David Pereira Alves Neves Lopes

Graduate in Computer Science

WEAQL: Scaling Relational Databases through Weak Consistency

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser:	João Carlos Antunes Leitão, Invited Assistant Professor, Faculdade de Ciências e Tecnologia Universidade Nova de Lisboa
Co-adviser:	Rodrigo Seromenho Miragaia Rodrigues, Full Professor, Instituto Superior Técnico Universidade de Lisboa

Examination Committee

Chairperson:	Ludwig Krippahl, Assistant Professor
Members:	João Leitão, Invited Assistant Professor
	Miguel Matos, Post-doc Researcher

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

May, 2016

WeaQL: Scaling Relational Databases through Weak Consistency

This document was created using the (pdf)LaTeX processor, based in the "unlthesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/unlthesis [2] http://www.di.fct.unl.pt

To all my family and friends,

Acknowledgements

First, I would like to thank my advisor, Professor João Leitão, for all the support and commitment he put into this dissertation. The knowledge he shared with me and the feedback provided throughout the execution of this work had a crucial impact on the success of this dissertation. Second, I would like to thank my co-adviser, Professor Rodrigo Rodrigues, for giving me the opportunity to work with him. His support and experience were essential for the realization of this dissertation. In conjunction with Instituto de Engenharia de Sistemas e Computadores - Investigação e Desenvolvimento (INESC-ID), he also provided me with a research-grant in the context of the Dependable Cloud project, for which I am very grateful. Third, I would like to thank Cheng Li, for all the availability and expertise shared with me, particularly during the implementation phase of this dissertation. Finally, I also want to thank Professor Nuno Preguiça for all the useful contributions he gave to this dissertation.

To the Department of Informatics of the New University of Lisbon, thank you for providing me the conditions and environment which enabled me to successfully finish my Bachelor's and Master's degree.

To my family, thank you for all the support and motivation you gave me throughout my academic journey. In particular, I want to thank my parents and brother for having to deal with my bad temper, and for the endless demonstrations of love and care. To my fiancée, Filipa, thank you for all your tolerance and love. In the most discouraging times, you were were always there to support me and cheer me up.

Finally, I also want to thank all my friends for the great moments which we spent together. In particular, for all the lunches, snacks and coffees at MySpot and Mininova that you all shared with me, but also for all the sleepless nights spent in the Department of Informatics while listening to loud music.

Abstract

Today's Internet services handle massive amounts of requests and data from users all over the world. To achieve the performance and responsiveness required by these services, their providers and operators rely on replication to allow the processing of concurrent updates, many times resorting to weak consistency models to avoid the costs of replica coordination. In this context, NoSQL databases have emerged as a practical solution to deploy highly available and scalable applications. However, this solution places a heavy burden on developers as it is no longer possible for them to benefit from the convenient abstractions and the easy-of-use programming model provided by SQL-based relational databases, such as the transactional semantics and integrity constraints.

To explore how to efficiently scale SQL-based relational databases, in this dissertation, we explore the potential of using replication to improve their scalability and performance. In more detail, the aim of this dissertation is to develop a weakly consistent relational database replication middleware. First, we leverage Conflict-Free Replicated Data Types and transform every SQL statement into its commutative form, so that most of the time replication can be achieved in an expedited way by leveraging weak consistency. Second, to preserve SQL integrity constraints on top of weakly consistent replication, we strive to leverage multiple *coordination-free* mechanisms, while resorting to (efficient) coordination only when necessary.

We have evaluated a prototype of our solution using the TPC-C benchmark and show that when compared with two state of the art approaches for replicating SQL databases, our solution yields better performance and scalability.

Keywords: Replication; Weak Consistency; Relational Database; CRDTs;

Resumo

Os serviços de Internet da atualidade lidam com enormes quantidades de pedidos e dados provenientes de utilizadores espalhados por todo o mundo. Para atingir o desempenho e agilidade necessários nestes serviços, os provedores dos mesmos replicam-nos para tornar possível o tratamento dos pedidos de clientes de forma paralela, muitas vezes recorrendo a modelos de consistência fraca para evitar os custos envolvidos na coordenação das várias réplicas. Neste contexto, as bases de dados NoSQL emergiram como uma solução prática para suportar aplicações com elevados requisitos de disponibilidade e escalabilidade. No entanto, esta solução requer um esforço extra dos programadores no desenvolvimento das suas aplicações, uma vez que deixam de poder beneficiar das abstrações e modelo de programação suportado pelas bases de dados relacionais, como por exemplo, transações e restrições de integridade.

Para explorer soluções eficientes para escalar bases de dados relacionais, nesta dissertação exploramos o potencial de usar replicação para melhorar o seu desempenho e escalabilidade. Em maior detalhe, o objetivo desta dissertação é desenvolver uma base de dados relacional com consistência fraca. Em primeiro lugar, tiramos partido de *Conflict-Free Replicated Data Types*, ao transformar todos os comandos SQL provenientes da aplicação na sua forma comutativa, possibilitando assim que a sua replicação seja feita de forma ágil recorrendo a mecanismos com semântica de consistência fraca. Em segundo lugar, para preservar as principais restrições de integridade fornecidas pela linguagem SQL, utilizamos, sempre que possível, técnicas que não requerem coordenação e, recorremos a mecanismos de coordenação (eficientes) quando estritamente necessário.

Avaliamos um protótipo da nossa solução utilizando o *benchmark* TPC-C, e os resultados obtidos mostram que, quando comparados com duas conhecidas soluções para replicação de bases de dados relacionais, a nossa solução obteve melhorias substanciais em termos de desempenho e escalabilidade.

Palavras-chave: Replicação; Consistência Fraca; Base de Dados Relacionais; CRDTs

Contents

List of Figures

List of Algorithms

List of Tables

Acronyms

1SI One-Copy Snapshot Isolation.

2PC 2-Phase Commit.

CC Causal Consistency.

