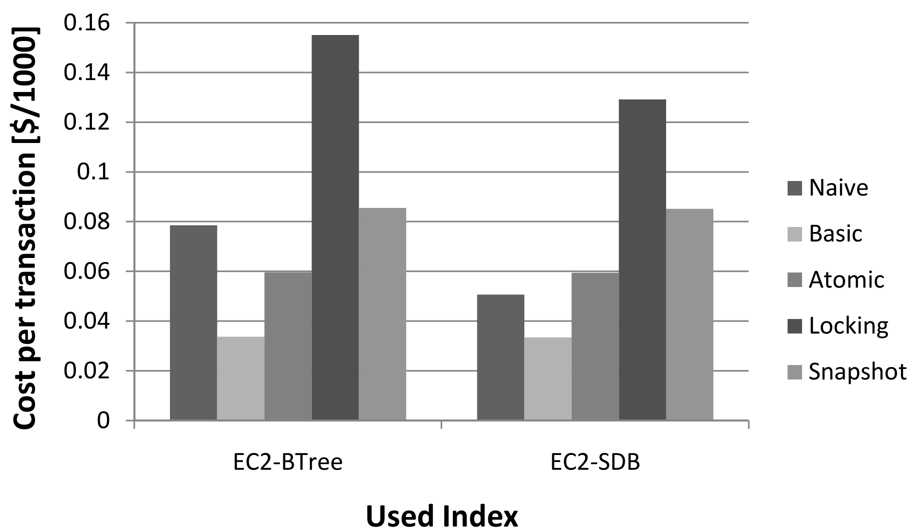

Figure 3.9: Cost per WI (milli-$), EC2-BTree and EC2-SDB

the level of the cost in these two figures: It would be expected that pushing the client application stack on to EC2 servers increases the cost because EC2 server must be paid whereas the end-users' machines are free resources. However, only for the SDB configuration significant cost savings can be realized. A comparison of the different cost factors for the EC2-BTree in Figure 3.10 and EU-BTree in Figure 3.8 reveals that only the *Naïve* protocol profits from the EC2 server. It has a significantly lower response time and hence, also a lower throughput. As a consequence, fewer transactions have to share the cost for EC2 than in the other protocols.

Overall, the cost difference between the EU and EC configuration can mainly be attributed to the network, and not to the EC2 machines. Every WI requires generating and transferring a web-page. This now becomes significantly more expensive than in the client solution, where many WIs are directly answered from cache. Furthermore, pure data transfer causes less traffic than the HTML pages that now need to be transfered. Only *Naïve* and *Locking* do not profit from the the reduced network traffic. The *Naïve* protocol requires the submission of the whole pages for updates, whereas *Locking* has an increased cost factor to keep the pages up-to-date and thus is better hosted in the cloud than executed on the client.

Network traffic is also the reason why in the EC2 configuration the costs of using the SDB index are no longer significantly lower than those for using the client-side BTree. Mostly, the SDB index takes advantage of a reduced network traffic; but as traffic between EC2 and SDB is free, these savings no longer play an important role. On the other hand, for the SDB version, the transfer cost for every web page has now also to be paid.

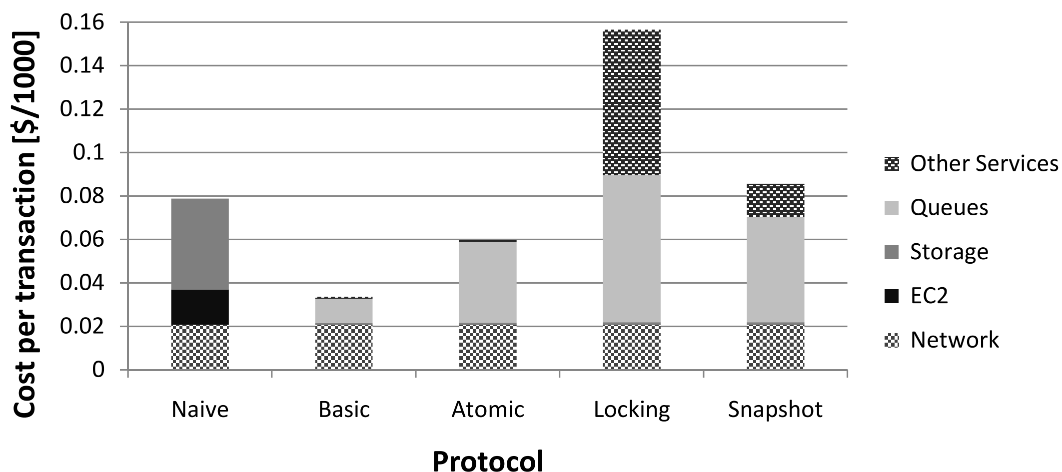

Figure 3.10: Cost per WI (milli-$), EC2-BTree Cost Factors

Except for the *Naïve* protocol, in the here presented scenario, the cost to rent CPU cycles on EC2 servers is not relevant. However, with a smaller number of users and fewer WIs, the cost becomes increasingly important.

Subtracting the network expense from the transaction cost reveals that *Basic* and *Atomicity* are cheaper in the EC2 configuration, whereas *Locking* and *Snapshot Isolation* are more expensive. Furthermore, for most protocols, the ratio between queue requests and the other services shifted. This is an artefact of our particular experiments: As mentioned in Section 3.5, checkpoints are carried out periodically given a specified *checkpoint interval*. In all our experiments, the *checkpoint interval* was set to 45 seconds. Since EC2 runs more TPC-W transactions per second than EU (Section 3.8.3), EC2 runs more TPC-W transactions per checkpoint. In fact, we see this in the shift of the cost factor. The *Basic* protocol in Figure 3.8 faces some non-ignorable cost for the locking service (because of the checkpointing), whereas in the EC2 configuration in Figure 3.10 this factor does not play a significant role since more transactions are issued inside a checkpoint interval. As a result, the cost for checkpointing is distributed over more transactions, and lead to the reduced cost for the *Basic* and *Atomicity* protocols. The reason why *Locking* and *Snapshot Isolation* are still more expensive, is again the increased throughput. More pages are updated, and thus, more requests are necessitated to ensure the freshness of the data and the stronger levels of consistency. This effect overlays the savings gain through the better distribution of the checkpointing cost. It would have been fair to have different checkpoint interval parameter settings and/or to apply individual wait-times for the EC2 and EU configurations in order to reduce the issued request, but, for the sake of uniformity, we have not done this.

In the EC2 configuration, cost in USD are even less predictability based on other performance metrics such as response time or throughput. Here, *Snapshot isolation* is significantly more expensive than *Atomic* although they have almost the same response time. Again, this caused by the additional interactions with services in relation to the checkpointing behavior.

3.8.5 Scalability

To test the scalability of our implementation, we use the EC2 B-Tree configuration. This configuration allows us using the Amazon cluster to increase the number of machines and rely only on S3 and not on other external service, which might become a bottleneck.[19]. We test the scalability by increasing the number of EC2 instances from 4 to 20 servers and issuing as many requests against it as possible. Further, we ensure that the queue, counter and lock service do not become a bottleneck preventing scalability. Currently, our service implementation does not support automatic scaling and load-balancing, so that we have to provision the service. A separate experiment indicated that approximately one EC2 service server providing the locking, queuing and counter service is required for two fully-utilized application servers. Thus, next to the 20 application servers we use 10 additional EC2 instances to provide the required services. All queues, locks and counters are randomly distributed across those servers (see also Section 3.7.4). The Naïve approach was not considered in this experiment as it does not provide the throughput of the other protocols.

The results of the scalability experiment are shown in Figure 3.11. All protocols scale almost linearly. Naturally, the *Basic* protocol has the highest throughput with 3000 WIPS. The protocol only provides eventual consistency and has the lowest overhead. *Atomic*

[19]For example, another experiment has shown that a TPC-W implementation entirely built on top of SimpleDB does not scale beyond 200 WIPS.

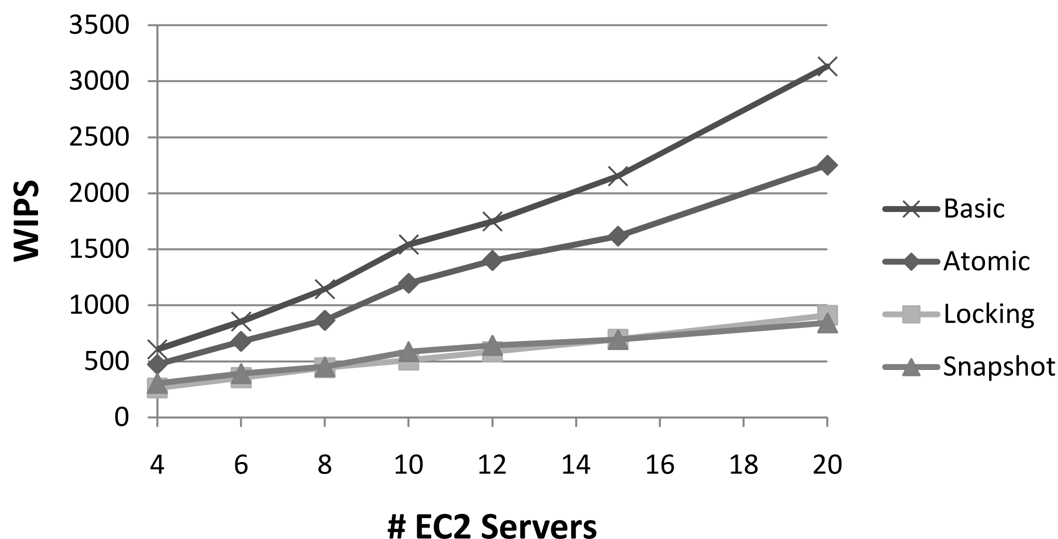

Figure 3.11: Web Interactions per Second (WIPS), Varying Number of EC2 Application Servers

also scales linearly but does not achieve the same throughput as the *Basic* protocol. *Locking* and *Snapshot Isolation* scale almost linearly. The reason for this is that every server only hosts a very limited number of database processes (10) which operate completely independent from each other including separate buffers. Each process executes one transaction at a time, which together with the fine-grained record locking leads to only a few aborts. This way, even the protocols providing stronger levels of consistency are able to scale. However, as more inserts are done and more pages are updated, *Locking* and *Snapshot Isolation* require more recent copies than the *Atomic* and the *Basic* protocols, preventing 100% linear scalability for the strong consistent protocols.

In order to put the performance numbers in relation to other systems: MySQL Replication 5.0.51 with ROWA (read-once, write-all) replication installed on four EC2 Medium-High-CPU instances and additional six Tomcat 6.0.18 application servers running the TPC-W benchmark on the same type of EC2 instances are able to scale up to 450 WIPS before the master becomes the bottleneck. Further, MySQL 5.0.51 Cluster is only able to handle around 250 WIPS with a similar setup. Although the comparison is not completely fair, since our TPC-W implementation is different and we do not use Tomcat, the numbers show the basic potential.[20]

3.8.6 Tuning Parameters

Reconsidering the discussions of Sections 3.5 to 3.7, there are three important tuning parameters:

- *Checkpoint Interval:* Defines the interval in when the PU messages from the queues are written to the page. The lower the value is set, the faster updates become visible.
- *Page Size:* Page size as in traditional database systems.
- *TTL:* Time-to-live is another parameter to control the freshness of the data. It determines the time period for which a page in the cache is valid.

The Checkpoint Interval parameter has already been studied in [BFG$^+$08]. The results of [BFG$^+$08] are directly applicable to the experimental environment used in this work. The Checkpoint Interval is an effective way to trade cost with freshness of data. The remainder of this section studies alternative settings for the other two parameters.

[20]The complete performance study is currently under review for SIGMOD 2010.

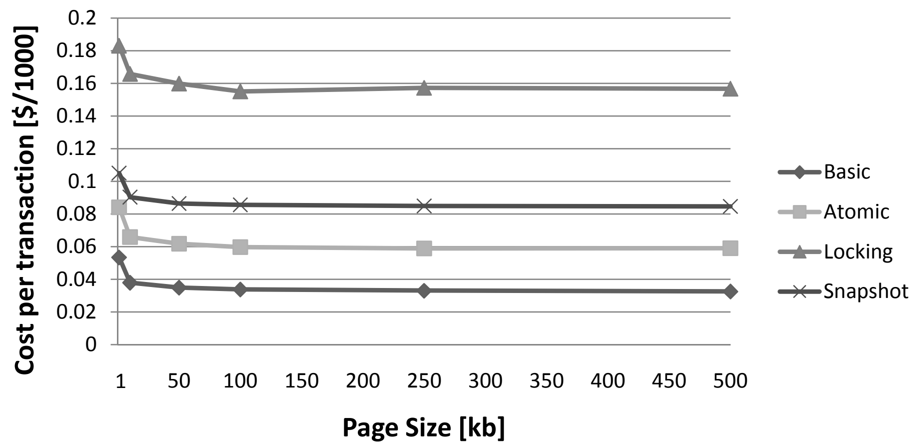

Figure 3.12: Cost per WI (milli-$), EC2-BTree, Varying Page Size

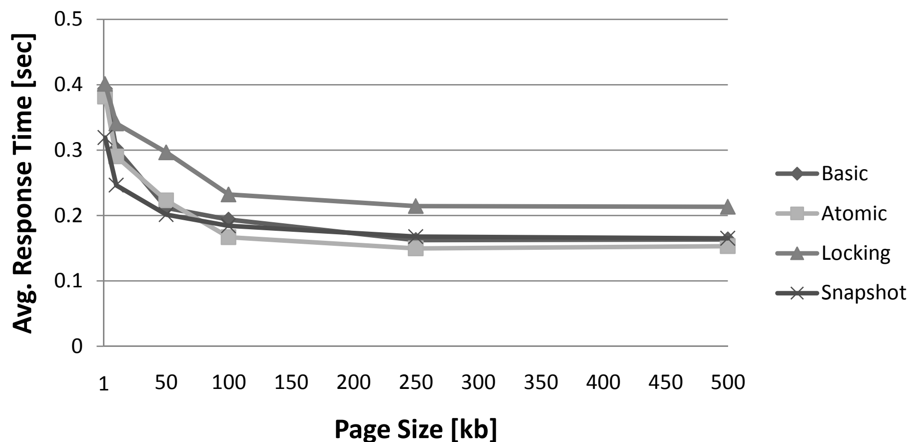

Figure 3.13: Avg. Response Time (secs), EC2-BTree, Varying Page Size

3.8.6.1 Page Size

Figure 3.12 shows the cost per WI for the different protocols in the EC2-BTree configuration with a varying page size (we excluded *Naïve* to improve the presentation). The cost decreases quickly and stabilizes between 50 and 100kb. An increasing page size yields fewer interactions with S3. The higher network traffic does not play a role for the EC2 configuration, as network bandwidth is for free inside AWS.

Figure 3.13 shows the average response time with a varying page size, again in the EC-BTree configuration. The response time drops with increasing page size and eventually stabilizes between 100kb and 500kb. The reasoning is the same as before: A larger page size reduces the number of interactions with S3, but bigger page sizes require longer transfer times. Clearly, the savings depend on the configuration. If SimpleDB

is used as an index, the effect is less pronounced because SimpleDB is probed in the granularity of individual entries. The page size parameter is only applicable to data pages and, hence, less significant for SimpleDB. Further, if the application stack is hosted on the client the difference is even more significant because of the larger transfer times.

3.8.6.2 Time-to-Live

In order to complete the sensitivity analysis on alternative parameter settings, Figures 3.14 and 3.15 show the cost and response times of the various protocols in the EU-BTree configuration with a varying TTL parameter. The TTL parameter controls how

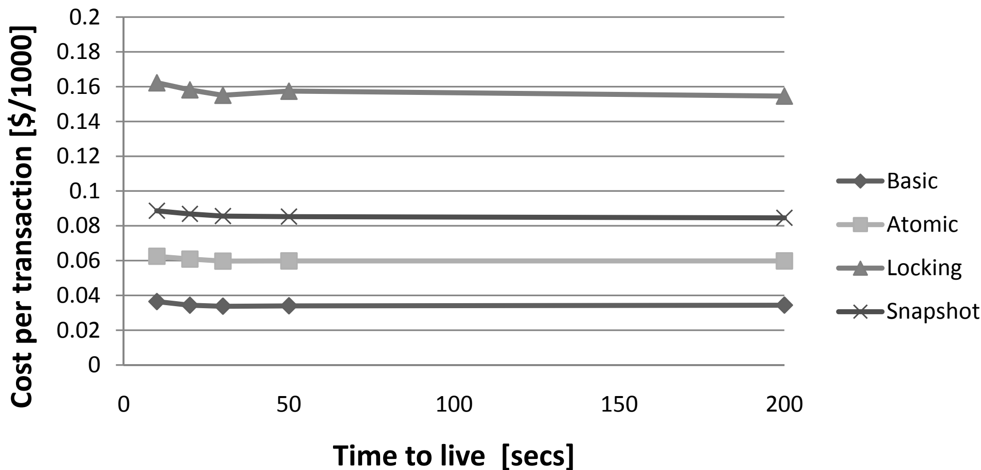

Figure 3.14: Cost per WI (milli-$), EC2-BTree, Varying TTL

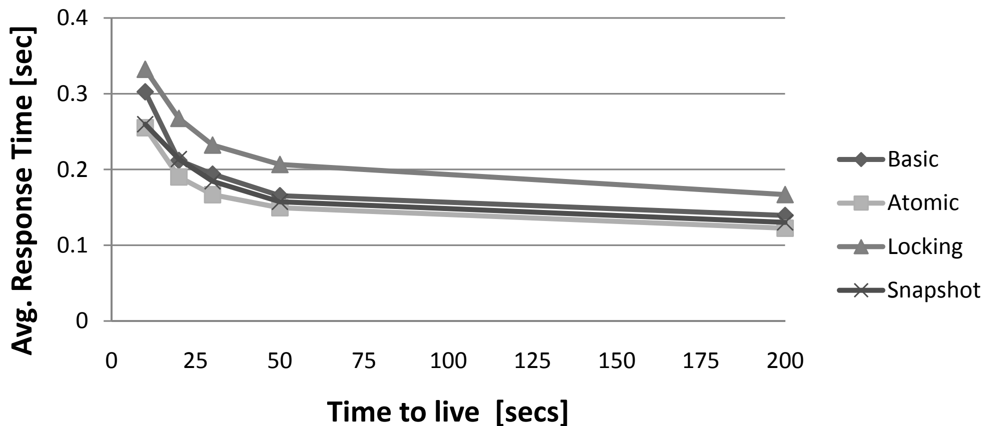

Figure 3.15: Avg. Response Time (secs), EC2-BTree, Varying TTL

long a page can be cached in a client's buffer pool before checking whether a new version of the page is available on S3. While the influence on the cost is rather insignificant (S3 read requests are cheap compared to put request), the response time can be significantly improved by a higher TTL parameter. A higher TTL requires less interaction with S3, which is one of the most performance-critical operations, even if the page is just probed for a change. However, a high TTL parameter may lead to more staleness and thus, also influences the user experience.

3.8.7 Bulk-Loading Times

Table 3.5 shows the different loading times performed by one process from EC2. SimpleDB and the client-side BTree have comparable performance. In both solutions we make use of the offered bulk-loading operation. However, in regard to the cost, the BTree is an order of magnitude cheaper. That is due to the fact that SimpleDB has relatively small bulk-loading chunks, which we need to call several times. Although the latency for every chunk is small inside the Amazon infrastructure, it has an impact on the cost as every request is charged.

Index	*Time [secs]*	*Cost [milli-$]*
EU-BTree	129	7.0
EU-SDB	137	54.5

Table 3.5: Cost (milli-$) and Time (secs) for Bulk-Loading 10000 Products from EC2

3.9 Related Work

This work has mainly been inspired by the recent development of Amazon's Dynamo [DHJ+07] and Google's BigTable [CDG+06] towards ultra-scalable storage systems (see also Section 2.3.2). Both systems provide just *eventual consistency* guarantees. Recently, work has been published on storage services providing stronger guarantees. This includes Yahoo PNUTS [CRS+08] and Google's Megastore [FKL+08]. Both systems are able to work on certain snapshots of data and have some concurrency control mechanism for a restricted group of objects. Unfortunately, both systems are not yet publicly available (Megastore is most likely the service attached to Google's AppEngine, but not exposed as a single service) and therefore cannot be studied in detail here. Another

restriction of those services is, that the consistency is only guaranteed for a smaller set of objects. As this might be sufficient for some applications, for others it might not. The protocols presented in this work do not rely on such assumptions. Nevertheless, studying the possibilities to directly guarantee different consistency levels inside the cloud storage is part of our future work.

In the distributed systems and database literature, many alternative protocols to coordinate reads and writes to (replicated) data stored in a distributed way have been devised. The authoritative references in the DB literature are [BHG87] and, more recently, [WV02]. For distributed systems, [TS06] is the standard textbook which describes the alternative approaches and their trade-offs in terms of consistency and availability. This work is based on [TS06] and applies distributed systems techniques to data management with utility computing and more specifically S3. To our best knowledge, this is the first attempt to do so for S3. The only other work on S3 databases that we are aware of makes S3 the storage module of a centralized MySQL database [Atw07].

Utility computing has been studied since the 90s; e.g., in the OceanStore project at UC Berkeley. Probably, it has seen its biggest success in the scientific community where it is known as *grid computing* [FK04]. Grid computing was designed for very specific purposes; mostly, to run a large number of analysis processes on scientific data. Amazon has brought the idea to the mass. Even S3, however, is only used for specific purposes today: large multi-media objects and backups. The goal of this chapter has been to broaden the scope and the applicability of utility computing to general-purpose web-based applications.

Supporting scalability and churn (i.e., the possibility of failures of nodes at any time) are core design principles of peer-to-peer systems [Hel04]. Peer-to-peer systems also enjoy similar consistency vs. availability trade-offs. We believe that building databases on utility computing such as S3 is more attractive for many applications because it is easier to control security , to control different levels of consistency (e.g., atomicity and monotonic writes), and provide latency guarantees (e.g., an upper bound for all read and write requests). As shown in Figure 3.1, S3 serves as a centralized component which makes it possible to provide all these guarantees. Having a centralized component like S3 is considered to be a “no-no” in the P2P community, but in fact, S3 is a distributed (P2P) system itself and has none of the technical drawbacks of a centralized component. In some sense, this work proposes to establish data management overlays on top of S3 in a similar way as the P2P community proposes creating network overlays on top of the Internet.

Postponing updates is also not a new idea and was for the first time studied in the

context of databases systems in [SL76]. The idea between log queues and differential files is quite similar, although we do not deal with a completely distributed environment.

3.10 Summary

Web-based applications need high scalability and availability at low and predictable cost. No user must ever be blocked either by other users accessing the same data or due to hardware failures at the service provider. Instead, users expect constant and predictable response times when interacting with a web-based service. Utility computing (aka cloud computing) has the potential to meet all these requirements. Cloud computing was initially designed for specific workloads. This chapter showed the opportunities and limitations to apply cloud computing to general-purpose workloads, using AWS and in particular S3 for storage as an example. The chapter showed how the *textbook* architecture to build database systems can be applied in order to build a cloud database system. Furthermore, the chapter presented several alternative consistency protocols which preserve the design philosophy of cloud computing and trade cost and availability for a higher level of consistency. Finally, an important contribution of this chapter was to study alternative client-server and indexing architectures to effect applications and index look-ups.

The experimental results showed that cloud computing and in particular the current offerings of providers such as Amazon are not attractive for high-performance transaction processing if strong consistency is important; such application scenarios are still best supported by conventional database systems. Furthermore, while indeed virtually infinite scalability and throughputs can be achieved, the cost for performance (i.e., $ per WIPS in the TPC-W benchmark) is not competitive as traditional implementations that are geared towards a certain workload. Cloud computing works best and is most cost-effective if the workload is hard to predict and varies significantly because cloud computing allows to provision hardware resources on demand in a fine-grained manner. In traditional database architectures, the hardware resources must be provisioned for the expected peak performance which is often orders of magnitudes higher than the average performance requirements (and possibly even the real peak performance requirements).

In summary, we believe that cloud computing is a viable candidate for many Web 2.0 and interactive applications. Given the current trends and increased competition on the cloud computing provider market, we expect that cloud computing will become more attractive in the future.

From our point of view, this work is still the beginning towards the long-term vision to implement full-fledged database systems on top of cloud computing services. Clearly, there are many database-specific issues that still need to be addressed. There are still a number of optimization techniques conceivable in order to reduce the latency of applications (e.g., caching and scheduling techniques). Furthermore, query processing techniques (e.g., join algorithms and query optimization techniques) and new algorithms to, say, bulk-load a database, create indexes, and drop a whole collection need to be devised. For instance, there is no way to carry out chained I/O in order to scan through several pages on S3; this observation should impact the design of new database algorithms for storage services. Furthermore, building the right security infrastructure will be crucial for the success of an information system in the cloud.

Finally, the here presented architecture and protocols became part of the Sausalito product developed by 28msec Inc. Sausalito is an XQuery application platform deployed on top of the infrastructure services from Amazon. Sausalito combines the application server with the database stack similar to the EC2 client-server configuration presented in this chapter. The programming language for all layers is XQuery (compare Section 2.4).

Chapter 4

Consistency Rationing

4.1 Introduction

The previous chapter has shown how stronger levels of consistency and transactions can be implemented on top of a cloud storage service. However, choosing the right level of consistency for an application is a complicated task. It can not only influence the performance but also significantly impact the cost. This chapter proposes a framework on how to set the consistency level for an application.

The key observation behind the work reported in this chapter is that not all data needs to be treated at the same level of consistency. For instance, in a web shop, credit card and account balance information naturally require higher consistency levels, whereas user preferences (e.g., "users who bought this item also bought..." data) can be handled at lower consistency levels. The previous chapter has shown that the price of a particular consistency level can be measured in terms of the number of service calls needed to enforce it. Since increasing levels of consistency require more calls, higher levels of consistency (significantly) increase the costs per operation.[1] Similarly to the cost of consistency, the price of inconsistency can be measured in terms of the percentage of incorrect operations that are caused by using lower levels of consistency.

[1] This observation also holds for more traditional architectures (e.g., a database clusters) in the absence of a perfect partitioning scheme. Again, perfect partitioning refers to a partitioning scheme where all possible transactions can be executed on a single server of the system. As perfect partitioning is hard if not impossible to find for the majority of web scenarios, strong consistency even with rather traditional protocols (e.g., 2-Phase-Locking, or snapshot isolation with 2-phase-commit) requires more overhead compared to relaxed consistency models (e.g., eventual consistency). As the overhead implies message passing between physical machines, it can be assumed, that stronger levels of consistency significantly increase the latency and the costs.

This percentage can often be mapped to an exact cost in monetary terms (e.g., the penalty costs of compensating a booking error or of losing a customer). These figures are typically well-known to the companies offering such services.

To find the right balance between cost, consistency, and availability is not a trivial task. In large scale systems, high consistency implies high cost per transaction and reduced availability (Section 3.6 and 3.8.4) but avoids penalty costs. Low consistency leads to lower costs per operation but might result in higher penalty costs (e.g., overselling of products in a web shop). To make matters more complicated, this balance depends on several factors, including the application semantics. In this chapter, we propose to bypass this dilemma by using a dynamic consistency strategy: reduce the consistency requirements when possible (i.e., the penalty cost is low) and raise them when it matters (i.e., the penalty costs would be too high). The adaptation is driven by a cost model and different strategies that dictate how the system should behave. We call this approach *Consistency Rationing* in analogy to *Inventory Rationing* [SPP98]. Inventory rationing is a strategy for inventory control where inventories are monitored with varying precision depending on the value of the items. Following this idea, we divide the data into three categories (A, B, and C), and treat each category differently depending on the consistency level provided.

The *A category* contains data for which a consistency violation would result in large penalty costs. The *C category* contains data for which (temporary) inconsistency is acceptable (i.e., no or low penalty cost exists; or no real inconsistencies occur). The *B category* comprises all the data where the consistency requirements vary over time depending on, for example, the actual availability of an item. This is typically data that is modified concurrently by many users and that often is a bottleneck in the system. In this chapter, we focus on the B category. It is in this category where we can make a significant trade-off between the cost per operation and the consistency level provided. We describe several use cases motivating such B category data. Then we develop a cost model for the system and discuss different strategies for dynamically varying the consistency levels depending on the potential penalty costs. The goal of these strategies is to minimize the overall cost of operating over the cloud storage. Our experiments on a cloud database service implemented on top Amazon's S3 indicate that significant cost benefits can be obtained from the dynamic policies we introduce here.

4.1.1 Contributions

This chapter makes the following contributions:

- We introduce the concept of *Consistency Rationing*, which allows applications to obtain the necessary levels of consistency at the lowest possible cost.
- We define and analyze a number of policies to switch consistency protocols at run-time. Our experiments show that dynamically adapting the consistency outperforms statically assigned consistency guarantees.
- We introduce the notion of probabilistic guarantees for consistency (i.e., a percentile) using temporal statistics for numerical and non-numerical values. Such statistical guarantees are very important in terms of service level agreements although, to our knowledge, this is the first time that probabilistic consistency guarantees are studied in detail.
- We present a complete implementation of Consistency Rationing on top of Amazon's S3. We report on the cost ($) and performance of running the TPC-W benchmark [Cou02] at several consistency categories, mixes of categories, and different policies of the B category. The results of the experiments provide important insights on the cost of running such systems, on the cost structure of each operation, and on how to optimize these costs using appropriate costs models.

4.1.2 Outline

The remainder of this chapter is organized as follows: Section 4.2 describes several use cases for Consistency Rationing in the cloud. Section 4.3 describes Consistency Rationing, the ABC analysis and how guarantees mix. Section 4.4 presents a set of alternative strategies to handle B data. Section 4.5 covers the implementation of Consistency Rationing in the cloud. Section 4.6 summarizes the results of our experiments using the TPC-W benchmark. Section 4.7 discusses related work and Section 4.8 concludes this chapter.

4.2 Use Cases

The need for different consistency levels is easily identifiable in a variety of applications and has already been studied in different contexts (e.g., in [CRS+08, YV00]). In the following, we present potential use cases in which Consistency Rationing could be applied.

Web Shop Assume a conventional web shop built on top of a cloud storage service. A typical web shop stores different kinds of data [Vog07]. There is, for example, data of customer profiles and credit card information, data about the products sold, and records on user's preferences (e.g., "users who bought this item also bought . . . ") as well as logging information. These examples already indicate that there are different categories of data in terms of value and need for consistency. The customer's credit card information and the price of the items must be handled carefully. Buyer preferences and logging information could even be lost without any serious damage (e.g., if the system crashes and cached data had not been made persistent).

The designer of such a web shop application could use Consistency Rationing as follows: (1) account information is categorized as *A data* accessible under strong consistency guarantees (i.e., serializability). (2) Product inventory data is categorized as *B data*. As long as the available stock is high enough, the system tolerates some inconsistencies. When the available inventory drops below a certain threshold, access should be done only under strong consistency guarantees to avoid overselling. (3) Buying preferences and logging information is classified as *C data*. The system does not need to see the most up-to-date state at all times and does not need to take any actions to grant exclusive accesses in case of updates.

Since the cost of running such a web shop in the cloud is determined by the cost per operation, applying Consistency Rationing will necessarily reduce the overall cost by using cheaper operations when possible while at the same time minimizing the cost caused by inconsistencies.

Ticket reservations for movie theaters or operas, as well as flight booking systems of an airline company follow the same operational model as the web shop. The only difference is that the cost of inconsistencies (i.e., overbooking or losing tickets) can be significantly more expensive than in a web shop. The advantage of Consistency Rationing is that it allows designers to adjust the cost function and adaptation strategies to optimize the total operational costs. In our approach, transactions may span multiple categories and are not restricted to operate within a single category (Section 4.3.4).

This allows to partition (ration) the data using any partitioning scheme to optimize the total costs.

Auction System Typical for an online auction system is that an item to be auctioned starts to become very popular in the final stages of its auction. The last minutes of an auction are usually of the highest interest for a bidder. During this period of time, the item's current price should always be up-to-date and modified under strong consistency guarantees. When the end of an item's auction is several days ahead, bids can be updated using lower consistency levels without any effect on the system.

An auction system may utilize this information to implement different strategies for bringing an item up-to-date. If the end of the auction is in the near future (e.g., in one or two hours), the item is treated with strong consistency. Conversely, if the end of the auction lies further away, the item is treated with lower guarantees.

The difference between the auction system and the web shop use case is that the selection of the consistency protocol is done on a time basis instead of on a value basis. Here, the *time* is used as a threshold that determines which consistency guarantees to select. In contrast, the web shop example used the *value* of a data item as the threshold.

Collaborative Editing Collaborative Editing allows people to work simultaneously on the same document or source base (e.g., Google Docs, Version control, Wiki's). The main functionality of such a system is to detect conflicts during editing and to track the history of changes. Traditionally, such systems work with strong consistency. If the system detects a conflict, the user is usually required to resolve the conflict. Only after resolving the conflict the user is able to submit the change as long as no other conflict has been generated in the meantime. Although in real deployments there might be some parts of the document that are updated frequently by different parties (e.g., the citations of a paper), people tend to organize themselves upfront to avoid conflicts from the beginning. Hence, conflicts are unlikely for most parts of the document and no concurrency control is required. In contrast, those parts which are frequently updated by several parties, would best be handled by strong consistency guarantees to avoid conflicts all together.

Other than in the previous examples, the selection of the consistency protocol is based on the likelihood of conflicts and not on the value or time. Our *General policy* described in Section 4.4.1 addresses this use case by automatically adopting the strategy based on the update frequency.

4.3 Consistency Rationing

The use cases in Section 4.2 indicate that not all data needs the same consistency guarantees. This fact is well-known in standard database applications and addressed by offering different consistency levels on a per-transaction basis. Although it is possible to relax every ACID property, in this work we focus on Isolation and Consistency, and assume that Atomicity and Durability are given.

There is a great deal of work on relaxing consistency guarantees, both in distributed systems (e.g., eventual consistency, read-write monotonicity, or session consistency [TS06]) and transactions (e.g., read committed, read uncommitted, serializability, or (generalized) snapshot isolation [BBG+95]). The main lesson learned from all this work is that the challenge in relaxed consistency models is to provide the best possible cost/benefit ratio while still providing understandable behavior to the developer. With this in mind, we consider only two levels of consistency (session consistency, and serializability) and divide the data into three categories.

4.3.1 Category C - Session Consistency

The C category encompasses data under *session consistency*. Session consistency has been identified as the minimum consistency level in a distributed setting that does not result in excessive complexity for the application developer [TS06]. Below session consistency, the application does not see its own updates and may get inconsistent data from consecutive accesses.

Clients connect to the system in the context of a session. As long as the session lasts, the system guarantees *read-your-own-writes monotonicity*. The monotonicity guarantees do not span sessions. If a session terminates, a new session may not immediately see the writes of a previous session. Sessions of different clients will not always see each other's updates. After some time (and without failures), the system converges and becomes consistent (see also Section 3.6.2). Further, we assume that inconsistencies because of client crashes should always be avoided (e.g., half applied transactions) and thus, always assume Atomicity properties.

Conflict resolution in the C category for concurrent updates depends on the type of update. For non-commutative updates (e.g., overrides), the last update wins. For commutative updates (numerical operations, e.g., add), the conflict is resolved by applying the updates one after each other. Nevertheless, both approaches can lead to inconsistencies if, for example, the update is dropped or an integrity constraint is violated.

Session consistency is cheap with respect to both, transaction cost as well as response time, because fewer messages are required than for strong consistency guarantees such as serializability. It also permits extensive caching which lowers cost even further and increases performance. Cloud databases should always place data in the C category if inconsistencies cannot occur (e.g., data is never accessed by more than one transaction at a time) or there is neither monetary nor administrative cost when temporary inconsistencies arise.

4.3.2 Category A - Serializable

The A category provides *serializability* in the traditional transactional sense. Data in this category always stays consistent and all transactions that modify data in the A category are isolated. In cloud storage, enforcing serializability is expensive both in monetary costs as well as in terms of performance. These overheads in cost and performance exist because of the more complex protocols needed to ensure serializability in a highly distributed environment (see Section 3.6.4). These protocols require more interaction with additional services (e.g., lock services, queueing services) which results in higher cost and lower performance (response times) compared to ensuring session consistency.

Data should be put in the A category if consistency as well as an up-to-date view is a must. We provide serializability using a pessimistic concurrency protocol (2-Phase-Locking). We choose serializability over, for example, snapshot isolation to ensure transactions always see the up-to-date state of the database. Again, the protocol should also provide Atomicity guarantees.

4.3.3 Category B - Adaptive

Between the data with session consistency (C) and the data with serializability (A) guarantees, there exists a wide spectrum of data types and applications for which the required level of consistency depends on the specific situation. Sometimes strong consistency is needed, sometimes it can be relaxed. Given the double impact of transactional consistency in cloud database settings (cost and performance), we introduce the B category to capture all the data with variable consistency requirements.

It is also the case that for many of the applications that would run in a cloud database setting, the price of inconsistencies can be quantified (see the use cases above). Ex-

amples include: refunds to customers for wrongly delivered items, overheads for overbooking, costs of canceling orders, etc. Cost in this sense can refer to either cost in actual money, or e.g., reduced user experience or the administrative overhead to resolve a conflict. Because not all the updates to B data automatically result in a conflict (Section 4.2), a non-trivial trade-off exists between the penalty cost for inconsistencies and the advantages of using relaxed consistency.

In our implementation, data in the B category switches between session consistency and serializability at run-time. If it happens that one transaction operates at session consistency and another transaction operates under serializability for the same B data record, the overall result is session consistency.

In the remainder of this chapter we analyze in detail different policies to switch between the two possible consistency guarantees. These policies are designed to switch automatically and dynamically, thereby reducing costs for the user of the cloud infrastructure both in terms of transaction costs as well as in terms of the costs of inconsistencies.

vspace*12pt

4.3.4 Category Mixes

Our cloud database infrastructure provides consistency guarantees on the data rather than on transactions. The motivation is that data has different characteristics in terms of cost. For example, bank accounts and product stocks require isolation; logging data or customer profiles do not. Defining consistency on data, rather than on transactions, allows handling the data according to its importance. A side effect of this approach, however, is that transactions may see different consistency levels as they access different data.

If a single transaction processes data from different categories, every record touched in a transaction is handled according to the category guarantees of the record. Therefore, operations on A data read from a consistent view and modifications will retain a consistent state. Reads from A data will always be up-to-date. Reads from C data might be outdated depending on caching effects. As a logical consequence, the result of joins, unions, and any other operations between A and C data provide only C guarantees for that operation. In most situations, this does no harm and is the expected/desirable behavior. For example, a join between account balances (A data) and customer profiles (C data) will contain all up-to-date balance information but might contain old customer addresses.

If it is necessary from the application point of view, transactions are allowed to specify which guarantees they need. This allows a transaction to see the most up-to-date state of a record. However, it does not guarantee that the transaction has the exclusive right to update the record. If one transaction writes under session consistency and another under serializibility, inconsistency can still arise. Techniques to actively inform all clients of a change in the consistency model (e.g. by setting a flag in the collection) are beyond the scope of this work.

4.3.5 Development Model

Consistency Rationing introduces additional complexity into the development process of applications running on cloud storage. First, the data must be rationed into consistency categories. This process is driven by the operational costs of transactions and of inconsistencies. Second, the required consistency must be specified at the collection (i.e., relation) level together with the policy and all integrity constraints. This can be done by annotating the schema similar to [YG09].

We envision that the development process of an application and Consistency Rationing can be split into different processes. During the development process, strong consistency guarantees are assumed. The programmer will follow the usual database programming model of explicitly stating transactions. Independent of the categorization, the programmer will always issue a *start_transaction* command at the beginning of a transaction and a *commit_transaction* command at its end. When the application gets deployed, the data is rationed according to cost. The rationing may be done by a person from outside the development department. Of course, the assumption is that this split of development and rationing does not affect the correctness of the system. Which properties an application has to fulfil in order to split the development process is out of the scope of this thesis and part of future work.

4.4 Adaptive Policies

In this section, we present five different policies to adapt the consistency guarantees provided for individual data items in category B. The adaptation consists in all cases of switching between serializability (category A) and session consistency (category C). The policies differ on how they determine the necessity of a switch. The *General policy* looks into the probability of conflict on a given data item and switches to serializability

if this probability is high enough. The *Time policy* switches between guarantee levels based on time, typically running at session consistency until a given point in time and then switching to serializability. These two first policies can be applied to any data item, regardless of its type. For the very common case of numeric values (e.g., prices, inventories, supply reserves), we consider three additional policies. The *Fixed threshold policy* switches guarantee levels depending on the absolute value of the data item. Since this policy depends on a fixed threshold that might be difficult to define, the remaining two policies use more flexible thresholds. The *Demarcation policy* considers relative values with respect to a global threshold while the *Dynamic policy* adapts the idea of the *General policy* for numerical data by both analyzing the update frequency and the actual values of items.

4.4.1 General Policy

The General policy works on the basis of a conflict probability. By observing the access frequency to data items, it is possible to calculate the probability that conflicting accesses will occur. Higher consistency levels need to be provided only when the probability of conflict is high enough.

4.4.1.1 Model

We assume a distributed setup with n servers (i.e., threads are considered to be separate servers) implementing the different levels of consistency described in Chapter 3. Servers cache data with a cache interval (i.e., time-to-live) CI. Within that interval, C data is read from the cache without synchronizing. Furthermore, two updates to the same data item are always considered as a conflict (we use no semantic information on the operations). If we further assume that all servers behave similarly (i.e., updates are equally distributed among the servers and independent from each other), the probability of a conflicting update on a record is given by:

$$P_c(X) = \underbrace{P(X > 1)}_{(i)} - \underbrace{\sum_{k=2}^{\infty} \left(P\left(X = k\right) \left(\frac{1}{n}\right)^{k-1} \right)}_{(ii)} \tag{4.1}$$

X is a stochastic variable corresponding to the number of updates to the same record within the cache interval CI. $P(X > 1)$ is the probability of more than one update of

the same record in one cache interval CI. However, a conflict can only arise if the updates are issued on different servers. Hence, the remaining part (ii) of the equation calculates the probability that the concurrent updates happen on the same server and subtracts this from the probability of more than one update. The equation does not consider the probability of two simultaneous conflicts on the same record. This is because we assume that conflicts can be detected and corrected (e.g., by simply dropping conflicting updates) and that the probability of two conflicts on the same record during the time it takes to detect a conflict (e.g., the cache-interval) is negligible.

Similar to [TM96], we assume that the arrival of transactions is a Poisson process, so that we can rewrite the equation (4.1) around a single variable with mean arrival rate λ. Since the probability density function (PDF) of a Poisson distribution is given by:

$$P_\lambda (X = k) = \frac{\lambda^k}{k!} e^{-\lambda} \tag{4.2}$$

Equation (4.1) can be rewritten as:

$$P_c(X) = \underbrace{\left(1 - e^{-\lambda}(1+\lambda)\right)}_{(iii)} - \underbrace{\sum_{k=2}^{\infty} \left(\frac{\lambda^k}{k!} e^{-\lambda} \left(\frac{1}{n}\right)^{k-1} \right)}_{(iv)} \tag{4.3}$$

If $n > 1$ and if the probability of a conflict is supposed to be rather small (e.g., 1% or less), the second term (iv) can be ignored (simulations show that the terms for $k > 3$ are negligible). Hence, the following expression can be considered an upper bound for the probability of a conflict:

$$P_C(X) = \left(1 - e^{-\lambda}(1+\lambda)\right) \tag{4.4}$$

If more precision is needed, the first one or two summands of (iv) can also be taken into account.

4.4.1.2 Temporal Statistics

To calculate the likelihood of a conflict at run-time without requiring a centralized service, every server gathers temporal statistics about the requests. We use a sliding window with size w and sliding factor δ. The window size defines how many intervals a window contains. The sliding factor specifies the granularity at which the window moves. For every time window, the number of updates to each B data item is collected. All complete intervals of a window build a histogram of the updates. Hence, the window

size acts as a smoothing factor. The larger the window size, the better are the statistics and the longer the time to adapt to changes in arrival rates. The sliding factor affects the granularity of the histogram. For simplicity, the sliding factor δ is assumed to be a multiple of the cache interval CI. To derive the arrival rate λ for the whole system from the local statistics, it is sufficient to calculate the arithmetic mean $\bar{x}$ of the number of updates to a record and multiply it by the number of servers n (which is assumed to be globally known) divided by the sliding factor δ:

$$\lambda = \frac{\bar{x}n}{\delta} \tag{4.5}$$

As the statistics are gathered using a sliding window, the system is able to dynamically adapt to changes in the access patterns.

As the update rate of an item has to be small for handling it as a category C item, local statistics can easily mislead the statistics for small window sizes. To overcome the problem, the local statistics can be combined into a centralized view. The simplest way to achieve this would be to broadcast the statistics from time to time to all other servers. Furthermore, if the record itself carries its statistical information (see Section 4.5), even the broadcast is for free. Thus, by attaching the information to the record, the statistical information can be collected when a data item is cached.

4.4.1.3 Setting the Adaptive Threshold

When to switch between consistency levels is a critical aspect of the approach as it affects both costs and correctness.

Let C_x be the cost of an update to a record in category x. This cost reflects only the additional cost per record in a running transaction without the setup cost of a transaction. Let C_O be the cost of consistency violations. A record should be handled with weak consistency only if the expected savings of using weak consistency is higher than the expected cost of inconsistency $E_O(X)$:

$$C_A - C_C > E_O(X) \tag{4.6}$$

If $C_A - C_C > E_O(X)$ then the record should be handled with session consistency (C data). If $C_A - C_C < E_O(X)$, the record should be handled with strong consistency (A data). Assuming $E_O(X) = P_C(X) * C_O$, a record should be handled with weak consistency if the probability of conflict is less then $(C_A - C_C)/C_O$:

$$P_C(X) < \frac{C_A - C_C}{C_O} \tag{4.7}$$

The same equation can be used for optimizing parameters other than cost. For example, if the throughput of a system is to be optimized and we assume that resolving conflicts reduces performance, the same formula can be used by substituting the costs with performance metrics.

A key aspect of the General policy is that a user can simply specify either a fixed probability of inconsistency or provide a cost function independently of what cost means in the particular case. The rest is handled automatically by the system. Consistency, in this sense, becomes a probabilistic guarantee. The probability of inconsistencies will be adjusted depending on how valuable consistency is for a user.

4.4.2 Time Policies

The time policies are based on a timestamp that, when reached, indicates that the consistency guarantees must change. All such use cases tend to follow an auction-like pattern raising consistency levels when a deadline approaches.

The simplest of all time policies is to set a pre-defined value (e.g., 5 minutes). Up to 5 minutes before the deadline, the data is handled with session consistency only. Afterwards, the consistency level switches to strong consistency. Hence, this policy is the same as the Fixed threshold policy below, except that the decision when to switch consistency guarantees is time-based instead of value-based.

As before, defining this threshold is critical. Similar to the General policy, we can define the likelihood of a conflict $P_c(X_T)$; only this time the stochastic variable X_T changes with respect to time t.

In this context, it is often not meaningful to gather statistics for a record. For example, in the case of an auction, a record sees that the number of accesses increase as the deadline approaches and then drops to zero once the deadline is reached. Based on this, a simple method to set the threshold is to analyze a sample set of past auctions and derive the likelihood of a conflict in minute t before the time expires. More advanced methods can be adopted from inventory management or marketing, as similar methods are used to predict the life-cycle of products. However, such methods are beyond the scope of this thesis.

4.4.3 Policies for Numeric Types

In many use cases, most of the conflicting updates cluster around numerical values, such as the stock of items in a store, the available tickets in a reservation system, or the account balance in a banking system. These scenarios are often characterized by an integrity constraint defined as a limit (e.g., the stock has to be equal or above 0) and commutative updates to the data (e.g., add, subtract). These characteristics allow us to further optimize the General policy by considering the actual update values to decide which consistency level to enforce. This can be done with one of the following three policies. The *Fixed threshold policy* looks at the actual value of an item and compares it to a threshold. The *Demarcation policy* applies the idea of the Demarcation protocol [BGM94] and considers a wider range of values to make a decision. Finally, the *Dynamic policy* extends the conflict model of the General policy to numeric types.

4.4.3.1 Fixed Threshold Policy

The Fixed threshold policy defines that if the value of a record is below a certain threshold, the record is handled under strong consistency guarantees. Thus, a transaction that wants to subtract an amount Δ from a record, applies strong consistency if the current value v minus Δ is less than or equal to the threshold T:

$$v - \Delta \leq T \tag{4.8}$$

In comparison to the General policy, the Fixed threshold policy does not assume that updates on different servers conflict. Updates are commutative and can, therefore, be correctly resolved. Nevertheless, inconsistency can occur if the sum of all updates on different servers lets the value under watch drop below the limit.

Similar to finding the optimal probability in the General policy, the threshold T can be optimized. A simple way of finding the optimal threshold is to experimentally determine it over time, i.e., by adjusting T until the balance between run-time cost and penalty cost is achieved. To find a good starting point for T, one can always consider the statistics from the sales department in the company. Normally, the sales department applies similar methods to determine prices or to control the inventory.