CmRDT Commutative Replicated Data Type.

CRDT Conflict-Free Replicated Data Type.

CvRDT Convergent Replicated Data Type.

EC Eventual Consistency.

EZK Extensible Zookeeper.

RC Read Committed.

RPC Remote Procedure Call.

RR Repeatable Read.

SEC Strong Eventual Consistency.

SI Snapshot Isolation.

SPOF Single Point of Failure.

VC Vector Clock.

Chapter 1

Introduction

This is the first chapter of this document. In Section 1.1 we start by giving the context of the work developed in this dissertation In Section 1.2 we motivate the relevance of this dissertation. In Section 1.3 we start to unveil the solution proposed in this work and identify the main contributions of this dissertation. We end this chapter in Section 1.4, in which we present the structure of the remainder of this document.

1.1 Context

The widespread access to the Internet has enabled millions of people to interconnect, store personal data, and disseminate information online. These Internet services are accessed by users from all over the world that demand responsiveness and an always-on experience. Examples of such services are social networks like Facebook or Twitter, and distributed storages such as Google Drive and Dropbox [Dro15; Inc15].

This class of applications manages massive amounts of data. For instance, Facebook reported over 4 billion items shared and 4.5 billion ”likes” per day in 2013 [Fac13]. Applications that manage data at this scale require massive infrastructures to cope with the high load of requests. Hence, these applications are mostly deployed in huge clusters hosted in data centers scattered around the world.

Replication is a widely used technique to achieve high-performance, fault-tolerance, availability, and low latencies in cloud services. First, distributing data across different nodes (i.e. replicas) improves performance (in terms of throughput) because replicas can process users’ requests concurrently. Second, if a replica fails (for instance due to a crash), the system remains accessible, thus providing availability and fault-tolerance. Finally, if replicas are geographically distributed, clients can contact the closest available replica to also reduce user-perceived latencies, an aspect that has been demonstrated to have a direct impact on the revenue of applications operators [SB09; Gom].

Ideally, replicated services would ensure strong consistency and availability at all times. However, the CAP theorem [GL02] has famously demonstrated this impossibility, in a distributed system prone to partitions. Due to the underlying infrastructures where modern Internet applications are deployed, it is not realistic to assume that partitions will not occur, and, for this reason, the implications of the CAP theorem force applications to make non-trivial decisions regarding the guarantees they provide to costumers [Hal15]. Providing strong consistency requires coordination among replicas to ensure that all replicas synchronously transit to the next (correct) state. Applications that deal with sensible information (e.g. online banking services) or applications that cannot afford to let users read stale data tend to opt for strong consistency. However, this approach greatly limits the scalability, performance, and response time of the system [Bai+14; Li+12; Li+14]. And more importantly, availability, as the fault of a single replica can lead the system to halt. For this reason, many online services execute under weak consistency models such as Eventual Consistency (EC) and Causal Consistency (CC). This allows replicas to process requests locally (and without coordination), thus boosting performance and lowering user perceived latencies, while ensuring availability in the presence of faults. To ensure state-convergence, updates are lazily disseminated in the background [Lad+92]. Systems that employ these consistency levels are known as weakly consistent systems.

A key aspect of most (if not all) online services is data persistence. To retrieve and store data, applications primarily resort to databases. In the literature, one can find two major classes of databases: relational databases and NoSQL databases. Relational databases attract application's developers by providing powerful abstractions and useful mechanisms (such as the ACID properties or integrity constraints), which eases the process of reasoning about the interactions between the application and the database. Furthermore, relational databases offer efficient support for the SQL language, which is very convenient for developers to store and retrieve application's data. With the fast growing of the Internet, and given the recent trend of the new Internet services, now more than ever applications must be able to sustain a high number of concurrent requests, and be able to scale to an unprecedented number of clients. Unfortunately, relational databases were conceptually designed for centralized environments and favor strong consistency semantics. Thus, their utilization in large scale distributed Internet services poses substantial performance and scalability challenges [Bai+14].

NoSQL databases, such as Cassandra [LM10], Riak [Tec15], and DynamoDB [DeC+07], have emerged as an alternative, practical solution to deploy highly available, highly performing, and scalable applications. However, NoSQL databases also have important drawbacks [Agu+15]. In particular, NoSQL systems depart from the ACID semantics and SQL interface provided by relational databases, thus pushing the complexity of handling inconsistencies, integrity violations, and complex queries to the application's programmers.

1.2 Motivation

Scaling databases to suit the needs of modern Internet services has been a topic of intense research among both academic and industry communities in the last decades [Bai+13; Bai+14; Cha+14; Llo+11; PA04]. The emergence of NoSQL databases and the subsequent "SQL vs NoSQL" dichotomy has also led to an on-going argument among the database community.

On the one hand, the NoSQL community used the implications driven by the CAP theorem as a persuasive argument to suggest the new NoSQL paradigm as the right approach to handle the requirements of the new class of Internet applications [Pim]. This led to the appearance of many NoSQL databases, conceptually identical, which are typically associated with weakly consistent systems [Cha+08; DeC+07; LM10; Mon; Tec15]. While this approach naturally promotes conflicts (due to uncoordinated, concurrent updates of the same object by different replicas), this approach is inherently scalable and available, two key requirements for large scale applications. NoSQL systems accept this trade-off; to resolve conflicts, these systems either use straightforward strategies such as *last-writer-wins* as in Cassandra [LM10], or expose conflicts to the application, which has the responsibility to resolve them to ensure state convergence as in [DeC+07] and [Ter+95]. Despite the clear benefits that NoSQL provide, the heavy burden that NoSQL systems places on developers have been gradually noticed as an important drawback [Cor+12; Pim]. As an example, Google received many complaints about how extremely complex and error-prone is to develop applications on top of its' NoSQL store, BigTable [Cha+08; EL14; Pim; Shu+13]. This fact was one of the reasons why Google decided to build Spanner [Cor+12], a distributed database that more closely resembles the relational databases programming model.