The biggest drawback of the Fixed threshold policy is the static threshold. If the demand for a product changes or if *hot spot* products are present, the Fixed threshold policy behaves sub-optimally (see Experiment 2 in Section 4.6).

4.4.3.2 Demarcation Policy

The Demarcation protocol [BGM94] was originally proposed for replicated systems. The idea of the protocol is to assign a certain amount of the value (e.g., the stock) to every server with the overall amount being distributed across the servers. Every server is allowed to change its local value as long as it does not exceed a local bound. The bound ensures global consistency without requiring the servers to communicate with each other. If the bound is to be violated, a server must request additional shares from other servers or synchronize with others to adjust the bound. This protocol ensures that the overall value never drops below a threshold and that coordination occurs only when needed.

We can adopt the basic idea behind the Demarcation protocol as follows. Every server gets a certain share of the value to use without locking. In the following we assume, without loss of generality, that the value of the limit for every record is zero. If n is the number of servers and v the value (e.g., the stock of a product), we define the share that a server can use without strong consistency as $\left\lfloor \frac{v}{n} \right\rfloor$. Hence, the threshold T is defined as:

$$T = v - \left\lfloor \frac{v}{n} \right\rfloor \tag{4.9}$$

All servers are forced to use strong consistency only if they want to use more than their assigned share. By applying strong consistency, a server sees the current up-to-date value and inconsistencies are avoided. As long as all servers behave similarly and decrease the value in a similar manner, this method will ensure that the threshold will not fall below zero. In our context of cloud services, the Demarcation policy might not always ensure proper consistency because it is assumed that servers cannot communicate between each other. Only after a certain time interval (i.e., the cache interval), a record is brought up-to-date. It can happen that a server exceeds the threshold and continues to, for example, sell items (now in locking mode) while another server sells items up to its threshold without locking. In such situations, the value can drop below the established bound. The idea behind the Demarcation protocol is nevertheless quite appealing and we can assume that such scenarios occur only rarely.

An additional drawback of the Demarcation policy is, that for a large number of servers n, the threshold tends towards v and the Demarcation policy will treat all B data as A data. Thus, almost all transactions will require locking. Skewed distributions of data accesses are also problematic: if an item is rarely or unevenly used, the threshold could be lowered without increasing the penalty cost.

4.4.3.3 Dynamic Policy

The Dynamic policy implements probabilistic consistency guarantees by adjusting the threshold.

Model As for the General policy we fix a cache interval CI and assume that updates are equally distributed among servers. Hence, the probability of the value of a record dropping below zero can be written as:

$$P_C(Y) = P(T - Y < 0) \tag{4.10}$$

Y is a stochastic variable corresponding to the sum of update values within the cache interval CI. That is, Y differs from X of Equation 4.1 in that it does not reflect the number of updates but the sum of the values of all updates inside a cache interval. $P(T - Y < 0)$ describes the probability that the consistency constraint is violated (e.g., by buying more items before the servers apply strong consistency).

Temporal Statistics To gather the statistics for Y we again use a window with size w and sliding factor δ. Unlike for the General policy, we assume that the sliding factor δ is a factor of the checkpoint interval CI rather than a multiple. This has two reasons: First, the Dynamic policy requires the variance which can be more precisely derived with smaller sliding factors. Second, the policy concentrates on hot spots and not on rarely updated values. Events (i.e., updates) are not rare and hence, the required amount of data can be collected in less time.

For every sliding interval, the values with regard to all updates to B data in that interval are collected. In contrast to the General policy, this value contains the cumulated sum of all updates instead of the number of updates. All complete intervals of a window build a histogram of the updates. If a transaction wants to update a record, the histogram of the moving window is used to calculate an empirical probability density function (PDF) f using the standard formula. f is then convoluted CI/δ times to build the PDF for the whole checkpoint interval f_{CI}:

$$f_{CI} = \underbrace{f * f * \cdots * f}_{CI/\delta \text{ times}} \tag{4.11}$$

Convolution is needed, as it is important to preserve the variance.

To reason about the updates in the whole system, the number of servers n has to be known. Convoluting f_{CI} again n times leads to the PDF of the updates in the whole system:

$$f_{CI*n} = \underbrace{f_{CI} * f_{CI} * \cdots * f_{CI}}_{n \text{ times}} \tag{4.12}$$

Given f_{CI*n}, the cumulative distribution function (CDF) F_{CI*n} can be built, which finally can be used to determine the threshold for $P(T - X < 0)$ by looking up the probability that

$$F_{CI*n}(T) > P_C(Y). \tag{4.13}$$

To optimize $P_C(Y)$ we can use the same method as before:

$$F_{CI*n}(T) > \frac{C_A - C_C}{C_O}. \tag{4.14}$$

Note that the threshold T is defined in terms of when the probability is higher than $P_C(Y)$, not smaller.

Although there exist efficient algorithms for convoluting functions, if $n * CI/\delta$ is big enough and/or the item is often updated, the central limit theorem allows us to approximate the CDF with a normal distribution. This permits a faster calculation of the threshold to guarantee the percentile of consistency. The arithmetic mean $\bar{x}$ and the sample standard deviation s can be calculated using statistics of the histogram of the sliding window. That is, one can approximate the CDF F_{CI*n} by using the normal distribution with mean

$$\mu = \bar{x} \cdot CI/\delta \cdot n$$

and standard deviation

$$\sigma = \sqrt{s^2 \cdot CI/\delta \cdot n}.$$

We use statistics gathered at run-time to set the threshold for each individual record depending on the update frequency of the item. As the experiments in Section 4.6 reveal, this Dynamic policy is able to outperform all other policies in terms of overall performance *and* cost for the chosen setup. As the statistics are gathered at run-time, the system is also able to react to changes on the update rate of records.

4.5 Implementation

This section describes the implementation of Consistency Rationing on top of of the system presented in Chapter 3. Consistency Rationing, however, is a more general concept and thus, at the end of this section, we also sketch ways to implement Consistency Rationing on other platforms, such as Yahoo PNUTs.

4.5.1 Configuration of the Architecture

We leverage the architecture and protocols of Chapter 3 deployed on top of Amazon's infrastructure services. The client-server is configured as the combined application and database server running on EC2 and using Amazon's S3 as a persistent store. That is, clients connect to one EC2 server, which in turn implements the set of protocols presented in Chapter 3.5 and 3.6 to coordinate the writes to S3. The protocols already follow a layered design in which every layer increases the level of consistency, and thus, make it easy to use different layers for the different data categories. Consistency Rationing so far only makes use of the protocols for session consistency and 2-Phase-Locking (2PL). For session consistency with atomicity guarantees we use the protocol of Section 3.6.3 together with the protocol of Section 3.6.2 without session recovery in the event of server failures. Thus, the session is lost and monotonicity guarantee for a user might be violated if the server crashes. For 2PL we leverage the protocol of Section 3.6.5. For the sake of uniformity, both protocols use the Advanced Queuing Service (Section 3.3.4 and 3.7).

In addition to the logging presented in the previous chapter we also implement logical logging. In order to implement adaptive consistency guarantees, we extend the architecture with the necessary meta-data management, policies, and the statistical component to gather the required temporal statistics, which are explained in the following sub-sections in more detail.

4.5.2 Logical Logging

In order to allow higher rates of concurrency without inconsistencies, we implement *logical logging*. A logical log message contains the ID of the modified record and the operation Op performed to this record. To retrieve the newest state R_{new} of a record (e.g., when checkpointing the record), the operation Op from the PU queue is applied to the item R_{old}:

$$R_{new} \leftarrow R_{old} + Op.$$

In contrast to logical logging, the protocols presented in Section 3.4.5 use physical logging. Logical updates are robust against concurrent updates as long as the operations are commutative. That is, an update is never lost because it is overwritten and, independent of the order, all updates will become visible. However, also commutative operations may still lead to inconsistent results, for example, by violating consistency constraints like $value > 0$. To avoid problems with different classes of commutative

operations (e.g., multiplication) we restrict our implementation to *add* and *subtract* for numerical values. For non-numerical values (e.g., strings) logical logging behaves as physical logging (i.e., the last update to the string wins). Our Consistency Rationing approach supports both types of values (see Section 4.4).

4.5.3 Meta-data

Every collection in our system contains meta-data about its type. This information is stored on the storage service along with the collection information. Given the collection that a record belongs to, the system checks which consistency guarantees should be enforced. If a record is classified as A data, the system knows that it must apply strong guarantees for this item. On the other hand, if an item is classified as C data, the system operates at lower guarantees. For these two data categories (i.e., A and C data), the type-checking operation is enough to decide on the consistency protocol. For B data, the meta data contains further information: the name of the policy and additional parameters for it.

4.5.4 Statistical Component and Policies

The statistical component is responsible for gathering statistics at run-time. The statistics are collected locally. Based on the local information, the component reasons about the global state (Section 4.4). Using this reasoning, a single server is able to decide locally which protocol to use (i.e., session consistency or 2PL). Only if all servers make the same decision to use A consistency, the stronger consistency level is ensured. Thus, no centralized server is required to make qualified decisions as long as the window size is reasonably big and all requests are evenly distributed across the servers. However, consolidating statistics and broadcasting certain decisions from time to time would make the system more robust. We consider consolidating statistics and broadcasting as part of our future work.

The probabilistic policies differ in the kind of statistics they require. The General policy handles those records with C guarantees for which parallel updates on different servers are unlikely. Therefore, the policy needs information on the update frequency and is particularly suited for rarely updated items. On the other hand, the Dynamic policy is especially interesting for hot spot items. The policy allows a high rate of parallel updates with C guarantees until the likelihood of falling below a limit becomes too high.

For both policies, we store the statistical information directly with the record. For the General policy we reduce the required space for the statistics by using a simple approximation. If we aim for a conflict rate of less than 1%, simulations using Equation (4.3) of Section 4.4.1.1 show that the arrival rate has to be less than ≈ 0.22 independent of the number of servers. Having a sliding factor of δ, the average number of updates inside a cache interval to qualify for session consistency is less than $0.22 * \delta$. We use this property by bounding the value space per slide accordingly and by introducing a special value for update rates beyond the value space.

For example, if we assume a window size of 1 hour with a sliding factor of 5 minutes and a cache interval of 1 minute, 12 window slides need to be stored. The average update rate per slide has to be ≈ 1.1. By allowing some variance, we can use 4 bit per value, reserving one value, here number 16, to stand for updates above 15 and to be further treated as infinite. The window then requires $12 * 4bits = 48bits$ per record, which is an acceptable size.

The operations to gather the statistics are rather simple: per incoming update a simple increment is needed to increase the number in the current slide. Slides are updated in a round robin fashion and an additional number indicates the freshness of the data. Special attention is given to new records: a special flag is set, to avoid misbehavior before sufficient statistics for a new record are gathered.

Collecting statistics for the Dynamic policy works in a similar way. The main difference is that the sum of update values is collected (instead of the number of updates). Therefore, the space to gather the statistics cannot be optimized as before. For efficiency reasons, we only gather the complete statistics for hot spot records. All other records are treated using a default threshold value. The statistics gathered are rather small and may be kept in main memory. For example, in a system with 10,000 hot spot records within the B category, a window size of 100 values and 32 bits to maintain the number of updates per entry and record, only $10,000 * 100 * 32 \approx 4$ MB are needed.

The remaining calculations are described in Section 4.4.3.3. To reduce the estimation error, we convolute the distributions exactly if less than 30 updates were performed inside the window for a specific record. Otherwise, the normal distribution approximation is used to improve the calculation performance as described in Section 4.4.3.3. Another problem which occurs within the Dynamic policy is the start of the system, when no statistics are available. We solve the problem by using the Demarcation policy at the beginning and switching to the Dynamic policy as soon as sufficient statistics have become available.

4.5.5 Implementation Alternatives

The architecture of [BFG+08, 28m09] assumes a storage service which provides eventual consistency guarantees. Higher levels of consistency are achieved by using additional services and by means of protocols. Lately, several systems have appeared that provide higher levels of consistency such as Yahoo PNUTS [CRS+08].

Yahoo PNUTS provides a more advanced API with operations such as: *Read-any*, *Read-latest*, *Test-and-set-write(required version)* etc. Using these primitives, it is possible to implement session consistency directly on top of the cloud service without requiring additional protocols and queues. Serializability cannot be implemented by the API, but PNUTS offers *Test-and-set-write (required version)* which supports implementing optimistic concurrency control for A data. The meta-data, statistical component, and policies described above could be directly adapted. Unfortunately, Yahoo PNUTS is not a public system. Hence, we could not further investigate the implementation nor include it in our experiments. Nevertheless, the authors of PNUTS state that for example, *Read-any* is a cheaper operation than *Read-latest*. Hence, the same trade-offs between cost and consistency exist and Consistency Rationing could also be applied to PNUTS to optimize for cost and performance.

Another recent system is the Microsoft SQL Data Services [Lee08]. This service uses MS SQL Server as an underlying system, and builds replication and load balancing schemes on top. By using MS SQL Server, the system is able to provide strong consistency. However, a strong assumption underlies the system: data is not allowed to span several servers and transactions cannot span over several machines. Hence, the scalability of the system is limited.

Consistency Rationing could be also implemented inside MS SQL Data Services. In particular, the A and C categories and the simple strategies such as the Demarcation policy can help to improve performance. However, as achieving strong consistency is much cheaper in this scenario because no messages between different servers are required, the savings are most likely not as big as in a really distributed setup where data can grow infinitely and gets distributed over several machines.

Finally, Consistency Rationing is also a good candidate for traditional distributed databases such as cluster solutions. Even in a more traditional architecture, stronger levels of consistency require more messages passing between physical machines than weaker levels of consistency. For example, eventual consistency only requires sending reliable messages to persist the data (e.g. on the replicas). Actually, without replication or directly sending the request to all replicas, no additional messages are required at

all, as eventual consistency allows to postpone the consolidation of updates. In contrast, strong consistency, e.g. with a 2-Phase-Locking protocol, requires messages for requesting locks in addition to the messages for persisting the changes. Similarly, traditional snapshot isolation requires an extra round for the validation phase of a transaction. Even with piggy packing of messages and improved partitioning of the data (see for example H-Store [SMA+07, KKN+08]), more messages and coordination work is required, if strong consistency is desired. As message passing between physical machines is typically expensive, Consistency Rationing might also help to reduce the latency and the costs in traditional distributed databases. Again, most of the here-presented components can be applied immediately. However, exploring this direction further is beyond the scope of this thesis.

4.6 Experiments

This section describes the experiments we have performed to study the characteristics and trade-offs of different consistency policies. Our experiments are based on the TPC-W benchmark [Cou02]. The TPC-W benchmark models a web shop and aims to provide a fair comparison in terms of system performance. In all our benchmarks we report on response time and on cost per transaction. These numbers stand as one possible scenario and show the potential of Consistency Rationing.

4.6.1 Experimental Setup

TPC-W benchmark The TPC-W benchmark models a web shop, linking back to our first use case in Section 4.2. The TPC-W benchmark specifies that customers browse through the website of a web shop and eventually buy products as explained in Section 3.8.1.1. We used the same TPC-W implementation as described in Section 3.8.1.1 configured with the Ordering Mix to better reveal the characteristics and trade-offs of our Consistency Rationing approach. Further, the TPC-W benchmark defines 8 different data types (e.g., item data, containing the product information including the product stock). In order to study the characteristics of Consistency Rationing, we assign different consistency categories to the data types (see Table 4.1).

Data Categorization At the beginning of each experiment, the stock of each product is set to a constant value. The TPC-W benchmark defines that the stock of a product

Data	Category	
XACTS	A	(very valuable)
Item	B	(dependent on item's stock)
Customer	C	(few parallel updates)
Address	C	(few parallel updates)
Country	C	(few parallel updates)
Author	C	(few parallel updates)
Orders	C	(append-only, no updates)
OrderLine	C	(append-only, no updates)

Table 4.1: Data categorization

should be refilled periodically. In this case, the benchmark can run forever without a product's stock dropping below a certain threshold. We are interested in inconsistent states of the database in which - due to non-isolated, parallel transactions - the stock of a product drops below zero. To be able to measure these inconsistencies, we do not refill the product's stock but stop the experiment after a given time and count the oversells. All experiments are scheduled to run for 300 seconds and are repeated 10 times.

The Ordering Mix defines that 10% of all actions are purchase actions. One purchase action might buy several different products at once. The total number of products in one purchase is set to a random number between 1 and 6 (inclusively) so that at most six products are bought at once. It is also possible to buy multiple copies of one product. The amount of copies follows the 80-20 rule [GSE+94]. We implemented the 80-20 rule using Gray's self-similar distribution with the parameters $h = 0.2$ and $N = 4$. At most four copies are bought of any single product.

In our experiments, we study the effects of different settings of categorizations. That is, we assign consistency categories to the data types of the TPC-W benchmark: (1) A data, (2) C data, and (3) mixed data.

(1) First, we categorize all data types as A data. That is, all database transactions are isolated and preserve consistency. Categorizing all data types as A data complies with the requirements of the TPC-W benchmark.

(2) Second, we categorize all data types as C data. Database transactions provide atomicity but are are only session consistent. Thud, data might be stale and consistency is not preserved. In particular, oversells of products might occur as product purchases are not exclusive.

(3) Last, we define a mix of data type categorizations. This mix contains A data, C data, and B data. Given the data types of the TPC-W benchmark, we categorize these data types as shown in Table 4.1. Credit card information (XACTS) is defined as A data. Under all circumstances the latest credit card information is used. Items (i.e., products) are categorized as B data as they contain a numerical value that is used as threshold (i.e., the stock). The rest of the data is categorized as C data.

Costs In all our experiments, the database is hosted on S3 and the clients connect to the database via application servers running on EC2. The run-time cost is the cost in ($) of running the EC2 application servers, hosting the data on S3, and connecting to our additional services (i.e., the Advanced Queue Service and the Locking Service). The cost of running EC2 and using S3 is provided by Amazon. The cost of connecting to and using our additional services is the same as in Section 3.8.1 and priced with 0.01$ per 1000 requests. We measure the run-time cost in dollars per 1000 transactions. One transaction relates to exactly one action defined by the TPC-W benchmark.

The penalty cost of inconsistencies is the cost that a company incurs when a promised service cannot be established. Here, it is the cost of overselling products, which cannot be shipped and result in disappointed customers. In larger companies the penalty cost is usually well-known. Because we extremely stress our system (see below), the penalty cost of one oversold product was set to $0.01 per oversold product. The overall cost is the sum of the run-time cost and the penalty cost.

Parameters In our experiments we use 10 application servers hosted on EC2. These application servers carry out transactions following the Ordering Mix defined by the TPC-W benchmark. The system comprises 1000 products (that is less than in the experiments of Section 3.8 to enforce more contention) with an inventory level set to a uniformly distributed value between 10 and 100. The checkpointing interval is set to 30 seconds. Thus, after at most 30 seconds, the up-to-date version of a page is written to S3. The time-to-live of a page was set to 5 seconds. The window size of the Dynamic policy was set to 80 seconds, the sliding factor to 5. That is, it normally takes 80 seconds to adapt to a distribution of updates in our database.

To clearly reveal the individual characteristics of the individual consistency categories and the different adaptive strategies, we stress-test our system. In the 300 seconds of benchmark time, up to 12,000 items are sold corresponding to more than a quarter of the overall available products. This way, we are able to produce stable and repeatable results. Of course, real world scenarios have a lower load by far (while at the same time facing higher penalty costs for oversells). Producing stable results for such workloads

requires running the system for extremely long times, thus making an extensive performance study nearly impossible. For this reason, we only report on the numbers for our stressed system.

4.6.2 Experiment 1: Cost per Transaction

Optimizing the cost of a transaction (in $) is one of the key motivations for Consistency Rationing. This cost includes the cost of running the transactions as well as the penalty cost for overselling. In our first experiment, shown in Figure 4.1, we compare the overall cost per transaction for different consistency guarantees. The cost of A data is about ≈0.15$ per 1000 transactions. The cost of C data significantly varies with the distribution of updates. For the highly skewed 80-20 distribution, many oversells occur because of the high contention of writes to only a few data records. For the adaptive guarantee, we have chosen the Dynamic policy as this policy suits best the shopping scenario. The cost is the least of all three policies. Thus, the Dynamic policy finds the right balance between weak consistency (to save run-time money) and strong consistency (to avoid overselling products).

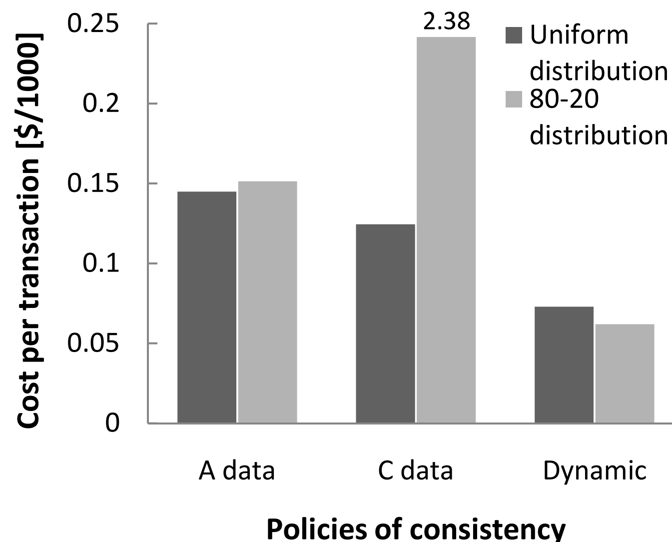

Figure 4.1: Cost (incl. the penalty cost) per trx [$/1000], Vary guarantees

Of course, the overall cost strongly depends on the actual value of the penalty cost. Figure 4.2 shows this impact by varying the penalty cost. For A data, the overall cost is constant because no inconsistencies occur and no penalty cost exists. With an increasing penalty cost, the overall cost for the Dynamic policy converges to the cost

of A data. The Dynamic policy adapts itself to the penalty cost (see Section 4.4.3.3). With increasing penalty cost, it enforces strong consistency for more and more transactions. Eventually, the cost of C data gets amazingly high. At a penalty cost of $0.1, the C data's overall cost is $0.74 ($23) per 1000 transactions for the uniform distribution (80-20 distribution). Therefore, the overall cost of C data is not shown in Figure 4.2.

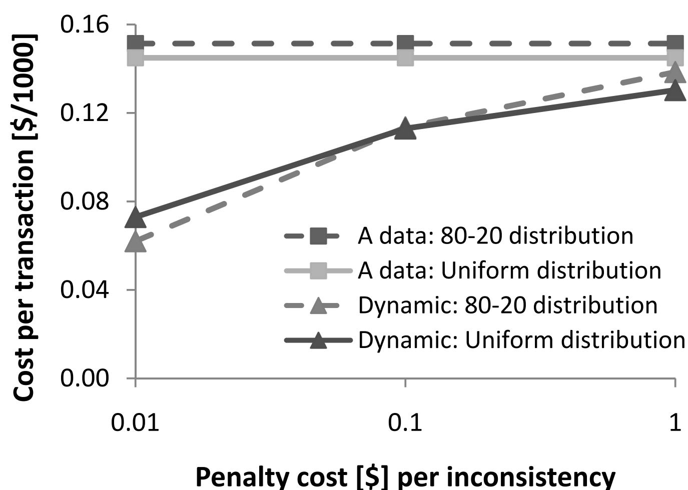

Figure 4.2: Cost per trx [$/1000], Vary penalty cost

Extrapolating the savings of the Dynamic policy to real world applications (such as Amazon or Ebay), the overall savings in IT costs can be rather significant.

4.6.3 Experiment 2: Response Time

To evaluate the performance of our system in more detail, we measure the response times of single actions of the TPC-W benchmark. Figure 4.3 shows the main findings. The numbers do not include the latency to the client, just the transaction time. The response time of A data is the slowest of all policies. Each transaction of A data has to get a lock before any reads or writes can be performed. In contrast, C data shows the fastest response times, as no locks need to be gathered.[2] The Dynamic policy is 24% slower than C data because it sometimes requires locks in order to not oversell products. Thus, the Dynamic policy is able to compete with state of the art policies

[2]The protocols, in particular the Atomicity protocol, is faster than in Section 3.8.3.2 because of the reduced data size. Furthermore, as a result of the higher concurrency and the smaller data-set significant better caching behaviour is achieved. As a comparison see also the effect of caching in the time-to-live experiment of Section 3.8.6.2.

in terms of performance while enabling us to optimize for cost as well (see previous experiment).

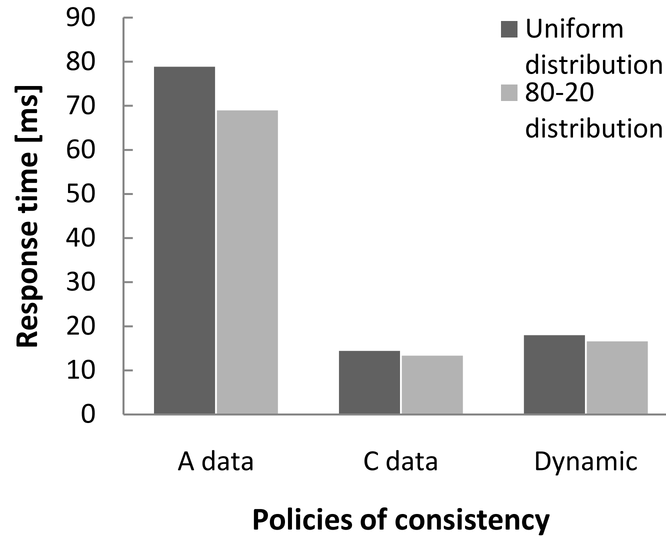

Figure 4.3: Response time [ms]

4.6.4 Experiment 3: Policies

Experiment 3 studies the differences between individual adaptive policies. We focus on the numerical policies as those are the ones that apply for the book-shop scenario, and compare the Fixed threshold policy (with thresholds $T = 40$ and $T = 12$) with the Demarcation policy and the Dynamic policy. Now the Dynamic policy for numerical data is just an advanced form of the General policy, and the Time policy is just a special setting for the Fixed threshold policy, so that the results in this section can serve as indications for the Time and General policy.

Figure 4.4 shows the cost per 1000 transactions in dollars. The fixed threshold of $T = 12$ has been optimized for the uniform distribution of updates to data records (see Experiment 4), and the Fixed threshold policy with $T = 12$ proves to be the cheapest policy for uniformly distributed updates. For skewed update distributions, the same threshold leads to very high costs. Setting the threshold to $T = 40$ lowers the cost for skewed updates but at the same time raises the cost for uniformly distributed updates. That is, the Fixed threshold policy is highly dependent on the threshold and is outperformed by more sophisticates policies. The Demarcation policy reduces the cost incurred for both distributions of updates and the Dynamic policy is even able to outperform the Demarcation policy.

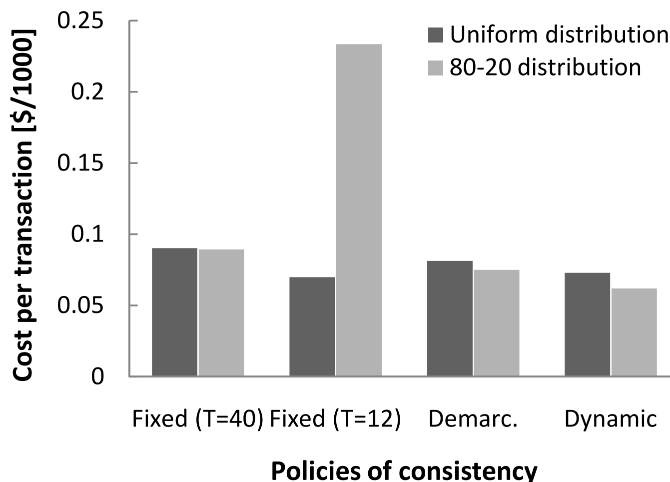

Figure 4.4: Cost per trx [$/1000], Vary policies

In Figure 4.5 the response times of the different policies can be seen. The Dynamic policy has the fastest response times of all policies. If the stock of a product in the database falls below the fixed threshold, the Fixed threshold policy will operate in strong consistency. The higher the threshold is set, the earlier this policy will start to require strong consistency. Therefore, the Fixed threshold policy shows a slower response time for the higher threshold. Even for $T = 12$, the Fixed threshold policy requires unnecessarily many locks compared to the Dynamic policy and the Demarcation policy.

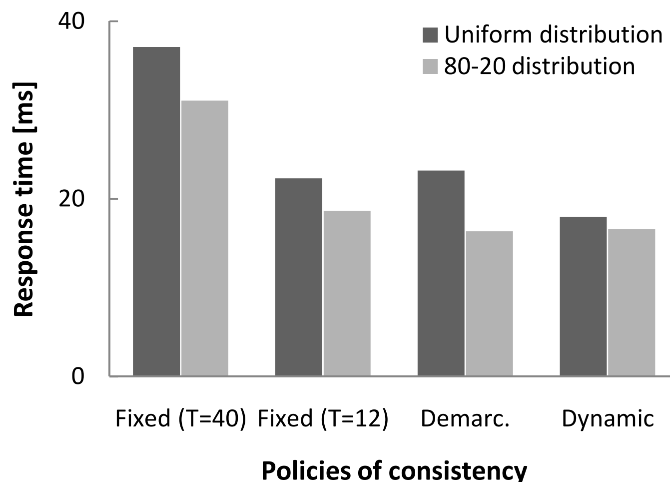

Figure 4.5: Response time [ms], Vary policies

Hence, the Dynamic policy outperforms all other policies that have been described in this chapter in terms of cost *and* response time. This is possible by utilizing statistics gathered at run-time, a possibility ignored by the other, more static policies. Obviously, the results depend on the use case and can be more or less significant depending on the workload and cost factors.

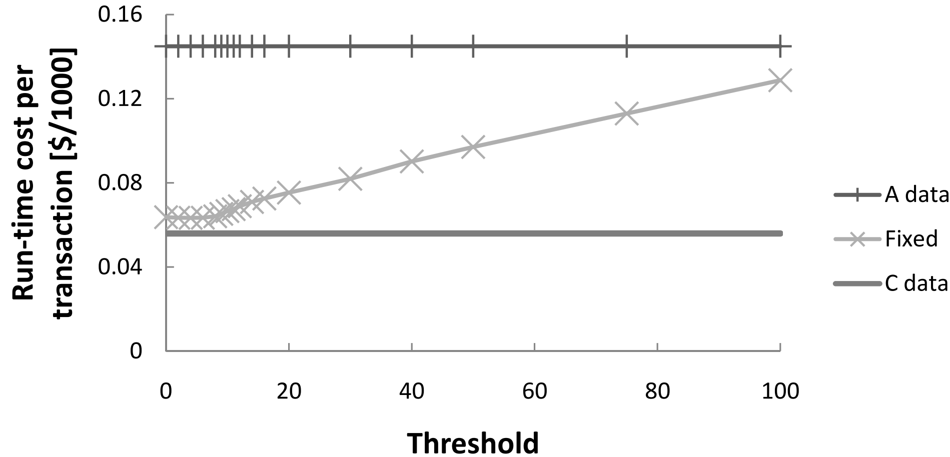

Figure 4.6: Runtime $, Vary threshold

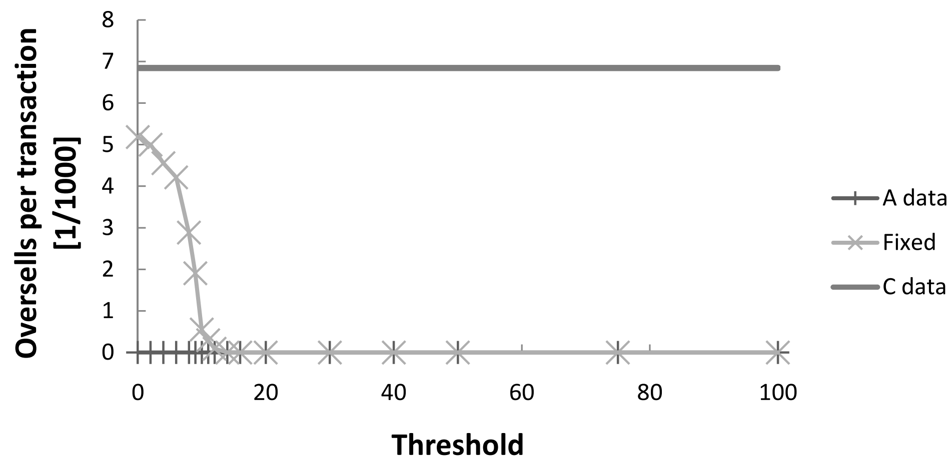

Figure 4.7: Oversells, Vary threshold

4.6.5 Experiment 4: Fixed Threshold

In our last experiment, we experimentally determine the optimal threshold for our benchmark setup assuming that updates are uniformly distributed among all products in the database. With this experiment, we expect to gain further insight into this value-based decision policy as well as the general Consistency Rationing approach.

Figure 4.6 shows the run-time cost of the Fixed threshold policy, A data, and C data in dollars per 1000 transactions. We vary the threshold of the Fixed threshold policy. As A data and C data are not affected by this threshold, the associated costs remain constant. As already seen in Experiment 3, the run-time cost of the Fixed threshold policy increases with an increase in the threshold. A higher threshold enforces an

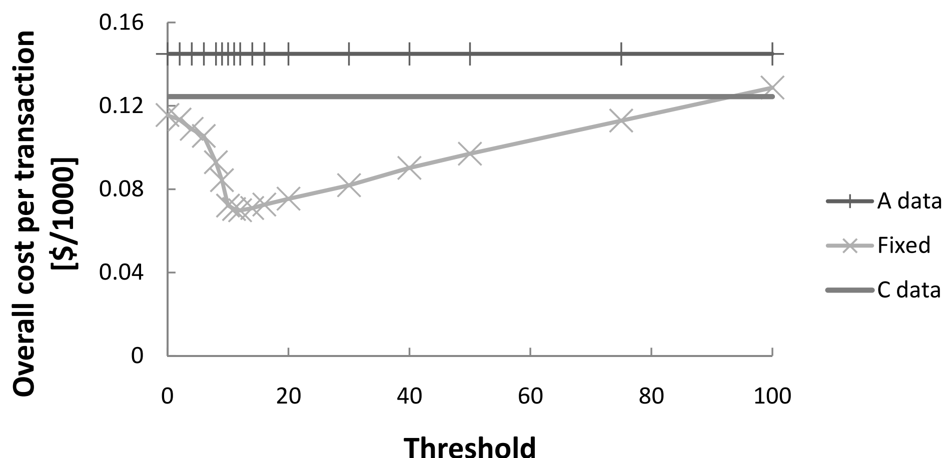

Figure 4.8: Overall $, Vary threshold

earlier switch to strong consistency, which is more expensive (e.g., because of requiring locks and sending more messages).

Figure 4.7 shows the oversells per 1000 transactions. In our stressed benchmark situation, C data shows almost 7 oversells per 1000 transactions. A data, of course, has no oversells at all because the data is always updated with strong consistency guarantees. Due to the earlier switch to strong consistency, the amount of oversells decreases with a higher threshold for the Fixed threshold policy. For a threshold of $T = 14$, no oversells occur.

Given these two figures, Figure 4.8 shows the overall cost per transaction for A data, C data, and the Fixed threshold policy. The overall cost is the sum of the run-time cost and the total penalty costs equalling the number of oversells times the penalty cost per oversell. This overall cost is minimized for $T = 12$. Both, A and C data, have a higher cost per transaction as the optimal setting of the Fixed threshold policy.

These figures demonstrate well how adaptation at run-time lets the consistency vary between the two extreme guarantees of session consistency and serializability.

4.7 Related Work

Many transaction and consistency models have been proposed in the distributed systems and database literature. Common references in the DB literature include [BHG87], [WV02], and [OV99]. In distributed systems, [TS06] is the standard textbook that describes alternative consistency models as well as their trade-offs in terms of consistency

and availability. Our work extends these established models by allowing levels of consistency to be defined on a per-data basis and adapting the consistency guarantees at run-time.

The closest approaches to what we propose in this chapter are [LLJ08], [YV00] and [GDN$^+$03]. [LLJ08] presents an Infrastructure for DEtection-based Adaptive consistency guarantees (IDEA). Upon the detection of inconsistencies, IDEA resolves them if the current consistency level does not satisfy certain requirements. In contrast, our approach tries to avoid inconsistencies from the beginning by using run-time information. [YV00] proposes a set of metrics to cover the consistency spectrum (e.g., numerical error, order error) and a variety of protocols to ensure those spectrums. Those protocols are similar to the Demarcation policy (Section 4.4.3.2). However, their work focuses on ensuring a deviation threshold for a value from the up-to-date view. Our focus is on ensuring a certain consistency constraint (e.g., a stock above 0). In [GDN$^+$03] the authors divide data into categories, for which they provide different replication strategies. The proposed replication strategy for stock data is again similar to the Demarcation policy but more conservative as it will never oversell a product and, under certain circumstances, not sell an item even if it is still on stock.

B data resembles to a certain extent IMS/FastPath and the Escrow transactional model [GK85, O'N86]. Escrow reduces the duration of locks and allows more transactions running in parallel by issuing predicates to a central system at the beginning of the transaction. If the system accepts these predicates, the transaction is safe to assume that these predicates will always hold in the future without keeping locks. In comparison IMS/FastPath does not guarantee that the predicates will hold for the whole transaction. Instead, the predicates are reevaluated at commit time. Both approaches guarantee strong consistency. We extend the ideas of Escrow and Fast Path by means of probabilistic guarantees and adaptive behavior. Furthermore, IMS/FastPath and Escrow require global synchronization to bring all predicates in order, whereas our approach avoids synchronization as long as possible.

Along the same lines, a great deal of work has been done on distributed consistency constraints [CGMW94, GT93] and limited divergence of replicas [OLW01, SRS04]. Distributed consistency constraints either ensure strong consistency or weaken the consistency in small intervals, which in turn can lead to inconsistencies. The work on limited divergence relaxes the consistency criteria of replicas to increase performance, but at the same time limits the divergence between the replicas (e.g, in value, or staleness). We extend these works by means of probabilistic consistency guarantees and optimization goals in both, performance *and* cost.

In [GLR05, LGZ04, GLRG04], the authors propose a technique to implement consistency constraints for local, cached copies in SQL processing. That is, users read from local outdated data snapshots if it is within the bounds of the consistency constraints. All writes are redirected to the backend, requiring traditional transaction models. Our work does not require a centralized backend and extends the ideas by the notion of probabilistic guarantees.

The Mariposa system is a distributed data management system that supports high data mobility [S+96]. In Mariposa, clients may hold cached copies of data items. All writes to a data item must be performed on the primary copy. A cost model in Mariposa determines where the primary copy is placed. This work is orthogonal to Mariposa as we do not address data locality and concentrate on switching consistency at run-time.

The authors of [TM96] propose adaptive concurrency control based on a stochastic model. However, their model does not consider inconsistency. Instead the implementation switches between optimistic and pessimistic control for the same level of guarantee.

H-Store [SMA+07, KKN+08] tries to completely avoid any kind of synchronization by analyzing transactions and partitioning the data. H-Store provides strong consistency guarantees but requires to know all the queries and transactions upfront. We do not make such an assumption. Furthermore, it might be possible to combine the two approaches with the corresponding improvement in performance.

4.8 Conclusion

In cloud computing storage services, every service request has an associated cost. In particular, it is possible to assign a very precise monetary cost to consistency protocols (i.e., the number of service calls needed to ensure the consistency level times the cost per call). Therefore, in cloud storage services, consistency not only influences the performance and availability of the systems but also the overall operational cost. In this chapter, we proposed a new concept called Consistency Rationing to optimize the run-time cost of a database system in the cloud when inconsistencies incur a penalty cost. The optimization is based on allowing the database to exhibit inconsistencies if it helps to reduce the cost of a transaction but does not cause higher penalty costs.

In our approach, we divide (*ration*) the data into three consistency categories: A, B, and C. The A category demands strong consistency guarantees and shows high cost per transaction. The C category demands session consistency, shows low cost, but will result in inconsistencies. Data in the B category is handled with either strong or session

consistency depending on the specified policy. In this chapter, we present and compare several of such policies to switch consistency guarantees including policies providing probabilistic guarantees. As shown in our experiments, Consistency Rationing can significantly lower the overall cost and improve the performance of a cloud-based database systems. Our experiments show further that adapting the consistency by means of temporal statistics has the biggest potential to lower the overall cost while maintaining acceptable performance.

Future work includes in particular extensions to the notion of probabilistic consistency guarantees. Possible improvements include: better and faster statistical methods, automatic optimizations with regards to other parameters (e.g., energy consumption), adding budget restrictions to the cost function, and relaxing other principles of the ACID paradigm (e.g., durability). Further, the notion of probabilistic consistency guarantee might also be applicable to deal with the CAP theorem, if availability instead of cost is the main optimization factor. However, going into more detail is beyond the scope of this thesis.

Chapter 5

Windowing for XQuery

5.1 Introduction

One of the main challenges of moving applications into the cloud is the need to master multiple languages [Hay08] . Many applications rely on a backend running SQL or other (simplified) languages. On the client side, JavaScript embedded within HTML is used and the application server, standing between the backend and the client, implements the logic using some kind of scripting language (such as PHP, Java or Python). XML plays a fundamental role in this architecture. XML is used for communication between the different layers. For instance, web and REST services exchange messages via XML. Atom feeds or RSS are further examples. Meta-data is another area where XML is used quite frequently. Examples for meta-data represented in XML are configuration files, schemas (e.g., XML schemas, the Vista file system), design specifications (e.g., Eclipse's XMI), interface descriptions (e.g., WSDL), or logs (e.g., Web logs). Furthermore, in the document world, big vendors such as Sun (OpenOffice) and Microsoft (MS Office) have also moved towards representing their data in XML. XHTML and RSS blogs are further examples that show the success of XML in this domain.

To overcome the variety of languages and the need to process XML in various places, there has been an increased demand to find the right paradigms to process data with a uniform language (see also Section 2.4. XQuery, XLinq or Ruby are examples for that development. Arguably, XQuery is one of the most promising candidates for this purpose [BCF$^+$07]. XQuery has been particularly designed for XML and is already successfully used as a programming language in several middle-tier application servers such as XL [FGK03], MarkLogic [Mar09] or Oracle WebLogic [Ora09, BCLW06]. Furthermore, XQuery is able to run on all layers of a typical web-application (i.e, database,

application and client layer) as demonstrated in [FPF+09, CCF+06, 28m09]. Finally, XQuery is also the choice of programming language for the Sausalito product of 28msec Inc. [28m09], which incorporates the architecture of Chapter 3. Sausalito is an XQuery application platform deployed on top of the infrastructure services from Amazon and combines the application server together with the database stack similar to the EC2 client-server configuration presented in Chapter 3.

Even though XQuery 1.0 is extremely powerful (it is Turing-complete), it lacks important functionality. In particular, XQuery 1.0 lacks support for window queries and continuous queries. This omission is especially disappointing because exactly this support is needed to process the main targets of XML: Communication data, meta-data, and documents. Communication data is often represented as a stream of XML items. For many applications, it is important to detect certain patterns in such a stream: A credit card company, for instance, might be interested in detecting if a credit card is used particularly often during one week as compared to other weeks. Implementing this audit involves a continuous window query. The analysis of a web log, as another example, involves the identification and processing of user sessions, which again requires a window query. Document formatting needs operations such as pagination to enable users to browse through the documents page by page; again, pagination is best implemented using a window query.

Both window queries and continuous queries have been studied extensively in the SQL world; proposals for SQL extensions are for instance described in [LS03, ABW06, CCD+03, Str07]. For several reasons this work is not applicable in the XML world: Obviously, SQL is not appropriate to process XML data. It can neither directly read XML data, nor can SQL generate XML data if the output is required to be XML (e.g., an RSS feed or a new paginated document). Also, the SQL extensions are not sufficiently expressive to address all use cases mentioned above, even if all data were relational (e.g., CSV). The SQL extensions have been specifically tuned for particular streaming applications and support only simple window definitions based on time or window size.

In this chapter we will extend XQuery in order to support window queries and continuous queries. The objective is to have a powerful extension that is appropriate for all use cases, including the *classic* streaming applications for which the SQL extensions were designed and the more *progressive* use cases of the XML world (i.e., RSS feeds and document management). At the same time, the performance needs to compare to the performance of continuous SQL queries: There should not be a performance penalty for using XQuery.

Obviously, our work is based on several ideas and the experience gained in the SQL

world from processing data streams. Nevertheless, there are important differences to the SQL data stream approach. In fact, it is easier to extend XQuery because the XQuery data model (XDM), which is based on *sequences of items* [FMM+06], is already a good match to represent data streams and windows. As a result, the proposed extensions compose nicely with all existing XQuery constructs and all existing XQuery optimization techniques remain relevant. In contrast, extending SQL for window queries involves a more drastic extension of the relational data model and a great deal of effort in the SQL world has been spent on defining the right mappings for these extensions. As an example, CQL [ABW06] defines specific operators that map streams to relations.

In summary, this chapter makes the following contributions:

- *Window Queries:* The syntax and semantics of a new `FORSEQ` clause in order to define and process complex windows using XQuery. Presently, the proposed extension has been accepted for the upcoming XQuery 1.1 standard.
- *Continuous Queries:* A simple extension to the XQuery data model (XDM) in order to process infinite data streams and use XQuery for continuous queries.
- *Use Cases:* A series of examples that demonstrate the expressiveness of the proposed extensions.
- *Implementation Design:* Show that the extensions can be implemented and integrated with little effort into existing XQuery engines and that simple optimization techniques are applicable in order to get good performance.
- *Linear Road Benchmark:* The results of running the Linear Road benchmark [ACG+04] on top of an open source XQuery engine which has been enhanced with the proposed extensions. The benchmark results confirm that the proposed XQuery extensions can be implemented efficiently.

The remainder of this chapter is organized as follows: Section 5.2 gives a motivating example. Section 5.3 presents the proposed syntax and semantics of the new `FORSEQ` clause used to define windows in XQuery. Section 5.4 proposes an extension to the XQuery data model in order to define continuous XQuery expressions. Section 5.5 lists several examples that demonstrate the expressive power of the proposed extensions. Section 5.6 explains how to extend an existing XQuery engine and optimize XQuery window expressions. Section 5.7 presents the results of running the Linear Road benchmark on an extended XQuery engine. Section 5.8 gives an overview of related work. Section 5.9 contains conclusions and suggests avenues for future work.

5.2 Usage Scenarios

5.2.1 Motivating Example

The following simple example illustrates the need for an XQuery extension. It involves a blog with RSS items of the following form:

```
<rss:item>
... <rss:author>...</rss:author> ...
</rss:item>
```

Given such a blog, we want to find all *annoying authors* who have posted three consecutive items in the RSS feed. Using XQuery 1.0, this query can be formulated as shown in Figure 5.1. The query involves a three-way self-join which is not only tedious to specify but also difficult to optimize. In contrast, Figure 5.2 shows this query using the proposed `FORSEQ` clause. This clause partitions the blog into sequences of postings of the same author (i.e., windows) and iterates over these windows (Lines 1-4 of Figure 5.2). If a window contains three or more postings, then the author of this window of postings is annoying and the query return this author (Lines 5 and 6). The syntax and semantics of the `FORSEQ` clause are defined in detail in Section 5.3 and need not be understood at this point. For the moment, it is only important to observe that this query is straightforward to implement and can be executed in linear time or better, if the right indexes are available. Furthermore, the definition of this query can easily be modified if the definition of *annoying author* is changed from, say, three to five consecutive postings. In comparison, additional self-joins must be implemented in XQuery 1.0 in order to implement this change. [1]

[1] In fact, the two queries of Figures 5.1 and 5.2 are not equivalent. If an author posts four consecutive postings, this author is returned twice in the expression of Figure 5.1, whereas it is returned only once in Figure 5.2.

```
for $first at $i in $rssfeed
let $second := $rssfeed[$i+1],
let $third := $rssfeed[$i+2]
where ($first/author eq $second/author) and
   ($first/author eq $third/author)
return $first/author
```

Figure 5.1: Annoying Authors: XQuery 1.0

```
forseq $w in $rssfeed tumbling window
   start curItem $first when fn:true()
   end  nextItem $lookAhead when
      $first/author ne $lookAhead/author
where count($w) ge 3
return $w[1]/author
```

Figure 5.2: Annoying Authors: Extended XQuery

5.2.2 Other Applications

The management of RSS feeds is one application that drove the design of the proposed XQuery extensions. There are several other areas; the following is a non-exhaustive list of further application scenarios:

- *Web Log Auditing:* In this scenario, a window contains all the actions of a user in a session (from login to logout). The analysis of a web log involves, for example, the computation of the average number of clicks until a certain popular function is found. Security audits and market-basket analyses can also be carried out on user sessions.
- *Financial Data Streams:* Window queries can be used in order to carry out fraud detection, algorithmic trading and finding opportunities for arbitrage deals by computing call-put parities [FKKT06].
- *Social Games / Gates:* An RFID reader at a gate keeps track of the people that enter and exit a building. People are informed if their friends are already in the building when they themselves access it.
- *Sensor Networks:* Window queries are used in order to carry out data cleaning. For instance, rather than reporting each individual measurement the average of the last five measurements (possibly, disregarding the minimum and maximum) is accounted for [JAF+06].
- *Document Management:* Different text elements (e.g., paragraphs, tables, figures) are grouped into pages. In the index, page sequences such as 1, 2, 3, 4, 7 are reformatted into 1-4, 7 [Kay06].