On the other hand, the SQL community has been continuously trying to scale relational databases to meet the requirements of large-scale applications. Most solutions proposals achieve this by sharding data and balancing the load across a set of shards. However, this sharding-based approach has a few well-known limitations: first, for some applications such as online social networks, it is extremely complex to devise an adequate partitioning scheme for the application data [Puj+10]; second, when executing transactions that span multiple machines, one has to pay the price of expensive coordination protocols such as 2-Phase Commit (2PC), which can limit performance and make it challenging to run certain workloads, such as long-running operations that access large data sets [Agu+15]; third, sharding heightens the need for replication for high availability, since a single node failure may lead the whole system to become unavailable, as the portion of data and state maintained by the failed node becomes unavailable.

While sharding is an essential technique to deploy large scale databases (such as the ones used by Facebook or Twitter), it is not so for lower to medium sized databases. In fact, the size of main memory in computers keeps increasing at a fast pace. For instance, Oracle recently reported that its new SPARC M6-32 systems can support up to 32 terabytes of memory [Ora]. This creates the opportunity to have the whole state and data associated with small to medium sized databases to permanently reside in memory, making the access to this data much faster, by avoiding the overheads of accessing durable storage.

Motivated by the notorious breach between NoSQL and SQL communities, in our work, we strive to bring together both approaches. We argue that it is not necessary to sacrifice performance and scalability to preserve an appealing programming model to developers, and, in contrast, it is not necessary to sacrifice the SQL semantics to support highly available, scalable, and performing services.

1.3 Proposed Solution

In a nutshell, the goal of this work is to improve the performance and scalability of SQL-based databases. To achieve this, we make the following key observations. First, we leverage the fact that many modern applications favor weak consistency models for improved performance. Second, we exploit the recent trends in hardware technologies to pursue an alternative (yet complementary) approach to sharding, in order to improve the performance and scalability of SQL-based databases.

The end result is materialized in WeaQL, a middleware replication layer that runs on top of SQL databases, and provides weakly consistent replication semantics to applications. To minimize coordination between replicas, WeaQL employs two complementary techniques. First, WeaQL transforms SQL statements used by applications to interact with the database into Conflict-Free Replicated Data Type (CRDT) operations, as to ensure that transactions become commutative and consequently state convergence can be achieved by design. Second, instead of partitioning the data between replicas, each replica hosts a full copy of the data, waiving the need of executing distributed transactions.

In designing WeaQL, we have to address a well-known tension between preserving SQL semantics – which requires coordination among replicas – and providing weak consistency – whose performance gains come from avoiding coordination, which can easily lead to violating SQL invariants. To address this challenge, we introduced a set of lightweight mechanisms to preserve SQL integrity constraints such as foreign key reference, uniqueness, and auto-increment values. These mechanisms provide a sweet spot between preserving the most important invariants of relational databases and minimizing the coordination overhead.

WeaQL was deployed in a cluster and compared against two popular solutions for database replication, MySQL Cluster [MyS15b] and Galera Cluster [Gal15]. To evaluate the performance and scalability of WeaQL, we used the TPC-C benchmark [Cou15].

1.3.1 Contributions

The main contributions of this dissertation are the following:

- A novel technique for scaling relational databases.

- A replication middleware layer that offers weak consistency semantics on top of relational databases with minimal efforts from programmers.
- A prototype of our solution.
- A experimental evaluation of our proposal comparing it with relevant state-of-the-art solutions.

1.4 Document Organization

The remainder of this document is organized as follows.

Chapter 2 presents fundamental topics related to the work proposed in this document, and discusses proposed solutions found in the literature.

Chapter 3 presents the design of WeaQL. In particular, we present the main challenges of our proposal and how WeaQL addresses them.

Chapter 4 describes the key components of WeaQL and its implementation details.

Chapter 5 we evaluate WeaQL, by comparing it to alternative solutions for database replication.

We close this document in **Chapter** 6, in which we draw conclusions about our approach, and point out possible lines of work for the future.

Chapter 2

Related Work

2.1 Overview

Relational databases have been a building block for many large scale Internet applications. Much of the popularity of relational databases comes from the programming model they expose to programmers. The support that relational databases offer to the SQL language enables developers to store and retrieve their data efficiently. Furthermore, developers can specify integrity constraints over their data to express the intended semantics of their applications, therefore eschewing the need of implementing these behaviors on the application-level. Finally, the transactional semantics hides concurrency anomalies to programmers and ease the development of concurrent applications. These applications provide services to millions of users that demand responsiveness, high availability, and fault-tolerance. To tackle these challenges, current systems replicate their data across multiple nodes (i.e. replicas).

Replication can be broadly defined as the act of maintaining multiple instances of the same object. Having multiple instances of an object improves performance, availability, and fault-tolerance. In the first case, the requests to access an object may be distributed over all the replicas that hold a copy of that object. Processing requests concurrently boost performance because more requests can be processed per time unit, thus improving throughput. Moreover, it allows workloads to be distributed over the available resources, which enables efficient resources utilization. Additionally, if one of the replicas becomes unavailable (for instance due to a failure) the object is still accessible by clients, thus providing availability and fault-tolerance.

Given that the state of an object usually over time, replication must also guarantee the consistency among different copies of the object. In a replicated environment, at some point, different replicas may hold different values for a given object. Depending on the application semantics, such occurrence may or may not be a problem. For instance, a flight reservation system should not allow inconsistencies between replicas because that could lead to overselling tickets to clients. In contrast, a simple information feed, such as the one offered by Facebook, tolerates inconsistencies without major negative consequences.