We compiled around sixty different use cases in these areas in a separate document [FKKT06]. All these examples have in common that they cannot be implemented well using the current Version 1.0 of XQuery without support for windows. Furthermore,

many examples of [FKKT06] cannot be processed using SQL, even considering the latest extensions proposed in [LS03, ABW06, CCD+03, Str07] because these examples require powerful constructs in order to define window boundaries. Most of these use cases involve other operators such as negation, existential and universal quantification, aggregation, correlation, joins, and transformation in addition to window functions. XQuery already supports all these operators which makes extending XQuery a natural candidate avoiding inventing a new language from scratch that addresses these applications.

5.3 FORSEQ Clause

5.3.1 Basic Idea

Figure 5.2 gives an example of the `FORSEQ` clause. The `FORSEQ` clause is an extension of the famous FLWOR expressions of XQuery. It is freely composable with other `FOR`, `LET`, and `FORSEQ` clauses. Furthermore, FLWOR expressions that involve a `FORSEQ` clause can have an optional `WHERE` and/or `ORDER BY` clause and must have a `RETURN` clause, just as any other FLWOR expression. A complete grammar of the extended FLWOR expression is given in Figure 5.3.

Like the `FOR` clause, the `FORSEQ` clause iterates over an input sequence and binds a variable with every iteration. The difference is that the `FORSEQ` clause binds the vari-

$$
\begin{aligned}
FLWORExpr &::= (\textbf{ForseqClause}|ForClause|LetClause)+\ WhereClause?\\
&\quad OrderByClause?\ \text{"return"}\ ExprSingle\\
ForseqClause &::= \text{"forseq"}\ \text{"\$"}VarName\ TypeDeclaration?\ \text{"in"}\ ExprSingle\ WindowType?\\
&\quad (\text{","}\ \text{"\$"}VarName\ TypeDeclaration?\ \text{"in"}\ ExprSingle\ WindowType?)*\\
WindowType &::= (\text{"tumbling window"}|\text{"sliding window"}|\text{"landmark window"})\ StartExpr\ EndExpr\\
StartExpr &::= \text{"start"}\ WindowVars?\ \text{"when"}\ ExprSingle\\
EndExpr &::= \text{"force"}?\ \text{"end"}\ WindowVars?\ \text{"when"}\ (\text{"newstart"}|ExprSingle)\\
WindowVars &::= (\text{"position"}|\text{"curItem"}|\text{"prevItem"}|\text{"nextItem"})\ \text{"\$"}VarName\ TypeDeclaration?\\
&\quad (\text{","}\ (\text{"position"}|\text{"curItem"}|\text{"prevItem"}|\text{"nextItem"})\ \text{"\$"}VarName\ TypeDeclaration?)*
\end{aligned}
$$

Figure 5.3: Grammar of Extended FLWOR Expression

able to a *sub-sequence* (aka window) of the input sequence in each iteration, whereas the FOR clause binds the variable to an *item* of the input sequence. To which sub-sequences the variable is bound is determined by additional clauses. The additional TUMBLING WINDOW, START, and END clauses of Figure 5.2, for instance, specify that $w is bound to each consecutive sub-sequence of postings by the same author. In that example, the window boundaries are defined by the change of author in the WHEN clause of the END clause (details of the semantics are given in the next subsection).

The running variable of the FORSEQ clause ($w in the example) can be used in any expression of the WHERE, ORDER BY, RETURN clauses or in expressions of nested FOR, LET, and FORSEQ clauses. It is only required that those expressions must operate on sequences (rather than individual items or atomic values) as input. In Figure 5.2, for example, the count function is applied to $w in the WHERE clause in order to determine whether $w is bound to a series of postings of an *annoying author* (three or more postings).

As shown in Figure 5.3, FLWOR expressions with a FORSEQ clause can involve an ORDER BY clause, just like any other FLWOR expression. Such an ORDER BY clause specifies in which order the sub-sequences (aka windows) are bound to the running variable. By default, and in the absence of an ORDER BY clause, the windows are bound in ascending order of the position of the *last* item of a window. If two (overlapping) windows end in the same item, then their order is implementation-defined. For instance, *annoying authors* in the example of Figure 5.2 are returned in the order in which they have made annoying postings. This policy naturally extends the order in which the FOR clause orders the bindings of its input variable in the absence of an ORDER BY clause.

The FORSEQ clause does not involve an extension or modification of the XQuery data model (XDM) [FMM+06]. Binding variables to sequences is naturally supported by XDM. As a result, the FORSEQ clause is fully composable with all other XQuery expres-

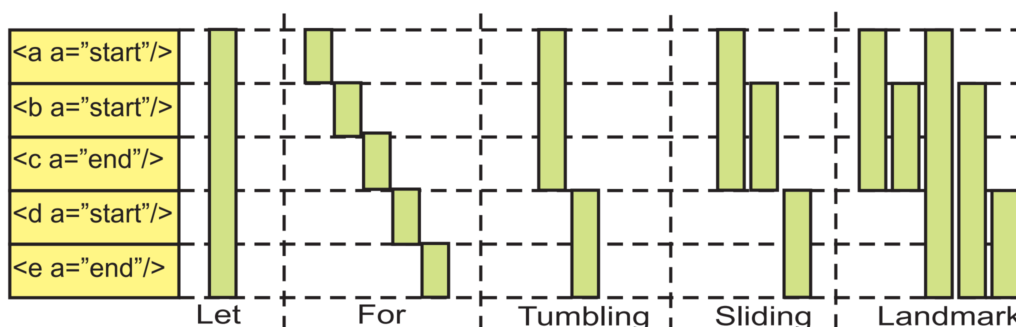

Figure 5.4: Window Types

sions and no other language adjustments need to be made. There is no catch here. In contrast, extending SQL with windows involves an extension to the relational data model. As mentioned in the introduction, a great deal of effort has been invested into defining the exact semantics of such window operations in such an extended relational data model.

Furthermore, the XQuery type system does not need to be extended, and static typing for the `FORSEQ` clause is straightforward. To give a simple example, if the static type of the input is *string**, then the static type of the running variable is *string+*. The "+" quantifier is used because the running variable is never bound to the empty sequence. To give a more complex example, if the static type of the input sequence is *string*, integer** (i.e., a sequence of strings followed by a sequence of integers), then the static type of the running variable is: *(string+,integer** | *string*,integer+)*; i.e., a sequence of strings, a sequence of integers, or a sequence of strings followed by integers. (Similarly, simple rules apply to the other kinds of variables that can be bound by the `FORSEQ` clause.)

5.3.2 Types of Windows

Previous work on extending SQL to support windows has identified different kinds of windows; i.e., tumbling windows, sliding windows, and landmark windows [GO03]. Figure 5.4 shows examples of these three types of windows; for reference, Figure 5.4 also shows how the traditional `FOR` and `LET` clauses of XQuery work. The three types of windows differ in the way in which the windows overlap: tumbling windows do not overlap; sliding windows overlap, but have disjoint first items; and landmark windows can overlap in any way. Following the experiences made with SQL, we propose to support these three kinds of windows in XQuery, too. This subsection describes how the `FORSEQ` clause can be used to support these kinds of windows.

Furthermore, previous work on windows for SQL proposed alternative ways to define the window boundaries (start and end of a window). Here, all published SQL extensions [LS03, ABW06, CCD$^+$03, Str07] propose to define windows based on size (i.e., number of items) or duration (time span between the arrival of the first and last item). Our proposal for XQuery is more general and is based on using predicates in order to define window boundaries. Size and time constraints can easily be expressed in such a predicate-based approach (examples are given in subsequent parts of this chapter). Furthermore, more complex conditions involving any property of an item (e.g., the author of a posting in a blog) can be expressed in our proposal. As one consequence of

having predicate-based window boundaries, the union of all windows does not necessarily cover the whole input sequence; that is, it is possible that an input item is not part of any window.

5.3.2.1 Tumbling Windows

The first kind of window supported by the FORSEQ clause is a so-called *tumbling window* [PS06]. Tumbling windows partition the input sequence into disjoint sub-sequences, as shown in Figure 5.4. An example of a query that involves a tumbling window is given in Figure 5.2 in which each window is a consecutive sequence of blog postings of a particular author. Tumbling windows are indicated by the TUMBLING WINDOW keyword as part of the WindowType declaration in the FORSEQ clause (Figure 5.3).

The boundaries of a tumbling window are defined by (mandatory) START and END clauses. These clauses involve a WHEN clause which specifies a predicate. Intuitively, the WHEN condition of a START clause specifies when a window should start. For each item in the sequence, this clause is checked for a match. Technically, a match exists if the effective Boolean value (EBV) [DFF^{+}07] of the WHEN condition evaluates to true. As long as no item matches, no window is started and the input items are ignored. Thus, it is possible that certain items are not part of any window. Once an item matches the WHEN condition of the START clause, a new window is *opened* and the matching item is the first item of that window. At this point, the WHEN condition of the END clause is evaluated for each item, including the first item. Again, technically speaking, the EBV is computed. If an item matches the END condition, it becomes the last item of the window.

Any XQuery expression can be used in a WHEN clause (Figure 5.3), including expressions that involve existential quantification (on multiple sub-elements) or nested FLWOR expressions (possibly with FORSEQ). The semantics of the START and END clauses for tumbling windows can best be shown using the automaton depicted in Figure 5.5. The condition of the START clause is not checked for an open window. A window is closed either when its END condition is fulfilled or when the input sequence ends.

To give two simple examples, the FOR clause of XQuery can be simulated with a FORSEQ clause as follows:

```
forseq $w in $seq tumbling window
   start when fn:true()
   end when fn:true()  ...
```

That is, each item opens and immediately closes a new window (both START and END

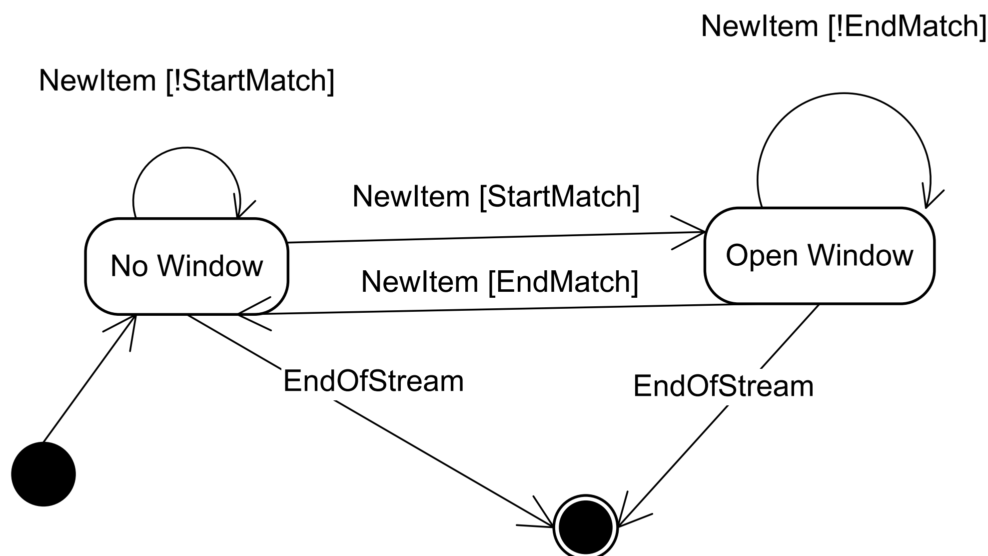

Figure 5.5: Window Automaton

conditions are set to true) so that each item represents a separate window. The LET clause can be simulated with a FORSEQ clause as follows:

```
forseq $w in $seq tumbling window
   start when fn:true()
   end when fn:false() ...
```

The first item of the input sequence opens a new window (START condition is true) and this window is closed at the end of the input sequence. In other words, the whole input sequence is bound to the running variable as a single window.

In order to specify more complex predicates, both the START and the END clause allow the binding of new variables. The first kind of variable identifies the position of a potential first (in the START clause) or last item (in the END clause), respectively. For instance, the following FORSEQ clause partitions the input sequence ($seq) into windows of size three with a potentially smaller last window:

```
forseq $x in $seq tumbling window
   start position $i when fn:true()
   end position $j when $j-$i eq 2 ...
```

For each window, `$i` is bound to the position of the first item of the window in the input sequence; i.e., `$i` is 1 for the first window, 4 for the second window, and so on. Correspondingly, `$j` is bound to the position of the last item of a window as soon as that item has been identified; i.e., `$j` is 3 for the first window, 6 for the second window, and so on. In this example, `$j` might be bound to an integer that is not a multiple of three for the last window at the end of the input sequence.

Both `$i` and `$j` can be used in the `WHEN` expression of the `END` clause. Naturally, only variables bound by the `START` clause can be used in the `WHEN` condition of the `START` clause. Furthermore, in-scope variables (e.g., `$seq` in the examples above) can be used in the conditions of the `START` and `END` clauses. The scope of the variables bound by the `START` and `END` clauses is the whole remainder of the FLWOR expression. For instance, `$i` and `$j` could be used in the `WHERE, RETURN` and `ORDER BY` clauses or in any nested `FOR`, `LET`, or `FORSEQ` clauses in the previous example.

In addition to positional variables, variables that refer to the previous (*prevItem*), current (*currItem*), and next item (*nextItem*) of the input sequence can be bound in the `START` and `END` clause. In the expression of Figure 5.2, for instance, the `END` clause binds variable `$lookAhead` to the item that comes after the last item of the current window (i.e., the first item of the next window). These extensions are syntactic sugar because these three kinds of variables can be simulated using positional variables; e.g., `end nextItem $lookAhead when $lookAhead ...` is equivalent to `end position $j when $seq[$j+1] ...` In both cases, an out-of-scope binding (at the end of the input sequence) is bound to the empty sequence.

5.3.2.2 Sliding and Landmark Windows

In the SQL world, so-called sliding and landmark windows have additionally been identified as being useful and applicable to many real-world situations. In contrast to tumbling windows, sliding and landmark windows can overlap. The difference between sliding and landmark windows is that two sliding windows never have the first item in common, whereas landmark windows do not have such a constraint (Figure 5.4). A more formal definition of sliding and landmark windows is given in [PS06].

The `FORSEQ` clause also supports sliding and landmark windows. As shown in Figure 5.3, only the `TUMBLING WINDOW` keyword needs to be replaced in the syntax. Again, (mandatory) `START` and `END` clauses specify the window boundaries. The semantics are analogous to the semantics of the `START` and `END` clauses of a tumbling window (Figure 5.5). The important difference is that each item potentially opens one (for slid-

ing windows) or several new windows (for landmark windows) so that conceptually, several automata need to be maintained at the same time. As the modifications to Section 5.3.2.1 are straightforward, we do not provide further details.

5.3.3 General Sub-Sequences

In its most general form, the FORSEQ clause takes no additional clauses; i.e., no specification of the window type and no START and END clauses. In this case, the syntax is as follows (Figure 5.3):

```
forseq $w in $seq ...
```

This general version of the FORSEQ clause iterates over all possible sub-sequences of the input sequence. These sub-sequences are not necessarily consecutive. For example, if the input sequence contains the items (a, b, c), then the general FORSEQ carries out seven iterations ($2^n - 1$, with n the size of the input sequence), thereby binding the running variable to the following sub-sequences: (a), (a,b), (b), (a,b,c), (a,c), (b,c), and (c). Again, the sequences are ordered by the position of their last item (Section 5.3.1); i.e., the sequence (a) preceeds the sequences that end with a "b" which in turn come before the sequences that end with a "c". Again, the running variable is never bound to the empty sequence.

This general FORSEQ clause is the most powerful variant. Landmark, sliding, and tumbling windows can be seen as special cases of this general FORSEQ. We propose to use special syntax for these three kinds of windows because use cases that need the three types of windows are frequent in practice [FKKT06]. Furthermore, the general FORSEQ clause is difficult to optimize. Use cases for which landmark, sliding, and tumbling windows are not sufficient, are given in [WDR06] for RFID data management. In those use cases, regular expressions are needed in order to find patterns in the input stream. Such queries can be implemented using the general FORSEQ clause by specifying the relevant patterns (i.e., regular expressions) in the WHERE clause of the general FORSEQ expression.

5.3.4 Syntactic Sugar

There are use cases which benefit from additional syntactic sugar. The following paragraphs present such syntactic sugar.

5.3.4.1 End of Sequence

As mentioned in Section 5.3.2.1, by default the condition of the `END` clause is always met at the end of a sequence. That is, the last window will be considered even if its last item does not match the `END` condition. In order to specify that the last window should only be considered if its last item indeed matches the `END` condition, the `END` clause can be annotated with the special keyword `FORCE` (Figure 5.3). The `FORCE` keyword is syntactic sugar because the last window could also be filtered out by repeating the `END` condition in the `WHERE` clause.

5.3.4.2 NEWSTART

There are several use cases in which the `START` condition should implicitly define the end of a window. For example, the day of a person starts every morning when the person's alarm clock rings. Implicitly, this event ends the previous day, even though it is not possible to concretely identify a condition that ends the day. In order to implement such use cases, the `WHEN` condition of the `END` clause can be defined as `NEWSTART`. As a result, the `START` condition (rather than the `END` condition) is checked for each open window in order to determine when a window should be closed. Again, the `NEWSTART` option is syntactic sugar and avoids that the condition of the `START` clause is replicated in the `END` clause.

5.3.5 Summary

Figure 5.3 gives the complete grammar for the proposed extension of XQuery's FLWOR expression with an additional `FORSEQ` clause. Since there are many corner cases, the grammar looks complicated at a first glance. However, the basic idea of the `FORSEQ` clause is simple. `FORSEQ` iterates over an input sequence, thereby binding a subsequence of the input sequence to its running variable in each iteration. Additional clauses specify the kind of window. Furthermore, predicates in the `START` and `END`

clauses specify the window boundaries. This mechanism is powerful and sufficient for a broad spectrum of use cases [FKKT06]. We are not aware of any use case that has been addressed in the literature on window queries that cannot be implemented in this way.

Obviously, there are many ways to extend XQuery in order to support window queries. In addition to its expressive power and generality, the proposed `FORSEQ` clause has two more advantages. First, it composes well with other `FOR, LET`, and `FORSEQ` clauses as part of a FLWOR expression. Any kind of XQuery expression can be used in the conditions of the `START` and `END` clauses, including nested `FORSEQ` clauses. Second, the `FORSEQ` clause requires no extension of the XQuery data model (XDM). As a result, the existing semantics of XQuery functions need not be modified. Furthermore, this feature enables full composability and optimizability of expressions with `FORSEQ`.

5.4 Continuous XQuery

The second extension proposed in this chapter makes XQuery a candidate language to specify continuous queries on potentially infinite data streams. In fact, this extension is orthogonal to the first extension, the `FORSEQ` clause: Both extensions are useful independently, although we believe that they will often be used together in practice.

The proposal is to extend the XQuery data model (XDM) [FMM+06] to support infinite sequences as legal instances of XDM. As a result, XQuery expressions can take an infinite sequence as input. Likewise, XQuery expressions can produce infinite sequences as output. A simple example illustrates this extension. Every minute, the temperature sensor in an ice-cream warehouse produces measurements of the following form: `<temp>-8</temp>`. Whenever a temperature of ten degrees or higher is measured, an alarm should be raised. If the stream of temperature measurements is bound to variable `$s`, this alarm can be implemented using the following (continuous) XQuery expression:

```
declare variable $s as (temp)** external;
for $m in $s where $m ge 10
return <alarm> { $m } </alarm>
```

In this example, variable `$s` is declared to be an external variable that contains a potentially infinite sequence of temperature measurements (indicated by the two asterisks). Since `$s` is bound to a (potentially) infinite sequence, this expression is illegal in XQuery

1.0 because the input is not a legal instance of the XQuery 1.0 data model. Intuitively, however, it should be clear what this continuous query does: whenever a temperature above 10 is encountered, an alarm is raised. The input sequence of the query is infinite and so is the output sequence.

Extending the data model of a query language is a critical step because it involves refining the semantics of all constructs of the query language for the new kind of input data. Fortunately, this particular extension of XDM for infinite sequences is straightforward to implement in XQuery. The idea is to extend the semantics of non-blocking functions (e.g., for, forseq, let, distinct-values, all path expressions) for infinite input sequences and to specify that these non-blocking functions (potentially) produce infinite output. Other non-blocking functions such as retrieving the ith element (for some integer i) are also defined on infinite input sequences, but generate finite output sequences. Blocking functions (e.g., order by, last, count, some) are not defined on infinite sequences; if they are invoked on (potentially) infinite sequences, then an error is raised. Such violations can always be detected statically (i.e., at compile-time). For instance, the following XQuery expression would not compile because the `fn:max()` function is a blocking function that cannot be applied to an infinite sequence:

```
declare variable $s as (temp)** external;
fn:max($s)
```

Extending the XDM does not involve an extension of the XQuery type system. *(temp)*** is the same type as *(temp)**. The two asterisks are just an annotation to indicate that the input is potentially infinite. These annotations (and corresponding annotations of functions in the XQuery function library) are used in the data flow analysis of the compiler in order to statically detect the application of a blocking function on an infinite sequence (Section 5.6).

A frequent example in which the `FORSEQ` clause and this extension for continuous queries are combined, is the computation of moving averages. Moving averages are for instance useful in sensor networks as described in Section 5.2.2: Rather than reporting the current measurement, an average of the current and the last four measurements is reported for every new measurement. Moving averages can be expressed as follows:

```
declare variable $seq as (xs:int)** external;
forseq $w in $seq sliding window
   start position $s when fn:true()
   end position $e when $e - $s eq 4
return fn:avg($w)
```

5.5 Examples

This section contains four examples which demonstrate the expressive power of the FORSEQ clause and continuous XQuery processing. These examples are inspired by applications on web log auditing, financial data management, building / gate control, and sensor networks (Section 5.2.2). A more comprehensive set of examples from these application areas and from real customer use cases can be found in [FKKT06].

5.5.1 Web Log Analysis

The first example involves the analysis of a log of a (web-based) application. The log is a sequence of entries. Among others, each entry contains a timestamp (tstamp element) and the operation (op) carried out by the user. In order to determine the user activity (number of *login* operations per hour), the following query can be used:

```
declare variable $weblog as (entry)* external;
forseq $w in $weblog tumbling window
   start curItem $s when fn:true()
   end nextItem $e when
      $e/tstamp - $s/tstamp gt 'PT1H'
return fn:count($w[op eq "login"])
```

This query involves a time-based tumbling window. The XQuery 1.0 recommendation supports the subtraction of timestamps, and 'PT1H' is the ISO (and XQuery 1.0) way to represent a time duration of one hour. This query also works on an infinite web log for online monitoring of the log because the FORSEQ clause is non-blocking.

5.5.2 Stock Ticker

The second example shows how FORSEQ can be used in order to monitor an (infinite) stock ticker. For this example, it is assumed that the input is an infinite sequence of (stock) ticks; each stock tick contains the symbol of the stock (e.g., "YHOO"), the price of that stock, and a timestamp. The stock ticks are ordered by time. The query detects whenever a stock has gained more than ten percent in one hour.

```
declare variable $ticker as (tick)** external;
```

```
forseq $w in $ticker sliding window
   start curItem $f when fn:true()
   end   curItem $l when $l/price ge $f/price * 1.1
             and $l/symbol eq $f/symbol
where $f/tstamp - $l/tstamp le 'PT1H'
return $l
```

This query uses sliding windows in order to detect all possible sequences of ticks in which the price of the last tick is at least ten percent higher than the price of the first tick, for the same stock symbol. The `WHERE` clause checks whether this increase in price was within one hour.

5.5.3 Time-Outs

A requirement in some monitoring applications concerns the definition of time-outs. For example, a doctor should be notified if the blood pressure of a patient does not drop significantly ten minutes after a certain medication has been given. As another example, a supervisor should react when a firefighter enters a burning building and stays longer than one hour. In order to implement the firefighter example, two data streams are needed. The first stream records events of the following form: `<event person = ``name'' direction=''in/out'' tstamp=''timestamp''/>`. The second stream is a heartbeat of the form: `<tick tstamp=''timestamp''/>`. This heartbeat could be generated by a system-defined function of the XQuery engine.

```
declare variable $evts as (event)** external;
declare variable $heartb as (tick)** external;
forseq $w in fn:union($evts, $heartb) sliding window
   start $curItem $in when $in/direction eq ``in''
   end $curItem $last
     when $last/tstamp - $in/tstamp ge 'PT1H'
      or ($last/direction = ``out'' and
       $last/person = $in/person)
where $last/direction neq ``out''
return <alarm> { $in } </alarm>
```

In this query, a new window is started whenever a firefighter enters the building. A window is closed either when the firefighter exits the building or when an hour has passed.