The amount of time replicas are allowed to diverge depends on four main aspects: *i*) the replication protocol used to disseminate information between replicas, *ii*) the consistency model employed by the application, *iii*) the deployment environment, and *iv*) the conditions of the network. Database replication protocols are discussed in Section 2.2. Consistency models are discussed in Section 2.3. Deployment environment is related to the underlying infrastructure in which replicas are executing (either in a cluster or in a geo-replicated setting). Network conditions are greatly influenced by the deployment environment. For instance, inside a cluster, network conditions deteriorate if links are congested. Links congestion arise due to the over subscription of communication channels (i.e., the ratio between the servers bandwidth to the total uplink bandwidth at the access layer [HM14]).

As a consequence of allowing replicas to individually change a global (and shared) state, conflicts and divergent states naturally emerge. Typically, these challenges are addressed by employing some deterministic technique such as *last-writer-wins*, or pushed to the application to be resolved [DeC+07; Ter+95]. CRDTs [Sha+11a; Sha+11b] go a step further and hide conflict resolution inside the data type itself, while still guaranteeing state-convergence, making these data types very useful for developers to use in applications which replicate their state over a set of nodes, and aim to taking benefits of weak consistency semantics to improve their performance.

The remainder of this chapter covers topics that are directly related to our work. Section 2.2 discusses replication in the context of relational databases. Section 2.3 comprises weakly consistent replication and the *trade-offs* associated. Section 2.4 introduces and discusses CRDTs. Finally, in Section 2.5, we discuss the processes of preserving invariants over replicated objects.

2.2 Database Replication

Before exploring databases in the context of a distributed system, for better understanding, we start by describing the system model in the context of a single replica.

2.2.1 System Model

Transactions are a fundamental feature of relational databases. A transaction is a logical unit of work that executes atomically (i.e., either all operations are successful or none is). Each transaction executes under an isolation level (one of the four ACID properties), which determines if the intermediate effects of ongoing transactions are exposed to other concurrent transactions. Stricter isolation levels protect transactions to see the changes induced by concurrent transactions, while more relaxed isolation levels do not. Serializability, a particular strong isolation level, is the traditional correctness criteria for objects in relational databases. For the sake of completeness, we now provide a formal definition of Serializability:

Definition 2.1. *Serializability: the execution of a set of concurrent transactions is equivalent to some serial execution of that same set of transactions [Bai15; Ber+86; Bor+11]*

Executing transactions under Serializability requires major efforts by the concurrency control protocol to provide an isolated environment where transactions execute. This greatly reduces performance (in this case transactions per time unit). To overcome this challenge, more relaxed isolation levels have been proposed. Snapshot Isolation (SI), Repeatable Read (RR), and Read Committed (RC) are popular isolation levels provided by most commercial database systems [MyS15a; Ora15; Pos15]. Applications can leverage these isolation levels to achieve better throughput, in exchange of relaxing the isolation between concurrent transactions.

To increase performance, many systems also distribute their data across multiple nodes. Such approach is further discussed in the following sections.

2.2.2 Distributed Databases

Databases are distributed mainly to improve performance, provide fault-tolerance, and increase the availability of a system. In the literature, one can find two main approaches to distribute the state of a database across a set of nodes, namely replication [Cec+08; Cha+14; DS06; Gal15; Gra+96; Gor; Wie+00] and partitioning [Agu+15; Cat11; DeC+07; MyS15b]. By replicating the state of the database across a set of nodes (i.e., replicas), each replica owns a copy of each object in the database. In contrast, partitioning consists in distributing the database state across a set of nodes (i.e., shards), meaning that each shard of the system owns only a portion of the whole data. Both approaches are complementary and have different advantages and disadvantages. Sharding on the one hand, heightens the need for replication for high availability since a single node failure may lead the whole system to become unavailable, as the portion of data and state maintained by the failed node becomes unavailable. Furthermore, operations that access multiple objects in the database might span multiple machines. In such case, one has to pay the price of expensive coordination protocols such as 2-Phase Commit, which can limit performance and make it challenging to execute certain workloads efficiently [Agu+15]. Replication on the other hand, might not be suitable for large databases, such as the ones operated by Facebook or Twitter. In this remainder of this section we will focus on replicated databases as they more closely relate to the work developed in this thesis.

As a consequence of replicating the same data across multiple nodes in the system, it is mandatory to ensure that they remain consistent across all replicas. Indeed, to ensure that replicas converge to the same final state, they cannot act as totally independent components, and must exchange information regarding the operations they executed in response to client's requests. Replication protocols, which are a fundamental aspect to ensure this, are discussed in the next section.

2.2.3 Database Replication Protocols

Database replication protocols can be discussed in two dimensions. The first dimension comprises *how* replicas transit state and includes active and passive replication. The second dimension refers to *where* and *when* updates might occur (Master-Slave or Multi-Master, and Eager or Lazy replication respectively).

The remainder of this section explores these protocols in more detail. We will use "update", "write", and "operation" interchangeably. Note that as read-only operations do not modify the state of the database, they are not relevant for ensuring replica convergence.

Active Replication

Active replication (also known as State Machine replication) [Sch90; Gor; Wie+00] is a non-centralized replication technique. The intuition at the core of this approach is that if all replicas execute the same sequence of operations they reach the same state. In active replication, operations are replicated to all replicas and executed independently in each one. To ensure state convergence, this approach requires two properties: *i*) that operations are executed in the same order in every replica, and *ii*) that operations are deterministic. It is easy to understand why both properties are necessary. First, if operations are executed in different orders, replicas may end up in an inconsistent state. For instance, when the effects of two write operations to the same object do not commute. Second, if operations are not deterministic, they might produce different outputs. A paradigmatic example of non-deterministic operations in the `SQL` language are time-related macros such as `now()`. Typical implementations of active replication resort to the properties that Atomic Broadcast provides [Gor; Wie+00] to ensure a total order of operations. For completeness, we now provide a formal definition of Atomic Broadcast:

Definition 2.2. *Atomic Broadcast: if a process delivers a message m before it delivers a message m', no process delivers m after delivering m'. [Gor; Wie+00]*

Ensuring the deterministic behavior of operations is usually left for the application programmers to guarantee. For instance, in the above example, the programmer could replace the function call `now()` with a pre-computed value instead.

Passive Replication

In contrast to active replication, in passive replication [Gor; Wie+00] operations are executed once in a single replica. The modifications induced by the execution of the

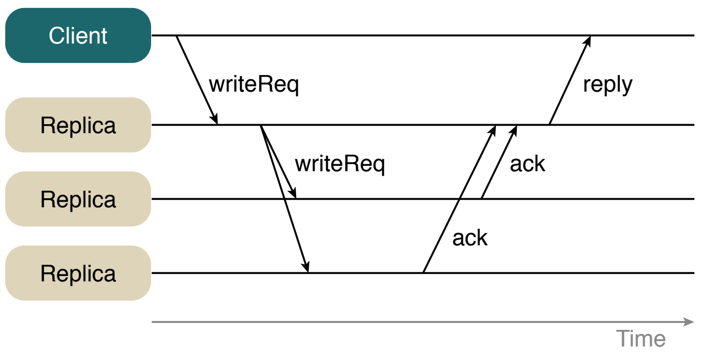


Figure 2.1: Eager Replication Protocol

operations (also called the operation's write-set) are then propagated to other replicas, which in turn install these modifications. Compared to active replication, this approach is more resource efficient because the operation need not to be executed, and only its side-effects must be applied.

Eager Replication

In an Eager Replication scheme (also known as synchronous replication) [KA00; Wie+00], after receiving an update, the replica immediately propagates it to others replicas and waits for them to acknowledge the correct execution of that operation. Upon receiving the acknowledgment from all replicas (or a subset of them, for instance, a majority [KA00]), the operation is said to be successfully completed and the client is notified. Eager replication assures serializable execution of requests because replicas are (synchronously) contacted (i.e., within the boundaries of the operation, from the perspective of the client). Hence, Eager Replication usually provides strong consistency. Figure 2.1 illustrates an eager replication protocol in action. As discussed previously, we can observe that before the client receives the reply to his request, all replicas of the system must acknowledge the correct execution of the operation. As a consequence, the latency of the operation is dominated by the slowest replica to acknowledge its execution.

Lazy Replication

By contrast to Eager Replication, in a Lazy Replication approach (also known as asynchronous replication, or optimistic replication) [DS06; Lad+92; SS05], operations are executed in a single replica and asynchronously disseminated to other replicas. Avoiding coordination during the execution of each operation greatly improves the response times as perceived by the user. This come at a cost of weakening the consistency guarantees since replicas are allowed to temporarily diverge. Hence, Lazy Replication does not ensure that clients read the latest version of an object (hence providing weak consistency guarantees). Figure 2.2 illustrates how requests are processed when resorting to lazy replication. Dashed arrows represent background communication.

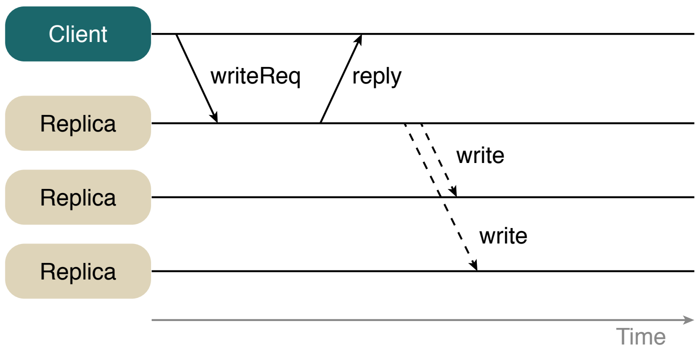


Figure 2.2: Lazy Replication Protocol

Master-Slave

In a Master-Slave setup (also known as Primary Copy) [Gor; Wie+00], one replica is designated the master replica while the others are called slaves. Only the master replica can process write requests. After receiving an update, the master replica propagates it accordingly to one of the schemes described above (i.e., either eagerly or lazily). Given that all write requests are directed to a single replica, ensuring a total order of updates is trivial and corresponds to the order by which the master replica processes the updates. A Master-Slave approach has two main drawbacks: *i*) the master replica is a potential bottleneck, particularly in a workload composed mostly of write operations, and *ii*) the master replica is a Single Point of Failure (SPOF). Master-Slave schemes are mostly used to improve read performance, given that read-only operations can be issued to any replica and coordination is not required for those operations. Figure 2.3 depicts the operation of a typical Master-Slave architecture. In more detail, we observe that write requests are directed to the master replica while read requests are issued to one of the slaves replicas.

Multi-Master

In a Multi-Master architecture (also known as update-anywhere replication), all replicas have the same capabilities. Writes can be performed by any replica, thus potentially boosting write throughput. This also enables users to contact any replica for the execution of write operations, which might be translated into lower latencies for those operations if the client is geographically close to the replica. Unfortunately, in this context coordination mechanisms are much more complex than in a Master-Slave architecture since concurrent updates on different replicas may conflict with each other [Gra+96]. Furthermore, if updates are lazily replicated to the other replicas in the system, conflicts may arise only after