An alarm is raised only in the latter case. This example assumes that the *fn:union()* function is implemented in a non-blocking way and consumes input continuously from all of its inputs.

5.5.4 Sensor State Aggregation

A frequent query pattern in sensor networks involves computing the current state of all sensors at every given point in time. If the input stream contains temperature measurements of the following form:

```
<temp id=''1''>10</temp>
<temp id=''2''>15</temp>
<temp id=''1''>15</temp>
```

then the output stream should contain a summary of the last measurement of each temperature sensor. That is, the output stream should look like this:

```
<values> <temp id=''1''>10</temp> </values>
<values> <temp id=''1''>10</temp>
         <temp id=''2''>15</temp> </values>
<values> <temp id=''1''>15</temp>
         <temp id=''2''>15</temp> </values>
```

This output stream can be generated using the following continuous query:

```
declare variable $sensors as (temp)** external;
forseq $w in $sensors landmark window
   start position $s when $s eq 1
   end when fn:true()
return <values> {
      for $id in fn:distinct-values($w/@id)
      return
           $w[@id eq $id][last()]
} </values>
```

Technically, within each window, the measurements are grouped by `id` of the sensor and the last measurement of each group is returned.

5.6 Implementation

This section describes how we extend the MXQuery engine[2], an existing Java-based open-source XQuery engine, in order to implement the FORSEQ clause and continuous XQuery processing. We use the extended MXQuery engine in order to validate all the use cases of [FKKT06] and run the Linear Road benchmark (Section 5.7).

5.6.1 MXQuery

The MXQuery engine was developed as part of a collaboration between Siemens and ETH Zurich. The main purpose of the MXQuery engine is to provide an XQuery implementation for small and embedded devices; in particular, in mobile phones and small gateway computers. Within Siemens, for instance, the MXQuery engine has been used as part of the Siemens Smart Home project in order to control lamps, blinds, and other devices according to personal preferences and weather information via web services. Recently, MXQuery has also been used as a reference implementation for the XQuery Update language and XQueryP the XQuery scripting extensions.

Since MXQuery has been designed for embedded systems, it has a simple and flexible design. The parser is a straightforward XQuery parser and creates an expression tree. In a functional programming language like XQuery, the expression tree plays the same role as the relational algebra expression (or operator tree) in SQL. The expression tree is normalized using the rules of the XQuery formal semantics [DFF+07]. After that, the expression tree is optimized using heuristics. (MXQuery does not include a cost-based query optimizer.) For optimization, MXQuery only implements a dozen of essential query rewrite rules such as the elimination of redundant sorts and duplicate elimination. The final step of compilation is code generation during which each expression of the expression tree is translated into an iterator that can be interpreted at run-time. As in SQL, iterators have an *open(), next(), close()* interface [Gra93]; that is, each iterator processes its input an *item* at a time and only processes as much of its input as necessary. The iterator tree is often also called *plan*, thereby adopting the SQL query processing terminology. Figure 5.6 gives a sample plan for the FOR query that raises an alarm when the temperature raises above ten degrees (first query of Section 5.4).

[2]The MXQuery engine can be downloaded via Sourceforge. MXQuery is short for MicroXQuery.

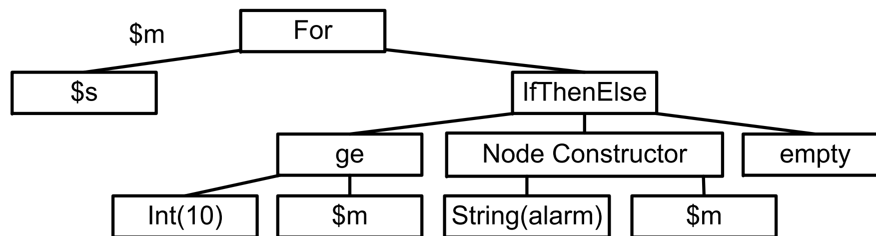

Figure 5.6: Example Plan (First Query of Section 5.4)

Like Saxon [Kay09], BEA's XQuery engine [FHK+04], and FluXQuery [KSSS04], MXQuery has been designed as a *streaming* XQuery engine. As shown in Figure 5.7, MXQuery can take input data from multiple sources; e.g., databases, the file system, or streaming data sources such as RSS streams or message queues. In order to take input from different sources, the data sources must implement the iterator API. That way, data from data sources can be processed as part of a query plan at run-time. MXQuery already has predefined iterator implementations in order to integrate SAX, DOM, StaX, and plain XML from the file system. Furthermore, MXQuery has predefined iterator implementations for CSV and relational databases. In order to access the data as part of an XQuery expression, an external XQuery variable is declared for each data source as shown in the examples of Section 5.5. A special Java API is used in order to bind the iterator that feeds the data from the data source to the external variable.

As shown in Figure 5.7, MXQuery also has an internal, built-in store in order to materialize intermediate results (e.g., for sorting of windows). In the current release of MXQuery, this store is fully implemented in main memory.

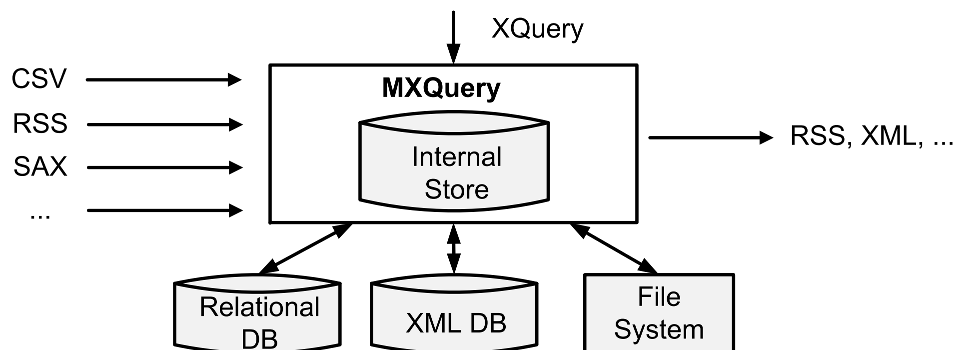

Figure 5.7: MXQuery Architecture

Although MXQuery has been particularly designed for embedded systems, its architecture is representative. The following subsections describe how we extend the MXQuery engine in order to implement the `FORSEQ` clause and work on infinite data streams. We believe that the proposed extensions are applicable to a wide range of XQuery engines.

5.6.2 Plan of Attack

In order to implement the FORSEQ clause, the following adaptations are made to the MXQuery engine:

- The parser is extended according to the production rules of Figure 5.3. This extension is straightforward and needs no further explanation.
- The optimizer is extended using heuristics to rewrite FORSEQ expressions. These heuristics are described in Section 5.6.4.
- The run-time system is extended with four new iterators that implement the three different kinds of windows and the general FORSEQ. Furthermore, the MXQuery internal main-memory store (Figure 5.7) is extended in order to implement windows. These extensions are described in Section 5.6.3.

To support continuous queries and infinite streams, the following extensions are made:

- The parser is extended in order to deal with the new ** annotation, which declares infinite sequences.
- The data flow analyses of the compiler are extended in order to identify errors such as the application of a blocking operator (e.g., count) to a potentially infinite stream.
- The run-time system is extended in order to synchronize access to data streams and merge/split streams.

The first two extensions (parser and type system) are straightforward and can be implemented using standard techniques of compiler construction [ASU86] and database query optimization [PHH92]. The third extension is significantly more complex, but not specific to XQuery. For our prototype implementation, we follow the approach taken in the Aurora project [ACC+03] as far as possible. Describing the details is beyond the scope of this thesis; the interested reader is referred to the literature on data stream management systems. Some features, such as persistent queues, recoverability, and security have not been implemented in MXQuery yet.

5.6.3 Run-Time System

5.6.3.1 Window Iterators

The implementation of the FORSEQ iterators for tumbling, sliding, and landmark windows is similar to the implementation of a FOR iterator: All these iterators implement second-order functions which bind variables and then execute a function on those variable bindings. All XQuery engines have some sort of mechanics to implement such second-order functions and these mechanics can be leveraged for the implementation of the FORSEQ iterators. The MXQuery engine has similar mechanics as those described in [FHK^{+}04] to implement second-order functions: In each iteration, a FORSEQ iterator binds a variable to a sequence (i.e., window) and then it executes a function on this variable binding. The function to execute is implemented as an iterator tree as shown in Figure 5.6 for a FOR iterator. This iterator tree encodes FOR, LET, and FORSEQ clauses (if any) as well as WHERE (using an IfThenElse iterator) ORDER BY and RETURN clauses. In general, the implementation of second-order functions in XQuery is comparable to the implementation of nested queries in SQL.

The only difference between a FOR iterator and a FORSEQ iterator is the logic that computes the variable bindings. Obviously, the FOR iterator is extremely simple in this respect because it binds its running variable to every item of an input sequence individually. The FORSEQ iterator for tumbling windows is fairly simple, too. It scans its input sequence from the beginning to the end (or infinitely for a continuous query), thereby detecting windows as shown in Figure 5.5. Specifically, the effective Boolean value of the conditions of the START or END clauses are computed for every new item in order to implement the state transitions of the automaton of Figure 5.5. These conditions are also implemented by iterator trees. The automata for sliding and landmark windows are more complicated, but the basic mechanism is the same and straightforward to implement.

5.6.3.2 Window Management

The most interesting aspects of the implementation of the FORSEQ iterators are the main memory management and garbage collection. The items of the input sequence are materialized in main memory. Figure 5.8 shows a (potentially infinite) input stream. Items of the input stream that have been read and materialized in main memory are represented as squares; items of the input stream which have not been read yet are

represented as ovals. The materialization of items from the input stream is carried out lazily, using the iterator model. Items are processed as they come in, thereby identifying new windows, closing existing windows, and processing the windows (i.e., evaluating the WHERE and RETURN clauses). This way, infinite streams can be processed. Full materialization is only needed if the query involves blocking operations such as ORDER BY, but such queries are illegal on infinite streams (Section 5.4).

According to the semantics of the different types of windows, an item can be marked in the stream buffer as *active* or *consumed*. An active item is an item that is involved in at least one open window. Correspondigly, consumed items are items that are not part of any active window. An item is immediately marked as consumed if no window is open and it does not match the START condition of the FORSEQ clause (Figure 5.5). Otherwise, an item is marked as consumed if all windows that involve this item have been fully processed; this condition can be checked easily by keeping a *position pointer* that keeps track of the minimum first postion of all open windows. In Figure 5.8, consumed items are indicated as white squares; active items are indicated as colored squares. In Figure 5.8, Window 1 is closed whereas Window 2 is still open; as a result only the items of Window 2 are marked as active in the stream buffer. (The postion pointer is not shown in Figure 5.8; it marks the start of Window 2.)

Consumed items can be garbage collected. To efficiently implement memory allocation and garbage collection in Java, the stream buffer is organized as a linked list of *chunks*. (For readibility, chunking and chaining are not shown in Figure 5.8.) That is, memory is allocated and de-allocated in chunks which can store several items. If all the items of a chunk are marked *consumed*, the chunk is released by de-chaining it from the linked list of chunks. Furthermore, all references to closed windows are removed. At this point, there are no live references left that refer to this chunk, and the space is reclaimed by the Java garbage collector.

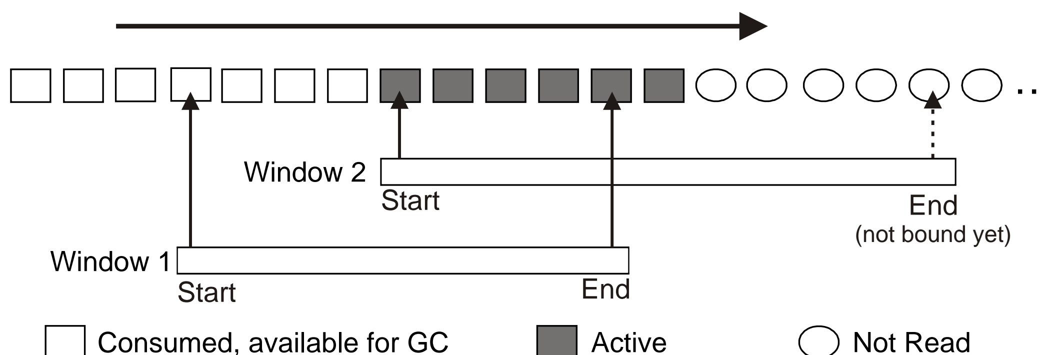

Figure 5.8: Stream Buffer

Windows are represented by a pair of pointers that refer to the first and last item of the window in the stream buffer. Open windows only have a pointer to the first item; the last pointer is set to NULL (i.e., unknown). Obviously, there are no open windows that refer to chunks in which all the items have been marked as *consumed*. As a result of this organization, items need to be materialized only once, even though they can be involved in many windows. Furthermore, other expressions that potentially require materialization can re-use the stream buffer, thereby avoiding copying the data.

5.6.3.3 General FORSEQ

The implementation of the general FORSEQ varies significantly from that of the three kinds of windows. In particular, representing a sub-sequence by its first and last item is not sufficient because the general FORSEQ involves the processing of non-contiguous sub-sequences. To enumerate all sub-sequences, our implementation uses the algorithm of Vance/Maier [VM96], including the bitmap representation to encode sub-sequences. This algorithm produces the sub-sequences in the right order so that no sorting of the windows is needed in the absence of an ORDER BY clause. Furthermore, this algorithm is applicable to infinite input streams. Additional optimizations are needed in order to avoid memory overflow for a general FORSEQ on infinite streams; e.g., the hopeless window detection, described in the next section.

5.6.4 Optimizations

This section lists several simple, but useful optimizations for our implementation. In particular, these optimizations are important in order to meet the requirements of the Linear Road benchmark (Section 5.7). Each of these optimizations serves one or a combination of the following three purposes: a.) reducing the memory footprint (e.g., avoid materialization); b.) reducing the CPU utilization (e.g., indexing); c.) improving streaming (e.g., producing results early). Although we are not aware of any streaming SQL engine which implements all these optimizations, we believe that most optimizations are also applicable for streaming SQL. A condition for most optimizations is the use of a predicate-based approach to define window boundaries. So far, no such streaming SQL proposals have been published.

The proposed list of optimizations is not exhaustive and providing a comprehensive study of the effectiveness of alternative optimization techniques is beyond the scope of this thesis. The list is only supposed to give an impression of the kinds of optimizations

that are possible. All these optimizations are applied in addition to the regular XQuery optimizations on *standard* XQuery expressions (e.g., [CAO06]). For example, rewriting reverse axes can be applied for FORSEQ queries and is then just as useful as it is for any other query.

5.6.4.1 Predicate Move-Around

The first optimization is applied at compile-time and moves a predicate from the WHERE clause into the START and/or END clauses of a FORSEQ query. The following example illustrates this optimization:

```
forseq $w in $seq landmark window
   start when fn:true()
   end when fn:true()
where $w[1] eq ``S'' and $w[last] eq ``E'' return $w
```

This query can be rewritten as the following equivalent query, which computes significantly fewer windows and can therefore be executed much faster and with lower memory footprint:

```
forseq $w in $seq landmark window
   start curItem $s when $s eq ``S''
   force end curItem $e when $e eq ``E''
return $w
```

5.6.4.2 Cheaper Windows

In some situations, it is possible to rewrite a landmark window query as a sliding window query or a sliding window query as a tumbling window query. This rewrite is useful because tumbling windows are cheaper to compute than sliding windows, and sliding windows are cheaper than landmark windows. This rewrite is frequently applicable if schema information is available. If it is known (given the schema), for instance, that the input sequence has the following structure “a, b, c, a, b, c, ...”, then the following expression

```
forseq $w in $seq sliding window
   start curItem $s when $s eq ``a''
   end curItem $e when $e eq ``c''
return $w
```

can be rewritten as the following equivalent expression:

```
forseq $w in $seq tumbling window
   start curItem $s when $s eq ``a''
   end curItem $e when $e eq ``c''
return $w
```

5.6.4.3 Indexing Windows

Using sliding and landmark windows, it is possible that several thousand windows are open at the same time. In the Linear Road benchmark, for example, this situation is the norm. As a result, with every new item (e.g., car position reading) the `END` condition must be checked several thousand times (for each window separately). Obviously, implementing such a check naïvely is a disaster. Therefore, it is advisable to use an index on the predicate of the `END` clause. Again, this indexing is illustrated with the help of an example:

```
forseq $w in $seq landmark window
  start curItem $s when fn:true()
  end curItem $e when $s/@id eq $e/@id
return $w
```

In this example, windows consist of all sequences in which the first and last items have the same *id*. (This query pattern is frequent in the Linear Road benchmark which tracks cars identified by their id on a highway.) The indexing idea is straightforward. An "@id" index (e.g., a hash table) is built on all windows. When a new item (e.g., a car position measurement with the *id* of a car) is processed, then that index is probed in order to find all matching windows that must be closed. In other words, the set of open windows can be indexed just like any other collection.

5.6.4.4 Improved Pipelining

In some situations, it is not necessary to store items in the stream buffer (Figure 5.8). Instead, the items can directly be processed by the `WHERE` clause, `RETURN` clause, and/or nested `FOR`, `LET`, and `FORSEQ` clauses. That is, results can be produced even though a window has not been closed yet. This optimization can always be applied if there is no `ORDER BY` and no `FORCE` in the `END` clause. It is straightforward to implement for tumbling windows. For sliding and landmark windows additional attention is required in order to coordinate the concurrent processing of several windows. The query of Section 5.5.1 is a good example for the usefulness of this optimization.

5.6.4.5 Hopeless Windows

Sometimes, it is possible to detect at run-time that the `END` clause or the predicate of the `WHERE` clause of an open window cannot be fulfilled. We call such windows *hopeless windows*. Such windows can be closed immediately, thereby saving CPU cost and main memory. The query of Section 5.5.2 is a good example for which this optimization is applicable: After an hour, an open window can be closed due to the `WHERE` condition even though the `END` condition of the window has not yet been met.

5.6.4.6 Aggressive Garbage Collection

In some cases, only one or a few items of a window are needed in order to process the window (e.g., the first or the last item). Such cases can be detected at compile-time by analyzing the nested expressions of the FLWOR expression (e.g., the predicates of the `WHERE` clause). In such situations, items in the stream buffer can be marked as *consumed* even though they are part of an open window, resulting in a more aggressive chunk-based garbage collection. A good example for this optimization is the query of Section 5.5.3.

5.7 Experiments and Results

5.7.1 Linear Road Benchmark

To validate our implementation of `FORSEQ` and continuous XQuery processing, we implement the Linear Road benchmark [ACG+04] using the extended MXQuery engine. The Linear Road benchmark is the only existing benchmark for data stream management systems (DSMS). This benchmark is challenging. As of 2009, the results of only three compliant implementations have been published: Aurora [ACG+04], an (unknown) relational database system [ACG+04], and IBM Stream Core [J+06]. The Aurora and IBM Stream Core implementations are low-level, based on a native (C) implementation of operators or processing elements, respectively. The implementation of the benchmark on an RDBMS uses standard SQL and stored procedures, but no details of the implementation have been published. There is also an implementation of the benchmark using CQL [ABW06]; however, no results of running the benchmark with that implementation have been published. To the best of our knowledge, our implementation is the first compliant XQuery implementation of the benchmark.

The benchmark exercises various aspects of a DSMS, requiring window-based aggregations, stream correlations and joins, efficient storage and access to intermediate results and querying a large (millions of records) database of historical data. Furthermore, the benchmark poses real-time requirements: all events must be processed within five seconds.

The benchmark describes a traffic management scenario in which the toll for a road system is computed based on the utilization of those roads and the presence of accidents. Both toll and accident information are reported to cars; an accident is only reported to cars which are potentially affected by the accident. Furthermore, the benchmark involves a stream of historic queries on account balances and total expenditures per day. As a result, the benchmark specifies four output streams: Toll notification, accident notification, account balances, and daily expenditures. (The benchmark also specifies a fifth output stream as part of a travel time planning query. No published implementation has included this query, however. Neither have we.)

The benchmark specification contains a data generation program which produces a stream of events composed of car positions and queries. The data format is CSV which is natively supported by MXQuery. Three hours worth of data are generated. An implementation of the benchmark is compliant if it produces the correct results and fulfills the five seconds real-time requirement. The correctness of the results are validated using

a validation tool so that load shedding or other load reduction techniques are not allowed.[3] Fulfilling the real-time requirements becomes more and more challenging over time: With a scale factor of 1.0, the data generator produces 1,200 events per minute at the beginning and 100,000 events per minute at the end.

The benchmark specifies different scale factors *L*, corresponding to the number of expressways in the road network. The smallest *L* is 0.5. The load increases linearly with the scale factor.

5.7.2 Benchmark Implementation

As mentioned in the previous section, our benchmark implementation is fully in XQuery, extended with `FORSEQ` and continuous queries. In a first attempt, we have implemented the whole benchmark in a single XQuery expression; indeed, this is possible! However, MXQuery was not able to optimize this huge expression in order to achieve acceptable (i.e., compliant) performance. As a consequence, we now (manually) partition the implementation into eight continuous XQuery expressions and five (temporary) stores; i.e., a total of 13 *boxes*. Figure 5.9 shows the corresponding workflow: The input stream, produced by the Linear Road data generator, is fed into three continuous XQuery expressions which in turn generate streams which are fed into other XQuery expressions and intermediate stores. Binding an input stream to an XQuery expression is done by *external* variable declarations as specified in the XQuery recommendation [BCF+07] and demonstrated in the examples in Section 5.5. This approach is in line with the approaches taken in [ACG+04, J+06], the only other published and compliant benchmark implementations. Aurora, however, uses 60 boxes (!).

Seven threads are used in order to run the continuous XQuery expressions and move data into and out of data stores. Tightly coupled XQuery expressions (with a direct link in Figure 5.9) run in the same thread. The data stores are all main-memory based (not persistent and not recoverable) using a synchronized version of the stream buffer described in Section 5.6.

[3] In our experiments, we encounter the same bugs with the validation tool and data generator as reported in [J+06]. Otherwise, all our results validated correctly.

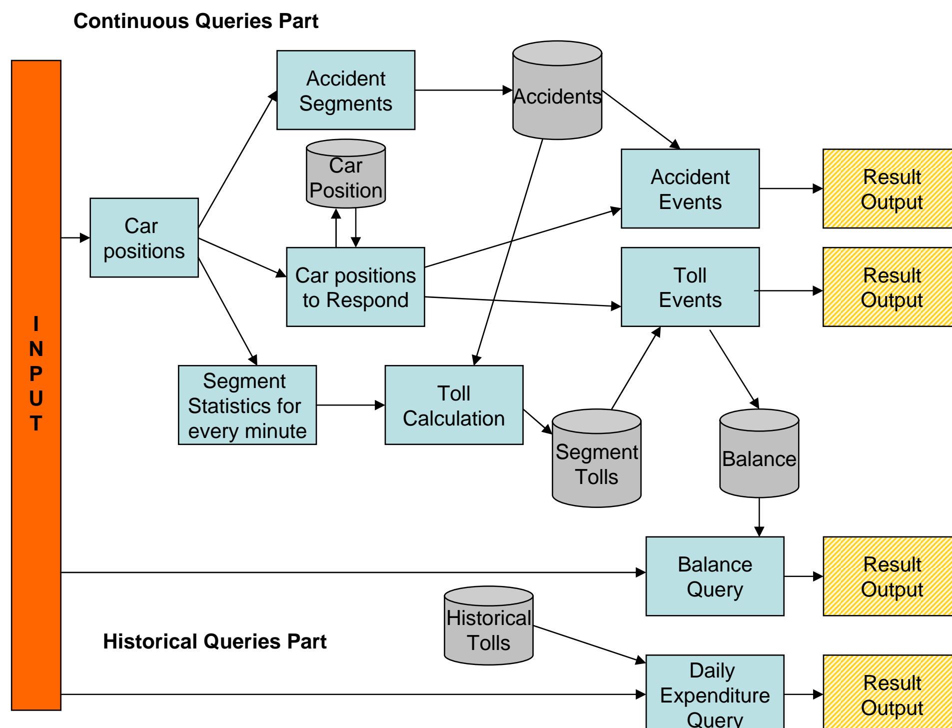

Figure 5.9: Data Flow of LR Implementation

5.7.3 Results

The implementation of the benchmark is evaluated on a Linux machine with a 2.2 GHz AMD Opteron processor and 4GB of main memory. Our hardware is comparable to the machines used in [ACG+04] and [J+06]. A Sun JVM in Version 1.5.0_09 is used, the maximum heap size is set to 2 GB which corresponds to the available RAM used in the experiments reported in [ACG+04],[J+06]. The results can be summarized as follows for the different scale factors **L**:

- *L=0.5 - 2.5:* MXQuery is fully compliant.
- *L=3.0:* MXQuery is not compliant. The response time exceeds the five seconds.