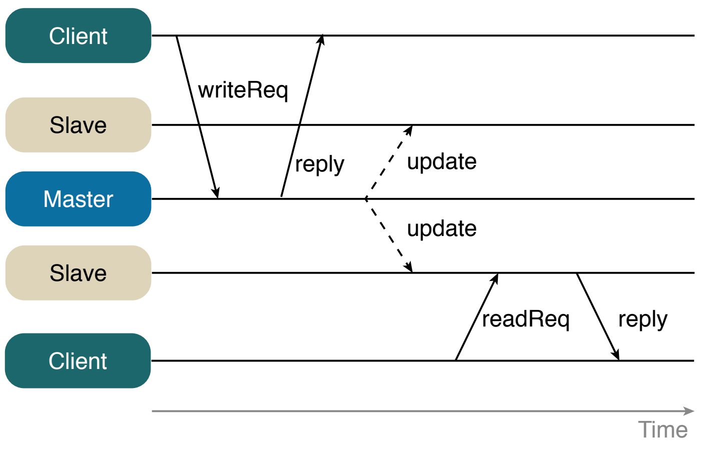


Figure 2.3: Master-Slave Replication Protocol

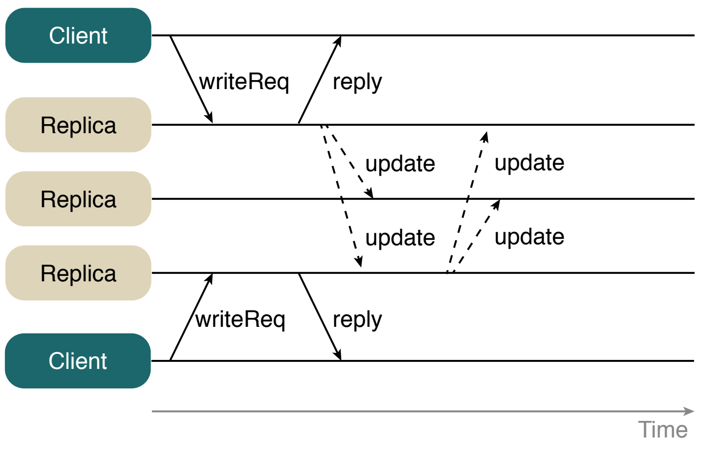


Figure 2.4: Multi-Master Replication Protocol

some replicas have already committed the operation. As a consequence, reconciliation techniques must be employed in order to resolve these conflicts [DeC+07; LM10; Ren+08; Sha+11b]. One of these techniques is discussed in detail in Section 2.4. Figure 2.4 depicts a Multi-Master architecture together with a lazy replication protocol.

2.2.4 Comparison with existing Systems

Plattner and Alonso introduced the concept of *satellites* databases [PA04; Pla+08]. In their system Ganymed, satellites databases serves read-only transactions while writes are tunneled to the master replica. This is achieved by a scheduler that separates and routes requests to the appropriate database replica. The same concept is applied by Salomie et al. in Multimed [Sal+11]. In more detail, Multimed treats a multi-core machine as a distributed system, where a subset of the cores act as satellites and another subset as the master replicas. In both cases, writes are centralized in a single node, which could become a bottleneck. Furthermore, in Ganymed, the presence of a network partition can prevent clients from communicating with the master replica leading to the unavailability of the system. WeaQL on the other hand, employs a multi-master architecture, therefore enabling every replica in the system to process write requests. This approach has the advantage to avoid potential bottlenecks on the master replica, particularly for write-intensive workloads. In addition, in contrast to Ganymed, the presence of network partitions does not render our the system unavailable, as long as clients can communicate with at least one replica.

In order to improve the scalability of replicated databases, Amza et al. employ a multi-master lazy replication strategy [Amz+03]. To minimize the number of conflicts between transactions, a central scheduler virtualizes the set of database replicas to the application servers as if it was a single database. All requests are tunneled to the scheduler, who is in responsible for minimizing conflicts and redirect requests to the adequate replicas. In order to minimize conflicts, the scheduler needs to know upfront which tables will be accessed by each transaction. As a consequence, applications must be adapted to provide this information to the scheduler. WeaQL also leverages a multi-master lazy replication technique to improve the scalability of replicated databases, however, we do not require programmers to feed any information regarding their transactions pattern. Besides that, contrary to the work of Amza et al., WeaQL does not require transactions to be totally ordered in a central scheduler in order to ensure replicas convergence.

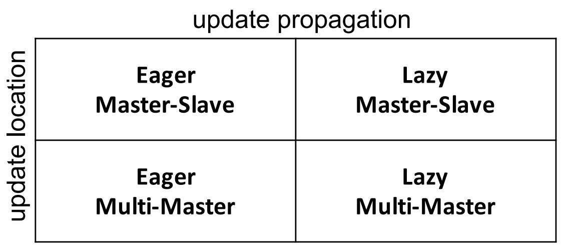
Lin et al. proposed a middleware solution that implements One-Copy Snapshot Isolation (1SI) [Lin+05], which guarantees that the SI properties are maintained in a distributed context, thus avoiding anomalies such as transaction inversion [DS06]. However, this solution requires synchronization on a per transaction basis, which hinders the scalability and performance of the system. In our work, we were willing to relax the Isolation properties in order to increase the performance and scalability of replicated databases, by exploring a lazy replication approach.

ConfluxDB, an extension of PostgreSQL [Cha+14], is also a multi-master lazy replicated system. It was designed to address the scalability challenges associated with Master-Slave solutions, by enabling the master replica to be partitioned across multiple nodes. The set of master partitions execute the 2PC protocol to agree on the decision regarding the execution of transactions, and produce an update log which is later consumed by the slaves replicas to be kept in sync. Similar to ConfluxDB, WeaQL also employs a multi-master lazy replication approach. However, in our work, every replica holds a full copy of the data, which enables WeaQL to completely avoid the execution of distributed transactions.

More recently, Aguilera et al. proposed Yesquel [Agu+15], a shard-based approach which aims to improve the scalability and performance of relational databases. To this end, Yesquel resorts to a new distributed data structure `YDBT`, specifically designed for the context of distributed databases. Furthermore, Yesquel preserves the SQL semantics by executing the 2PC consensus protocol for coordinating the shards involved in each transaction. WeaQL on the other hand, improves the scalability of relational databases by leveraging full replication. In addition, to preserve the SQL semantics, WeaQL exploits coordination-free techniques whenever possible, and only resorts to coordination when strictly required.

2.2.5 Summary

In the previous sections, we surveyed the multiple strategies found in the literature to replicate the state of a database across multiple replicas. In particular, we have discussed these strategies in two different dimensions. In the first dimension, we examined how replicas transit state and presented two approaches, namely active and passive replication. In the second dimension, we have considered how the updates are propagated between replicas and where they are allowed to be executed. Figure 2.5 summarizes the latter discussion and depicts the possible combinations of strategies that one might resort to when replicating the state of a database. We ended this section by comparing our work with existing solutions that resort to the database replication techniques described previously.



update location \ update propagation		
	Eager Master-Slave	Lazy Master-Slave
	Eager Multi-Master	Lazy Multi-Master

Figure 2.5: Database Replication Strategies Summary (adapted from [Wie+00])

2.3 Weakly Consistent Replication

The CAP theorem proposed by Eric Brewer and later demonstrated by Gilbert and Lynch [GL02] had a decisive impact on the design of distributed systems. The CAP theorem states that it is impossible to provide both strong consistency (i.e., linearizability [GL02]) and availability [BG13] while tolerating network partitions in a distributed system. We consider a system available if it continues to operate as expected, in the presence of partitions. For instance, we do not consider a system to be available if in the presence of partitions, it always responds with *no-op* to client's requests. In distributed systems comprised of hundreds of interconnected nodes, partitions will eventually arise [Hal15]. Therefore, for this class of systems, the choice has always been between consistency and availability. The trade-off between consistency and availability has been a hot topic in the research community for a long time.

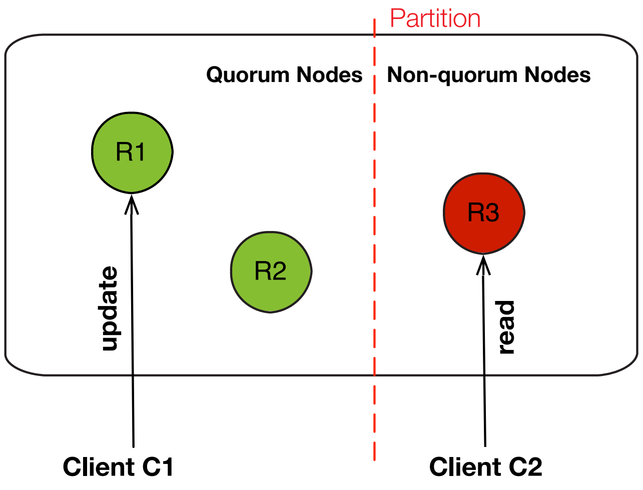


Figure 2.6: Partition in a Distributed System

Before exploring this trade-off, it is important to clarify that consistency, from the point of view of the CAP theorem, has a different meaning than the "Consistency" provided by the ACID properties of relational databases. While in a relational database Consistency refers to the guarantees that after the execution of each transaction the database remains in a consistent state, in distributed systems, Consistency refers to the guarantees the the applications provide when clients access their data [Vog08].

As an example, consider two processes P_1 and P_2. At some point, P_1 writes the value 1 in a particular data object. In a system that provides strong consistency, it is guaranteed that any subsequent requests that read this object (either from P_1 or P_2) will observe the updated value. To achieve strong consistency, replicas must coordinate and synchronously transit to the next state. In more detail, upon receiving an operation that modifies the state of the system, a subset of replicas (typically a quorum) must successfully acknowledge the execution of the operation before notifying the client. When an update is installed in a quorum of replicas, it is guaranteed that subsequent operations read the updated value because at least one of the replicas will participate in both quorums. This technique greatly reduces the performance of the system since more replicas must be contacted per operation, which has a severe impact on the latency of the operations. Furthermore, strong consistency is at odds with availability. Figure 2.6 depicts a situation of a system with three replicas which are experiencing a temporary partition. In this example, client C_1 contacts replica R_1 to issue an update operation. This operation is successful because R_1 is able to form a quorum with R_2. Client C_2, on the other hand, is connected to R_3, which is on the other side of the partition. Because R_3 cannot contact other replicas, operations in this replica will not succeed until the partition heals. Hence, this system is said to be unavailable. Systems that follow this approach are said to be "CP" since they favor consistency over availability when the system is experiencing a partition.

However, high availability has been an increasingly important aspect of online services. Indeed, it has been noticed [Gom] that performance and availability are two fundamental requirements when it comes to companies revenue and user satisfaction. In this context, weakly consistent replication has been the typical approach to meet these strict requirements. In systems which rely on weakly consistent replication, operations do not require upfront coordination (i.e., synchronization) among replicas. In more detail, operations are processed in the replica to where the request was originally issued (i.e., the local replica), and updates are lazily disseminated in the background to other replicas. Because replicas are not required to synchronize on a per operation basis, weakly consistent systems are able to achieve better performance (in terms of throughput) when compared to strongly consistent systems, while remaining available under the presence of partitions. In exchange, these systems cannot provide the properties associated with strong consistency, in particular that clients always observe the latest state of the system. We now resume to the example in figure 2.6. Client C_1 writes the value 1 to replica R_1. At this point, R_1 and R_2 are not in a consistent state because R_1 have not yet propagated the changes to its peers. Now consider the case where client C_2 issues a read operation to R_3. Because R_3 is not required to contact its peers to execute this operation, it will respond to this request based on his local state. Hence, despite the fact that the system is experiencing a partition, it can remain available if it desires so, at the cost of allowing clients to read stale data. To summarize, systems like the one just described favor Availability over Consistency in the presence of partitions, and as such are said to be “AP”.

The above discussion brings to light another aspect that must be taken into consideration. As a consequence of favoring Availability over Consistency, weakly consistent systems must be designed taking into consideration that conflicting operations might occur, for instance when two operations concurrently modify the same object on different replicas of the system. For instance, in the context of the example in Figure 2.6, a conflict would arise if C_1 and C_2 concurrently wrote different values for the same data object in replicas R_1 and R_2, respectively. In this case, a classic replication protocol that simply installs remote updates in a lazy fashion would leave both replicas in inconsistent final states, which would become permanent. To tackle this challenge, reconciliation processes must be adopted to ensure that replicas converge to the same state. In the literature, one can find two main classes of reconciliation policies. On the one hand, automatic conflict resolution techniques [LM10; Sha+11b], such as last-writer-wins, enables replicas to deterministically decide which update should be installed. Such approach can be implemented at the database level as in Cassandra [LM10], consequently hiding these conflicts from the application. On the other hand, application-specific reconciliation methods [DeC+07; Ter+95] enable a more fine-grained control over the conflict resolution policies and might be required to implement the adequate semantics envisioned by the developer. In this particular case, programmers must provide the merging procedures required to reconcile conflicting operations. In Section 2.4 we will further discuss the topic of conflict resolution policies.