The best published results so far are compliant with an *L* of 2.5 [ACG+04, J+06]. These implementations are low-level C implementations that do not use a declarative language (such as SQL or XQuery). An *L* of 2.5 is still out of reach for our implementation. However, the differences are surprisingly small (less than a factor of 2) given that our focus has been to extend a general-purpose XQuery engine whereas those implementations

directly target the Linear Road benchmark. Also, MXQuery is written in Java which comes with a performance penalty.

The only compliant SQL implementation of the benchmark [ACG+04] is at an *L* of 0.5 (contrasting an *L* of 1.5 of our XQuery implementation). The maximum response times of the SQL implementation at *L* 1.0 and 1.5 are several orders of magnitude worse than the benchmark allows (2031 and 16346 seconds, respectively). Details of that SQL implementation of the benchmark are not given; however, it seems that the overhead of materializing all incoming events in a relational database is prohibitive. As part of the STREAM project, a series of CQL queries are published in order to implement the benchmark. However, no performance numbers have ever been published using the CQL implementation (as of September 2008). In summary, there does not seem to be a SQL implementation of the benchmark that beats our XQuery implementation. Fundamentally, there is no reason why either SQL or XQuery implementations would perform better on this benchmark because essentially the same optimizations are applicable to both languages. Due to the impedance mismatch between streams and relations, however, it might be more difficult to optimize streaming SQL because certain optimizations must be implemented twice (once for operators on streams and once for operators on tables).

5.8 Related Work

As mentioned in the introduction, window queries and data-stream management have been studied extensively in the past; a survey is given in [GO03]. Furthermore, there have been numerous proposals to extend SQL; the most prominent examples are AQuery [LS03], CQL [ABW06], and StreaQuel [CCD+03]. StreamSQL [Str07] is a recent activity (started in November 2006) that tries to standardize streaming extensions for SQL. As part of all this work, different kinds of windows have been proposed. In our design, we have take care that all queries that can be expressed in these SQL extensions can also be expressed in a straightforward way using the proposed XQuery extensions. In addition, if desired, special kinds of streams such as the i-streams and d-streams devised in [ABW06] can be implemented using the proposed XQuery extensions. Furthermore, we have adopted several important concepts of those SQL extensions such as the window types. Nevertheless, the work on extending SQL to support windows is not directly applicable to XQuery because XQuery has a different data model and supports different usage scenarios. Our use cases, for instance, involve certain patterns, e.g., the definition of window boundaries using general constraints (e.g.,

on authors of RSS postings) that cannot be expressed in any of the existing SQL extensions. All SQL extensions published so far only enable specifying windows based on size or time constraints; those SQL extensions are thus not expressive enough to handle our use cases, even if the data is relational. Apparently, StreamSQL will adopt the predicate-based approach, but nothing has been published so far. (The StreamSQL documentation in [Str07] still uses size and time constraints only.)

Recently, there have also been proposals for new query languages in order to process specific kinds of queries on data streams. One example is SASE [WDR06] which has been proposed to detect patterns in RFID streams; these patterns can be expressed using regular expressions. Another proposal is WaveScript [GMN+07], a functional programming language in order to process signals in a highly scalable way. While such languages and systems are useful for particular applications, the goal of this work is to provide general-purpose extensions to an existing main-stream programming language. Again, we have made sure in our design that all the SASE and WaveScript use cases can be expressed using the proposed XQuery extensions; however, our implementation does not scale as well for those particular use cases as the SASE and Wavescope implementations.

There have been several prototype implementations of stream data management systems; e.g., Aurora [ACC+03], Borealis [AAB+05], Cayuga [DGH+06], STREAM [ABW06], and Telegraph [CCD+03]. All that work is orthogonal to the main contribution of this chapter. In fact, our implementation of the linear road benchmark makes extensive use of the techniques proposed in those projects.

The closest related work is the work on positional grouping in XQuery described in [Kay06]. This work proposes extensions to XQuery in order to layout XML documents, one of the usage scenarios that also drove our design. The work in [Kay06] is inspired by functionality provided by XSLT in order to carry out certain XML transformations. However, many of our use cases on data streams cannot be expressed using the proposed extensions in [Kay06]; our proposal is strictly more expressive. Furthermore, the work of [Kay06] does not discuss any implementation issues. Another piece of related XML work discusses the semantics of infinite XML (and other) streams [MLT+05]. That work is orthogonal to our work.

5.9 Summary

This chapter presented two extensions for XQuery: Windows and continuous queries. Due to their importance, similar extensions have been proposed recently for SQL, and several ideas of those SQL extensions (in particular, the types of windows) have been adopted in our design. Since SQL has been designed for different usage scenarios, it is important that both SQL and XQuery are extended with this functionality: Window queries are important for SQL; but they are even more important for XQuery! We have implemented the proposed extensions in an open source XQuery engine and ran the Linear Road benchmark. The benchmark results seem to indicate that XQuery stream processing can be implemented as efficiently as SQL stream processing and that there is no performance penalty for using XQuery.

Presently, the Windowing extension has been accepted for the upcoming XQuery 1.1 standard [CR08] with some syntactical changes. The semantics however remain the same except for the landmark window and the general sub-sequences described in Section 5.3.3. The concept of the landmark window and the general sub-sequences have been excluded from the recommendation for simplicity. However, the working group might include both in future revisions of XQuery if users demand for it. The syntax as adopted by the W3C XQuery working group can be found in Appendix A.2

Chapter 6

Conclusion

Cloud computing has become one of the fastest growing fields in computer science. It promises virtually infinite scalability and 100% availability at low cost. To achieve high availability at low cost, most solutions are based on commodity hardware, are highly distributed, and designed to be fault-tolerant against network and hardware failures. However, the main success factor of cloud computing is not technology-driven but economical. Cloud computing allows companies to outsource the IT infrastructure and to acquire resources on demand. Thus, cloud computing not only allows companies to profit from the economics of scale and the leverage effect of outsourcing but also avoids the common over-provisioning of hardware [PBA+08].

Although the advantages for deploying web-based database applications in the cloud are compelling, they come with certain limitations. Different providers offer different functionality and interfaces, which makes it hard to port applications from one provider to another. Systems sacrifice functionality and consistency to allow for better scaling and availability. In the following, we summarize the main contributions of this thesis towards the development of Database Management Systems on top of cloud infrastructure (Section 6.1). We then conclude this thesis by discussing ongoing and future work in Section 6.2.

6.1 Summary of the Thesis

The main contributions of this dissertation are linked to the following three topics: How to build a database on top of cloud infrastructure, consistency rationing, and an XQuery extension in order to build applications with XQuery. In the following, we outline the main contributions in each of those areas.

1. **Building a Database on Top of Cloud Infrastructure:** The first part of the thesis investigates how a transactional database system can be built on top of cloud infrastructure services while preserving the main characteristics of cloud computing (i.e., scalability, availability and cost efficiency). We defined a set of basic services and a corresponding (abstract) API as a generalization of existing cloud offerings. Based on this API, we proposed a shared-disk architecture for building web-based database applications on top of cloud infrastructure. Furthermore, we presented several alternative consistency protocols which preserve the design philosophy of cloud computing and trade cost/availability for a higher level of consistency.

   The experimental results show that the shared-disk architecture preserves the scalability even for strong consistency levels. Furthermore, the results demonstrate that cloud computing and, in particular, the current offerings of providers such as Amazon are not attractive for high-performance transaction processing if strong consistency is important; such application scenarios are still best supported by conventional database systems. The proposed architecture works best and most cost-efficiently for web-based applications where the workload is hard to predict and varies significantly. Traditional database architectures require to provision the hardware resources for the expected peak performance which is often orders of magnitudes higher than the average performance requirements (and possibly even the real peak performance requirements). The here proposed architecture enables scaling by simply adding new nodes, or in case the client runs the database stack, no interaction at all.

2. **Consistency Rationing:** Consistency rationing constitutes a new transaction paradigm, which not only defines the consistency guarantees on the data instead of at transaction level, but also allows for switching consistency guarantees automatically at run-time. A number of use cases and according policies for switching the consistency level are presented. The basic policies switch the consistency level based on fixed thresholds or time. The advanced policies force the system to dynamically adapt the consistency level by monitoring the data and/or gathering temporal statistics of the data. Thus, consistency becomes a probabilistic guarantee which can be balanced against the cost of inconsistency. The proposed methods have been implemented on top of the presented cloud database architecture. The experimental results show that Consistency Rationing has the potential not only to significantly lower the overall cost, but to improve the performance of a cloud-based database system at the same time.

3. **Windowing for XQuery:** Through the success of XML for communication, meta-data, and documents inside web applications, XQuery/XScript has been proposed as a unified programming languages for web-based application. XQuery is able to run on all layers of the application stack [FPF+09, Mar09, FGK03, Ora09, CCF+06], is highly optimizable and parallizable, and is able to work with structured and semi-structured data which is increasingly important with the growing amount of user-generated content. Although XQuery is turing-complete and already used as a programming language in the database and middle-tier [Mar09, FGK03, Ora09, 28m09], it lacks support for window queries and continuous queries. Window queries and continuous queries, however, play an important role for communication data, meta-data and documents. For example, the analysis of a web logs, formatting and combining of RSS feeds, monitoring of services, pagination of results, are all best solved by windowing and/or continuous queries.

   In the last part of the thesis we address this missing feature and propose two extensions to XQuery 1.0: The syntax and semantics of a new window clause and a simple extension to the XQuery data model in order to use XQuery for continuous queries. We show that the extensions can be implemented and integrated into existing XQuery engines. Additionally, we report the results of running the Linear Road benchmark on top of an open source XQuery engine which had been enhanced with the proposed extensions. The benchmark results confirm that the proposed XQuery extensions can be implemented efficiently. By now, the windowing extension has been accepted from the W3C XQuery working group for the upcoming XQuery 1.1 specification.

6.2 Ongoing and Future Work

Below, we outline several interesting topics of ongoing work and discuss possibilities for future research.

XQuery as a Platform: The proposed architectures and protocols of Chapter 3 are concerned with the lower levels of a database management system running on top of cloud infrastructure. At the same time, the new characteristics of cloud storage systems as a disk and the virtualization of machines also changes higher layers of the database system. For instance, it is not possible to carry out chained I/O in order to scan through several pages on S3 or to assume that the database stack is running on a single dedicated machine. It follows that, the cost models of query optimizers and the

query operators need to be revisited. In addition, there are still a number of optimization techniques conceivable in order to reduce the latency of applications (e.g., caching and scheduling techniques).

The use of XQuery as a programming language for the cloud also raises further challenges. Next to the query optimization techniques, the application framework, deployment or developing tools provide many new research questions.

Part of these research questions are now addressed in the 28msec's Sausalito product. The vision of 28msec is to provide a platform to easily develop, test, deploy, and host web applications using XQuery. The architecture of Sausalito incorporates the work of Chapter 3 and is deployed on Amazon's Simple Storage Service (S3) and Elastic Cloud Computing (EC2).

Cloudy: An additional research question which arises from the work of Chapters 3 and 4 concerns potential changes in the architecture when the infrastructure is owned and the different (dynamically changing) consistency levels and query processing could be integrated into the service. Although first systems start to provide more functionality such as PNUTS [CRS+08], MegaStore [FKL+08], or MS SQL Azure [Mic09], they either restrict the scalability (e.g., Azure) or provide restricted functionality and relaxed consistency guarantees (e.g., MegaStore and PNUTS). Integrating the advanced functionality into a cloud system is particularly interesting, as it allows to directly influence and extend the underlying techniques. However, it would also change the role of the user from a cloud consumer to a cloud provider, which might significantly increase the numbers of required resources.[1]

Cloudy is an on-going research project which starts adressing these research questions by building a new highly scalable cloud database service. With Cloudy, we are also investigating how other guarantees of a cloud storage system, besides consistency, can be relaxed. For instance, it might be beneficial to relax the durability property for certain data types (e.g., the loss of logging data or a user recommendation might be tolerable, whereas user accounts should never be lost). Furthermore, owning the infrastructure as a cloud provider also makes room for other optimization such as data placement, replication strategies or energy efficiency.

Benchmarking the Cloud: Section 3.8 already stated, that traditional benchmarks (like the TPC benchmarks) have to be adjusted for analyzing the novel cloud service. Traditional benchmarks report the maximum performance under a particular workload

[1] Even though it is possible for the user to host his own infrastructure inside Amazon, it still requires more administration and resources than using existing reliable infrastructure services.

for a fixed configuration. A cloud service should dynamically adapt to the load which is not captured by a (static) maximum performance number used in traditional benchmarks. A benchmark for the cloud should in particular test the new characteristics of cloud computing (i.e., scalability, pay-per-use and fault tolerance). In [BKKL09] we did a first step and proposed initial ideas for a new benchmark to test database applications in the cloud.

Appendix A

Appendix

A.1 Node Splitting

A page split requires to store two pages in an atomic operation; the old and new page, with half of the data each and the according new linking. [Lom96] proposes a 2 phase commit protocol in order to achieve this split atomically, which can not directly applied in our architecture. Algorithm 11 shows how to achieve a page split in the proposed architecture with Advanced Queues. Without advanced queues, the check in Line 10 has to be adjusted.

Algorithm 11 Page Checkpoint Protocol with Splitting

Require: page P, Collection C, $PropPeriod$, $X \leftarrow$ max nb of log records

1: **if** acquireXLock(P.Uri) **then**
2:     $StartTime \leftarrow$ CurrentTime()
3:     $V \leftarrow$ get-if-modified-since(P.Uri, P.Timestamp)
4:     **if** $V \neq Null$ **then**
5:         $P \leftarrow V$
6:     **end if**
7:     $M \leftarrow$ receiveMessage(P.Uri, X)
8:     **for all** messages m in M **do**
9:         **if** m is a SPLIT message **then**
10:             **if** m.Uri $\neq P$.nextUri $\wedge$ m.SplitKey $< P$.MaxKey **then**
11:                 delete(m.Uri)
12:                 deleteQueue(m.Uri)
13:             **end if**
14:         **else**
15:             **if** m.Key $> P$.Key **then**
16:                 sendMessage(P.Next, m)
17:             **else**
18:                 apply m to P
19:             **end if**
20:         **end if**
21:     **end for**
22:     **if** size(P) $> PAGESIZE$ **then**
23:         (P_2, $SplitKey$) $\leftarrow$ splitPage(P)
24:         *//Sending the split message is for recovery*
25:         sendMessage(P.Uri, pair(P_2.Uri, SplitKey))
26:     **end if**
27:     **if** CurrentTime() - $StartTime < LockTimeOut - PropPeriod$ **then**
28:         **if** $SplitKey \neq Null$ **then**
29:             createAdvancedQueue(P_2.Uri)
30:             put(P_2.Uri, P.Data)
31:         **end if**
32:         put(P.Uri, P.Data)
33:         **if** CurrentTime() - $StartTime < LockTimeOut - PropPeriod$ **then**
34:             **for all** messages m in M **do**
35:                 deleteMessage(P.Uri, m.Id)
36:             **end for**
37:         **end if**
38:     **end if**
39: **end if**

A.2 XQuery 1.1 Window BNF

The windowing BNF as adopted by the W3C for XQuery 1.1 [CR08].

$$
\begin{aligned}
FLWORExpr &::= InitialClause\ IntermediateClause*\ ReturnClause \\
InitialClause &::= ForClause \mid LetClause \mid WindowClause \\
IntermediateClause &::= InitialClause \mid WhereClause \mid GroupByClause \mid OrderByClause \mid CountClause \\
WindowClause &::= \text{"for"} (TumblingWindowClause \mid SlidingWindowClause) \\
TumblingWindowClause &::= \text{"tumbling"}\ \text{"window"}\ \text{"\$"} VarName\ TypeDeclaration?\ \text{"in"}\ ExprSingle \\
&\quad WindowStartCondition\ WindowEndCondition? \\
SlidingWindowClause &::= \text{"sliding"}\ \text{"window"}\ \text{"\$"} VarNameTypeDeclaration?\ \text{"in"}\ ExprSingle \\
&\quad WindowStartCondition\ WindowEndCondition \\
WindowStartCondition &::= \text{"start"}\ WindowVars\ \text{"when"}\ ExprSingle \\
WindowEndCondition &::= \text{"only"}?\ \text{"end"}\ WindowVars\ \text{"when"}\ ExprSingle \\
WindowVars &::= (\text{"\$"} CurrentItem)?\ PositionalVar?\ (\text{"previous"}\ \text{"\$"} PreviousItem)?\ (\text{"next"}\ \text{"\$"} NextItem)? \\
CurrentItem &::= QName \\
PreviousItem &::= QName \\
NextItem &::= QName
\end{aligned}
$$

Figure A.1: XQuery 1.1 - Windowing

List of Tables

2.1 Infrastructure as a Service Providers and Products 12
2.2 Platform as a Service Providers and Products 14

3.1 Amazon Web Service Infrastructure Prices (as of 20 Oct 2009) 37
3.2 Read Response Time, Bandwidth of S3, Varying Page Size 38
3.3 Read Response Time, Bandwidth of S3, Varying Page Size 40
3.4 Consistency Protocols 59
3.5 Cost (milli-$) and Time (secs) for Bulk-Loading 10000 Products from EC2 107

4.1 Data categorization 133

List of Figures

2.1 Cloud Continuum 11

2.2 Canonical Web Architecture (modified from [Con02, CBF03]) 17

2.3 Cloud Web Architecture(modified from [Con02, CBF03]) 19

2.4 Web application layers 28

3.1 Shared-Disk Architecture 50

3.2 Basic Commit Protocol 60

3.3 Avg. Response Time (secs), EU-BTree and EU-SDB 95

3.4 Avg. Write Response Time (secs), EU-BTree and EU-SDB, Update WIs Only 96

3.5 Avg. Response Time (secs), EC2-BTree and EC2-SDB 97

3.6 Avg. Response Time (secs), EC2-BTree and EC2-SDB, Update WIs Only 98

3.7 Cost per WI (milli-$), EU-BTree and EU-SDB 99

3.8 Cost per WI (milli-$), EU-BTree Cost factors 99

3.9 Cost per WI (milli-$), EC2-BTree and EC2-SDB 100

3.10 Cost per WI (milli-$), EC2-BTree Cost Factors 101

3.11 Web Interactions per Second (WIPS), Varying Number of EC2 Application Servers 103

3.12 Cost per WI (milli-$), EC2-BTree, Varying Page Size 105

3.13 Avg. Response Time (secs), EC2-BTree, Varying Page Size 105

3.14 Cost per WI (milli-$), EC2-BTree, Varying TTL 106

3.15 Avg. Response Time (secs), EC2-BTree, Varying TTL 106

4.1 Cost (incl. the penalty cost) per trx [$/1000], Vary guarantees 135
4.2 Cost per trx [$/1000], Vary penalty cost 136
4.3 Response time [ms] 137
4.4 Cost per trx [$/1000], Vary policies 138
4.5 Response time [ms], Vary policies 138
4.6 Runtime $, Vary threshold 139
4.7 Oversells, Vary threshold 139
4.8 Overall $, Vary threshold 140

5.1 Annoying Authors: XQuery 1.0 148
5.2 Annoying Authors: Extended XQuery 149
5.3 Grammar of Extended FLWOR Expression 150
5.4 Window Types 151
5.5 Window Automaton 154
5.6 Example Plan (First Query of Section 5.4) 164
5.7 MXQuery Architecture 164
5.8 Stream Buffer 167
5.9 Data Flow of LR Implementation 174

A.1 XQuery 1.1 - Windowing 187

List of Algorithms

1 Basic Commit Protocol 62
2 Page Checkpoint Protocol 64
3 Collection Checkpoint Protocol 67
4 Atomicity - Commit 75
5 Atomicitiy - Recovery 76
6 Advanced Atomicitiy - Commit 80
7 Advanced Atomicitiy - Recovery 81
8 GSI - BeginTransaction 82
9 GSI - BufferManager - Get Page 83
10 GSI - Commit 85
11 Page Checkpoint Protocol with Splitting 186

Bibliography

[10g09] 10gen. MongoDB. http://www.mongodb.org, Nov. 2009.

[28m09] 28msec, Inc. Sausalito. http://sausalito.28msec.com, Feb. Feb. 2009.

[AAB+05] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The Design of the Borealis Stream Processing Engine. In *Proc. of CIDR*, 2005.

[Aba09] Daniel J. Abadi. Data Management in the Cloud: limitations and opportunities. IEEE Data Engineering Bulletin, 32(1), 2009.

[ABW06] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *VLDB Journal*, 15(2):121–142, 2006.

[ACC+03] Daniel Abadi, Donald Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal*, 12(2):120–139, 2003.

[ACG+04] Arvind Arasu, Mitch Cherniack, Eduardo F. Galvez, David Maier, Anurag Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear Road: A Stream Data Management Benchmark. In *Proc. of VLDB*, pages 480–491, 2004.

[AFG+09] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb. 2009.

[Ama09a] Amazon. Amazon Web Services. http://aws.amazon.com/, Sep. 2009.

[Ama09b] Amazon. Amazon Web Services: Overview of Security Processes. http://awsmedia.s3.amazonaws.com/pdf/AWS_Security_Whitepaper.pdf, Nov. 2009.

[Ama09c] Amazon. Elastic Block Store (EBS). http://aws.amazon.com/ebs/, Aug. 2009.

[Ama09d] Amazon. Simple Storage Service (S3). http://aws.amazon.com/s3/, Aug. 2009.

[Ama09e] Amazon. SimpleDB Developer Guide (API Version 2009-04-15). http://docs.amazonwebservices.com/AmazonSimpleDB/2009-04-15/DeveloperGuide/, Apr. 2009.

[Apa08] Apache Software Foundation. ZooKeeper. http://hadoop.apache.org/zookeeper/, Aug. 2008.

[ASU86] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Atw07] Mark Atwood. A Storage Engine for Amazon S3. MySQL Conference and Expo, 2007. http://fallenpegasus.com/code/mysql-awss3.

[AYBB+09] Sihem Amer-Yahia, Chavdar Botev, Stephen Buxton, Pat Case, Jochen Doerre, Michael Dyck, Mary Holstege, Jim Melton, Michael Rys, and Jayavel Shanmugasundaram. XQuery and XPath Full Text 1.0. http://www.w3.org/TR/xpath-full-text-10/, Jul. 2009.

[BBG+95] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proc. of ACM SIGMOD*, pages 1–10, 1995.

[BCE+08] Vinayak R. Borkar, Michael J. Carey, Daniel Engovatov, Dmitry Lychagin, Till Westmann, and Warren Wong. XQSE: An XQuery Scripting Extension for the AquaLogic Data Services Platform. In *Proc. of ICDE*, pages 1229–1238, 2008.

[BCF+07] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, Jan. 2007.

[BCLW06] Vinayak R. Borkar, Michael J. Carey, Dmitry Lychagin, and Till Westmann. The BEA AquaLogic data services platform (demo). In Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors, *Proc. of ACM SIGMOD*, pages 742–744. ACM, 2006.

[BFF+07] Irina Botan, Peter M. Fischer, Daniela Florescu, Donald Kossmann, Tim Kraska, and Rokas Tamosevicius. Extending XQuery with Window Functions. In *Proc. of VLDB*, pages 75–86, 2007.

[BFG+08] Matthias Brantner, Daniela Florescu, David A. Graf, Donald Kossmann, and Tim Kraska. Building a database on S3. In *Proc. of ACM SIGMOD*, pages 251–264, 2008.

[BGM94] Daniel Barbará and Hector Garcia-Molina. The Demarcation Protocol: A Technique for Maintaining Constraints in Distributed Database Systems. *VLDB Journal*, 3(3):325–353, 1994.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[Bil92] Alexandros Biliris. The Performance of Three Database Storage Structures for Managing Large Objects. In *Proc. of ACM SIGMOD*, pages 276–285, 1992.

[Bje04] Havard K. F. Bjerke. Grid Survey. Technical Report, CERN, Aug. 2004.

[BKKL09] Carsten Binnig, Donald Kossmann, Tim Kraska, and Simon Loesing. How is the Weather tomorrow? Towards a Benchmark for the Cloud. In *Proc. of DBTest Workshop (SIGMOD 2009)*, 2009.

[BLP05] Shane Balfe, Amit D. Lakhani, and Kenneth G. Paterson. Trusted Computing: Providing Security for Peer-to-Peer Networks. In *Peer-to-Peer Computing*, pages 117–124, 2005.

[BosG09] Zuse Institute Berlin and onScale solutions GmbH. Scalaris: Distributed Transactional Key-Value Store. http://code.google.com/p/scalaris/, Aug. 2009.

[Bre00] Eric A. Brewer. Towards Robust Distributed Systems. In *Proc. of PODC*, page 7, 2000.

[BS77] Rudolf Bayer and Mario Schkolnick. Concurrency of Operations on B-Trees. *Acta Informatica*, 9(1):1–21, 1977.

[BT04] Jason E. Bailes and Gary F. Templeton. Managing P2P security. *ACM Communication*, 47(9):95–98, 2004.

[Bun09] Bungee Labs. Bungee Connect. http://www.bungeeconnect.com, Sept. 2009.

[Bur06] Michael Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proc. of OSDI*, pages 335–350, 2006.

[Cal08] Brad Calder. Windows Azure Storage - Essential Cloud Storage Services. In *Microsoft Professional Developers Conference (PDC)*, 2008.

[CAO06] Dunren Che, Karl Aberer, and Tamer Özsu. Query Optimization in XML Structured-Document Databases. *VLDB Journal*, 15(3):263–289, 2006.

[CB04] Thomas Connolly and Carolyn Begg. *Database Systems: A Practical Approach to Design, Implementation and Management (International Computer Science Series)*. Addison Wesley, 4 edition, May 2004.

[CBF03] Stefano Ceri, Aldo Bongio, and Piero Fraternali. *Designing Data-Intensive Web Applications*. Morgan Kaufman, 2003.

[CCD+03] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Frederick Reiss, and Mehul A. Shah. TelegraphCQ: Continuous Dataflow Processing. In *Proc. of ACM SIGMOD*, page 668, 2003.

[CCF+06] Jon Chamberlin, Michael J. Carey, Daniela Florescu, Donald Kossmann, and Jonathan Robie. XQueryP: Programming with XQuery. In *Proc. of XIME-P*, 2006.

[CCI09] CCIF. Cloud Computing Interoperability Forum (CCIF). http://groups.google.com/group/cloudforum, Sep. 2009.

[CDF+09] Don Chamberlin, Center Michael Dyck, Daniela Florescu, Jim Melton, Jonathan Robie, and Jérôme Siméon. XQuery Update Facility 1.0. http://www.w3.org/TR/xquery-update-10/, Jun. 2009.

[CDG+06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proc. of OSDI*, pages 205–218, 2006.

[CEF+08] Don Chamberlin, Daniel Engovatov, Dana Florescu, Giorgio Ghelli, Jim Melton, Jerome Simeon, and John Snelson. XQuery Scripting Extension 1.0 - W3C Working Draft 3. http://www.w3.org/TR/xquery-sx-10/, Dec. 2008.

[CGMW94] Sudarshan S. Chawathe, Hector Garcia-Molina, and Jennifer Widom. Flexible Constraint Management for Autonomous Distributed Databases. *IEEE Data Engineering Bulletin*, 17(2):23–27, 1994.

[CGR07] Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live - An Engineering Perspective. In *Proc. of PODC*, pages 398–407, 2007.

[Con02] Jim Conallen. *Building Web Applications with UML*. Addison-Wesley, 2 edition, 2002.

[Cor09] LinkedIn Corporation. LinkedIn. http://www.linkedin.com/, Aug. 2009.

[Cou02] Transaction Processing Performance Council. TPC-W 1.8. http://www.tpc.org/tpcw/, Feb. 2002.

[CR08] Don Chamberlin and Jonathan Robie. XQuery 1.1 - W3C Working Draft 3, Dec. 2008.

[CRS+08] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.

[DAF+03] Yanlei Diao, Mehmet Altinel, Michael J. Franklin, Hao Zhang, and Peter Fischer. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. *ACM Transactions on Database Systems (TODS)*, 28(4):467–516, 2003.

[Dat09] Database Research Group, University of Mannheim. Natix: A native XML database management system. http://pi3.informatik.uni-mannheim.de/natix.html, Sep. 2009.

[Dea09] Jeff Dean. Handling Large Datasets at Google: Current Systems and Future Directions. http://research.yahoo.com/files/6DeanGoogle.pdf, 2009.

[DFF+07] Denise Draper, Peter Fankhauser, Mary Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. http://www.w3.org/TR/xquery-semantics/, Jan. 2007.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *ACM Communication*, 51(1):107–113, 2008.

[DGH+87] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic Algorithms for Replicated Database Maintenance. In *Proc. of PODC*, pages 1–12, 1987.

[DGH+06] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. Towards Expressive Publish/Subscribe Systems. In *Proc. of EDBT*, pages 627–644, 2006.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proc. of SOSP*, pages 205–220, 2007.

[Eco08] The Economist. Let it rise: A special report on corporate IT. *The Economist*, 2008.

[eXi09] eXist. eXist-db Open Source Native XML Database. http://exist-db.org/, 2009.

[EZP05] Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. Database Replication Using Generalized Snapshot Isolation. In *Proc. of SRDS*, pages 73–84, 2005.

[Fac09] Facebook. Cassandra. http://incubator.apache.org/cassandra/, Aug. 2009.

[Feu05] Steven Feuerstein. *Oracle PL/SQL Programming*. O'Reilly & Associates, 4 edition, 2005.

[FGK03] Daniela Florescu, Andreas Grünhagen, and Donald Kossmann. XL: A Platform for Web Services. In *Proc. of CIDR*, 2003.

[FHK+04] Daniela Florescu, Chris Hillery, Donald Kossmann, Paul Lucas, Fabio Riccardi, Till Westmann, Michael J. Carey, and Arvind Sundararajan. The BEA Streaming XQuery Processor. *VLDB Journal*, 13(3):294–315, 2004.

[FK04] Ian Foster and Carl Kesselman, editors. *The Grid 2: Blueprint for a new computing infrastructure*. Elsevier, Amsterdam, 2004.

[FKKT06] Peter Fischer, Donald Kossmann, Tim Kraska, and Rokas Tamosevicius. FORSEQ Use Cases. http://www.dbis.ethz.ch/research/publications, 2006. Technical Report, ETH Zurich, November.

[FKL+08] JJ Furman, Jonas S Karlsson, Jean-Michel Leon, Alex Lloyd, Steve Newman, and Philip Zeyliger. Megastore: A Scalable Data System for User Facing Applications. Presentation at SIGMOD Products Day, Vacouver, Canada, 2008.

[FMM+06] Mary Fernandez, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh. XQuery 1.0 and XPath 2.0 Data Model (XDM), Oct. 2006.

[FPF+09] Ghislain Fourny, Markus Pilman, Daniela Florescu, Donald Kossmann, Tim Kraska, and Darin McBeath. XQuery in the Browser. In *Proc. of WWW*, pages 1011–1020, 2009.

[FZRL09] Ian T. Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud Computing and Grid Computing 360-Degree Compared. *CoRR*, 2009.

[Gar07] Simson Garfinkel. An Evaluation of Amazon's Grid Computing Services: EC2, S3, and SQS. Technical Report TR-08-07, Harvard University, 2007.

[Gar08] Gartner, Inc. Gartner Identifies Top Ten Disruptive Technologies for 2008 to 2012. http://www.gartner.com/it/page.jsp?id=681107, Mar. 2008.

[Gar09] Owen Garrett. Cloud Application Architectures. http://blog.zeus.com/the_zeus_blog/2009/05/cloud-application-architectures.html, May 2009.

[GDN+03] Lei Gao, Michael Dahlin, Amol Nayate, Jiandan Zheng, and Arun Iyengar. Application Specific Data Replication for Edge Services. In *Proc. of WWW*, pages 449–460, 2003.

[Gee06] David Geer. Will Software Developers Ride Ruby on Rails to Success? *IEEE Computer*, 39(2):18–20, 2006.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proc. of SOSP*, pages 29–43, 2003.

[GK85] Dieter Gawlick and David Kinkade. Varieties of Concurrency Control in IMS/VS Fast Path. *IEEE Database Engineering Bulletin*, 8(2):3–10, 1985.

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

[GLR05] Hongfei Guo, Per-Åke Larson, and Raghu Ramakrishnan. Caching with 'Good Enough' Currency, Consistency, and Completeness. In *Proc. of VLDB*, pages 457–468, 2005.

[GLRG04] Hongfei Guo, Per-Åke Larson, Raghu Ramakrishnan, and Jonathan Goldstein. Relaxed Currency and Consistency: How to Say "Good Enough" in SQL. In *Proc. of ACM SIGMOD*, pages 815–826, 2004.

[GMN+07] Lewis Girod, Yuan Mei, Ryan Newton, Stanislav Rost, Arvind Thiagarajan, Hari Balakrishnan, and Samuel Madden. The Case for a Signal-Oriented Data Stream Management System. In *Proc. of CIDR*, pages 397–406, 2007.

[GO03] Lukasz Golab and Tamer Özsu. Issues in Data Stream Management. *SIGMOD Record*, 32(2):5 –14, 2003.

[Goo08] Google. Google App Engine. http://code.google.com/appengine/, Dez. 2008.

[GR94] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, 1994.

[Gra93] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[GRS06] Giorgio Ghelli, Christopher Re, and Jérôme Siméon. XQuery!: An XML Query Language with Side Effects. In *EDBT Workshops*, pages 178–191, 2006.

[GSE+94] Jim Gray, Prakash Sundaresan, Susanne Englert, Kenneth Baclawski, and Peter J. Weinberger. Quickly Generating Billion-Record Synthetic Databases. In *Proc. of ACM SIGMOD*, pages 243–252, 1994.

[GT93] Ashish Gupta and Sanjai Tiwari. Distributed Constraint Management for Collaborative Engineering Databases. In *Proc. of CIKM*, pages 655–664, 1993.

[Hay08] Brian Hayes. Cloud Computing. *ACM Communication*, 51(7):9–11, 2008.

[Hel04] Joseph M. Hellerstein. Architectures and Algorithms for Interent-Scale (P2P) Data Management. In *Proc. of VLDB*, page 1244, 2004.

[Her09] Heroku, Inc. Heroku. http://heroku.com/, Sept. 2009.

[Hir09] Mikio Hirabayashi. Tokyo Cabinet. http://sourceforge.net/projects/tokyocabinet/, Sep. 2009.

[J+06] Navendu Jain et al. Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. In *Proc. of ACM SIGMOD*, 2006.

[JAF+06] Shawn R. Jeffery, Gustavo Alonso, Michael J. Franklin, Wei Hong, and Jennifer Widom. Declarative Support for Sensor Data Cleaning. In *Proc. of Pervasive Computing*, pages 83–100, 2006.

[JLB+04] Zamil Janmohamed, Clara Liu, Drew Bradstock, Raul F. Chong, Michael Gao, Fraser McArthur, and Paul Yip. *DB2 SQL PL: Essential Guide for DB2 UDB on Linux, UNIX, Windows, i5/OS, and z/OS*. IBM Press, 2 edition, 2004.

[JS82] Matthias Jarke and Joachim W. Schmidt. Query Processing Strategies in the PASCAL/R Relational Database Management System. In *Proc. of ACM SIGMOD*, pages 256–264, 1982.

[Kay06] Michael Kay. Positional Grouping in XQuery. In *Proc. of XIME-P*, 2006.

[Kay09] Michael Kay. SAXON: The XSLT and XQuery processor. http://saxon.sourceforge.net/, Jan. 2009.

[Kep04] Stephan Kepser. A Simple Proof for the Turing-Completeness of XSLT and XQuery. In *Proc. of Extreme Markup Languages*, 2004.

[KHAK09] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency Rationing in the Cloud: Pay only when it matters. In *Proc. of VLDB*, 2009.

[KKN+08] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-Store: A High-Performance, Distributed Main Memory. Transaction Processing System. In *Proc. VLDB Endowment*, volume 1, pages 1496–1499, 2008.

[KLL+97] David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proc. of STOC*, pages 654–663, 1997.

[Kos08] Donald Kossmann. Building Web Applications without a Database System. In *Proc. of EDBT*, page 3, 2008.

[Kra08] Tim Kraska. XQuery 1.1 Use Cases - W3C Working Draft 3 , Dec. 2008.

[KSSS04] Christoph Koch, Stefanie Scherzinger, Nicole Schweikardt, and Bernhard Stegmaier. FluXQuery: An Optimizing XQuery Processor for Streaming XML Data. In *Proc. of VLDB*, pages 1309–1312, 2004.

[Lab09] Morph Labs. Morph Application Platform (MAP). http://www.mor.ph/, Sep. 2009.

[Lam78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *ACM Communication*, 21(7):558–565, 1978.

[Lam98] Leslie Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[Lee08] Jason Lee. SQL Data Services - Developer Focus (Whitepaper). http://www.microsoft.com/azure/data.mspx, Jun. 2008.

[LGZ04] Per-Åke Larson, Jonathan Goldstein, and Jingren Zhou. MTCache: Transparent Mid-Tier Database Caching in SQL Server. In *Proc. of ICDE*, pages 177–189, 2004.

[LLJ08] Yijun Lu, Ying Lu, and Hong Jiang. Adaptive Consistency Guarantees for Large-Scale Replicated Services. In *Proc. of NAS*, pages 89–96, 2008.

[LMS05] Paul J. Leach, Michael Mealling, and Rich Salz. A Universally Unique IDentifier (UUID) URN Namespace. http://tools.ietf.org/html/rfc4122, Jul. 2005.

[Lom96] David B. Lomet. Replicated Indexes for Distributed Data. In *Proc. of PDIS*, pages 108–119, 1996.

[LS03] Alberto Lerner and Dennis Shasha. AQuery: Query Language for Ordered Data, Optimization Techniques, and Experiments. In *Proc. of VLDB*, pages 345–356, 2003.

[LY81] Philip L. Lehman and s. Bing Yao. Efficient Locking for Concurrent Operations on B-Trees. *ACM Transactions on Database Systems (TODS)*, 6(4):650–670, 1981.

[Mac07] Don MacAskill. Scalability: Set Amazon's Servers on Fire, Not Yours. Talk at ETech Conf, http://blogs.smugmug.com/don/files/ETech-SmugMug-Amazon-2007.pdf, 2007.

[Mar09] MarkLogic. MarkLogic Server. http://www.marklogic.com, Nov. 2009.

[MBB06] Erik Meijer, Brian Beckman, and Gavin M. Bierman. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *Proc. of ACM SIGMOD*, page 706, 2006.

[MCS96] Mark L. McAuliffe, Michael J. Carey, and Marvin H. Solomon. Towards Effective and Efficient Free Space Management. *SIGMOD Record*, pages 389–400, Jun 1996.

[Mer88] Ralph C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In *Proc. of CRYPTO*, pages 369–378, London, UK, 1988. Springer-Verlag.

[MG09] Peter Mell and Tim Grance. Nist working definition of cloud computing. http://csrc.nist.gov/groups/SNS/cloud-computing/index.html, Aug. 2009.

[Mic09] Microsoft. Azure Services. http://www.microsoft.com/azure/services.mspx, Aug. 2009.

[MLT+05] David Maier, Jin Li, Peter Tucker, Kristin Tufte, and Vassilis Papadimos. Semantics of Data Streams and Operators. In *Proc. of ICDT*, 2005.

[Moz09] Mozilla. Mozilla Developer Center - XPath. https://developer.mozilla.org/en/XPath, Aug. 2009.

[Off09] ServePath Corporate Office. GoGrid. http://www.gogrid.com, Sep. 2009.

[OLW01] Chris Olston, Boon Thau Loo, and Jennifer Widom. Adaptive Precision Setting for Cached Approximate Values. In *Proc. of ACM SIGMOD*, pages 355–366, 2001.

[O'N86] Patrick E. O'Neil. The Escrow Transactional Method. *ACM Transactions on Database Systems (TODS)*, 11(4):405–430, 1986.

[Ora09] Oracle. Oracle WebLogic integration. http://www.oracle.com/technology/products/weblogic/, Aug. 2009.

[OV99] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1999.

[PBA+08] Daryl C. Plummer, Thomas J. Bittman, Tom Austin, David W. Cearley, and David Mitchell Smith. Cloud Computing: Defining and describing an emerging phenomenon. Gartner Special Reports, 2008.

[PETCR06] Esther Palomar, Juan M. Estévez-Tapiador, Julio César Hernández Castro, and Arturo Ribagorda. Security in P2P Networks: Survey and Research Directions. In *EUC Workshops*, pages 183–192, 2006.

[PHH92] Hamid Pirahesh, Joseph Hellerstein, and Waqar Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *Proc. of ACM SIGMOD*, 1992.

[PS06] Kostas Patroumpas and Timos Sellis. Window Specification over Data Streams. In *Proc. of ICSNW*, 2006.

[Rac09] US Inc. Rackspace. The Rackspace Cloud Hosting Products. http://www.rackspacecloud.com/cloud_hosting_products, Sep. 2009.

[RD01] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proc. of Middleware*, pages 329–350. Springer-Verlag, 2001.

[Red09] Redis. Redis: A persistent key-value database. http://code.google.com/p/redis/, Aug. 2009.

[RG02] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw Hill Higher Education, 3rd edition, 2002.

[Rig08] RightScale LLC. Amazon's Elastic Block Store explained. http://blog.rightscale.com/2008/08/20/amazon-ebs-explained/, Aug. 2008.

[RMH96] Robbert Van Renesse, Yaron Minsky, and Mark Hayden. A Gossip-Style Failure Detection Service. In *Proc. of Middleware*, pages 55–70, 1996.

[Ros09] Max Ross. Transaction Isolation in App Engine. http://code.google.com/appengine/articles/transaction_isolation.html, Sep. 2009.

[RW04] Jeanne W. Ross and George Westerman. Preparing for utility computing: The role of IT architecture and relationship management. *IBM Systems Journal*, 43(1):5–19, 2004.

[S+96] Michael Stonebraker et al. Mariposa: A Wide-Area Distributed Database System. *VLDB Journal*, 5(1):048–063, 1996.

[Sal09] Salesforce.com, Inc. Salesforce.com. http://www.salesforce.com, Sep. 2009.

[Sen08] Soumitra Sengupta. SQL Data Services: A lap around. In *Microsoft Professional Developers Conference (PDC)*, 2008.

[SL76] Dennis G. Severance and Guy M. Lohman. Differential Files: Their Application to the Maintenance of Large Databases. *ACM Transactions on Database Systems*, 1(3):256–267, 1976.

[SMA+07] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In *Proc. of VLDB*, pages 1150–1160, 2007.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proc. of SIGCOMM*, pages 149–160. ACM, 2001.

[SPP98] Edward A. Silver, David F. Pyke, and Rein Peterson. *Inventory Management and Production Planning and Scheduling*. Wiley, 3 edition, 1998.

[SRS04] Shetal Shah, Krithi Ramamritham, and Prashant J. Shenoy. Resilient and Coherence Preserving Dissemination of Dynamic Data Using Cooperating Peers. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):799–812, 2004.

[SS05] Yasushi Saito and Marc Shapiro. Optimistic Replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.

[Sto86] Michael Stonebraker. The Case for Shared Nothing. *IEEE Data Engineering Bulletin*, 9(1):4–9, 1986.

[Str07] StreamSQL.org. StreamSQL documentation. http://streamsql.org/pages/documentation.html, Jan. 2007.

[TDP+94] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent B. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Proc. of PDIS*, pages 140–149, 1994.

[The09a] The Apache Software Foundation. HBase. http://hadoop.apache.org/hbase/, Aug. 2009.

[The09b] The Apache Software Foundation. The CouchDB Project. http://couchdb.apache.org/, Aug. 2009.

[The09c] The Apache Software Foundation. The Apache Hadoop project . http://hadoop.apache.org, Aug. 2009.

[The09d] The Voldemort Team. Voldemort - A distributed Database. http://project-voldemort.com/, Sep. 2009.

[TM96] Ann T. Tai and John F. Meyer. Performability Management in Distributed Database Systems: An Adaptive Concurrency Control Protocol. In *Proc. of MASCOTS*, page 212, 1996.

[Tod09] Todd Hoff (ed). High Scalability: Building bigger, faster more reliable websites. http://highscalability.com/, Aug. 2009.

[TS06] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2 edition, 2006.

[Tuu09] Ville Tuulos. Ringo - Distributed Key/Value Storage for Immutable Data. http://github.com/tuulos/ringo, Sep. 2009.

[UPvS09] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. A Survey of DHT Security Techniques. *ACM Computing Surveys*, 2009.

[VM96] Bennet Vance and David Maier. Rapid Bushy Join-Order Optimization with Cartesian Products. In *Proc. of ACM SIGMOD*, 1996.

[Vog07] Werner Vogels. Data Access Patterns in the Amazon.com Technology Platform. In *Proc. of VLDB*, page 1, Sep 2007.

[VRMCL09] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A Break in the Clouds: Towards a Cloud Definition. *Proc. of SIGCOMM*, 39(1):50–55, 2009.

[WB09] Craig D. Weissman and Steve Bobrowski. The Design of the Force.com Multitenant Internet. Application Development Platform. . In *Proc. of SIGMOD*, pages 889–896, 2009.

[WDR06] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-Performance Complex Event Processing over Streams. In *Proc. of ACM SIGMOD*, 2006.

[WV02] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems*. Morgan Kaufmann, 2002.

[Xen08] Xen. Xen Hypervisor. http://www.xen.org/, Dec. 2008.

[YBS08] L. Youseff, M. Butrico, and D. Da Silva. Towards a Unified Ontology of Cloud Computing. In *Grid Computing Environments Workshop (GCE08)*, 2008.

[YG09] Aravind Yalamanchi and Dieter Gawlick. Compensation-Aware Data types in RDBMS. In *Proc. of ACM SIGMOD*, 2009.

[YSY09] Fan Yang, Jayavel Shanmugasundaram, and Ramana Yerneni. A Scalable Data Platform for a Large Number of Small Applications. In *Proc. of CIDR*, 2009.

[YV00] Haifeng Yu and Amin Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. In *Proc. of OSDI*, pages 305–318, 2000.

[Zam09] Brett Zamir. XQuery USE ME (XqUSEme) 1.5. https://addons.mozilla.org/en-US/firefox/addon/5515, Sep. 2009.

[Zor09] Zorba. Zorba: The XQuery Processor. http://www.zorba-xquery.com/, Sep. 2009.

Curriculum Vitae: Tim Kraska

Affiliation during Doctoral Studies

12/2006 - 03/2010 **Research Assistant** in Computer Science, Department of Computer Science, Swiss Federal Institute of Technology Zurich (ETH), Switzerland

Education

since 03/2007 **PhD Student in Computer Science** within the System Group, Department of Computer Science, Swiss Federal Institute of Technology Zurich (ETH), Switzerland

06/2004 - 10/2006 **Master of Science in Information Systems (MScIS)** from the Westfälische Wilhelms-Universität Münster, Germany

03/2005 - 03/2006 **Master in Information Technology (MIT)** from the University of Sydney, Australia

10/2001 - 05/2004 **Bachelor of Science in Information Systems (BScIS)** from the Westfälische Wilhelms-Universität Münster, Germany

08/1991 - 06/2000 **Abitur** from the Leibniz-Gymnasium, Dortmund, Germany

Work Experience

04/2006 - 08/2006 **Software developer**, Fraunhofer Institute for Algorithms and Scientific Computing SCAI, Bonn, Germany

08/2002 to 06/2004 **Associated project manager and software developer**, Institute of Supply Chain Management, Westfälische Wilhelms-Universität Münster, Germany, a joint project with McKinsey & Company, Düsseldorf, Germany

03/1997 - 02/2003 **Consultant and software developer**, ICN GmbH & Co. KG, Dortmund, Germany

06/2001 - 01/2004 **Private lecturer**, Evangelisches Erwachsenenbildungswerk Westfalen und Lippe e.V. and Vereinigte Kirchenkreise Dortmund und Lünen, Germany

09/2000 - 07/2001 **Civilian service**, Vereinigte Kirchenkreise Dortmund und Lünen, Germany

Awards and Scholarships

07/2006 - 10/2006 **DAAD short-term scholarship**, Deutscher Akademischer Austauschdienst, Germany

07/2005 - 03/2006 **School of Information Technology Scholarship for outstanding achievements**, University of Sydney, Australia,

10/2005 **Siemens Prize for Solving an Industry Problem in Research Project Work** for the master thesis, University of Sydney, Australia,

Professional Affiliations

- Member of the W3C XQuery Working Group
- Member of the FLWOR Foundation and the Zorba XQuery Processor Team
- Member of ACM SIGMOD and the *Gesellschaft für Informatik* (GI), Germany