Whether an application should opt for strong or weak consistency greatly depends on its requirements. For instance, an online banking system should opt for strong consistency to avoid negative balances on customer's accounts. In contrast, social networks like Twitter or Facebook tend to opt for weak consistency because it provides a better performance and user experience, without introducing major negative consequences. Picking strong or weak consistency is achieved effectively by the consistency model employed. In the literature, many consistency models have already been proposed: Linearizability [HW90]; Serializability [BG81]; Causal Consistency [Alm+13; Lam78; Llo+11]; Eventual Consistency [Vog08]. More recently, many systems have adopted a more flexible approach and offer multi-level consistency, enforcing both strong or weak consistency (e.g., on a per operation basis) [Kra+09; Li+12; Li+14; Ter+13]. The following sections discuss in more detail two particular weak consistency models, namely Eventual Consistency and Causal Consistency, which are closely related to our work.

2.3.1 Eventual Consistency

For completeness, we start by presenting an informal definition of Eventual Consistency.

Definition 2.3. *Eventual Consistency: **Eventual delivery:** An update executed at one node eventually executes at all nodes. **Termination:** All update executions terminate. **Convergence:** Nodes that have executed the same updates eventually reach an equivalent state. [EL14]*

Eventual Consistency is a particular weak consistency model. The only guarantee that Eventual Consistency provides is that if all updates activity stops, then eventually all replicas reach the same logical state (i.e., replicas are indistinguishable). While in theory Eventual Consistency seems to provide very few guarantees to applications, the empirical evidence shows that in fact, eventual consistent systems are often strongly consistent [BT11; EL14]. As an example, Vogels determined that the inconsistency window (i.e., the time required for an update be visible in all replicas) for the read operation on Amazon SimpleDB's was less than $500ms$ in most cases [Vog09].

Implementing an EC system is rather straightforward. Replicas execute requests locally to ensure responsiveness, and asynchronously (in the background) disseminate information about the operations they have executed. The process of exchanging information between replicas is often called anti-entropy because it allows replicas to diminish the divergence between them. The inconsistency window can be shortened by employing anti-entropy protocols more frequently.

2.3.2 Causal Consistency

Causal Consistency is another form of weak consistency. Causal Consistency enables the tracking of relationships between operations, which in some applications might be necessary to provide an intuitive behavior. For instance, consider a chat application with three users exchanging messages among them; user A sends m_1 to all other participants.

User B reads m_1 and replies with m_2. In this scenario, it does not make sense for user C to receive m_2 before m_1. Clearly, there is an implicit relationship between m_1 and m_2, because m_2 only "happened" in reply of m_1. Lamport in [Lam78] characterizes this relationship as "happened-before" relationship, which is denoted by $\rightarrow$. This relationship defines a partial order $\prec$ of events (in the above example, messages) in a distributed system [Lam78]. Formally, "happened-before" relationship is defined as follows [Lam78]:

Definition 2.4. ***(1)*** *If a and b are events in the same process, and a comes before b, then $a \rightarrow b$* ***(2)*** *If a is the sending of a message by one process and b is the receipt of the same message by another process, then $a \rightarrow b$* ***(3)*** *If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$.*

Causal Consistency [Alm+13; Lam78; Llo+11] guarantees that operations become visible to users in an order that respects their causal order. Hence, when the above scenario is executed under Causal Consistency, user C is guaranteed to observe $m2$ only after observing $m1$ because there is a partial order $\prec$ such as $m1 \prec m2$ (see definition 2.4).

Tracking "happened-before" relationships can be achieved by using Vector Clocks (VCs). Vector Clocks consist of a vector of N entries, where N is the number of nodes in the system [SS05]. In this context, each node typically generates new VCs values by incrementing its own entry on the VC. These values are then used to capture the relationship between operations in the system. As an example, when some replica R_i processes an operation op_1, it increments $VC[i]$ and associates the new vector clock VC_1 with op_1. Afterward, a second operation op_2 is issued to replica R_i, which correspondingly generates a second vector clock VC_2 to it. It is said that the vector clock VC_2 dominates VC_1 if $VC_2 \neq VC_1$ and $\forall i \in 1...N, VC_2[i] \geq VC_1[i]$. In other words, operation op_2 "happened before" op_1 if VC_2 dominates VC_1. If neither vector clock dominates the other, the operations are said to be concurrent [SS05].

Causal Consistency has gained much popularity and is now employed in many geo-replicated systems [Alm+13; Llo+11; Llo+13]. The reasons are three-fold. First, preserving the causality between operations promotes a more intuitive behavior of applications, as illustrated in the previous example where users exchanged messages among them. Second, implementing Causal Consistency can be achieved with a modest performance penalty when compared to Eventual Consistency [BG13]. Third, it is also known that Causal Consistency is the strongest consistency model that is available in the presence of network partitions [Mah+11].

2.3.3 Comparison with existing Systems

In the last decade, the appearance of NoSQL databases has led many weakly consistent systems to be developed, both by academia and industry [Cha+08; Coo+08; DeC+07; Fit04; LM10; Llo+11; Llo+13; Tec15]. In Dynamo [DeC+07], each object is identified by a key and is replicated across N replicas (hence tolerating up to $N-1$ failures). Replicas are placed in a ring and each one is responsible for a subset of the key-space. Read and write operations specify a parameter - R and W respectively - that correspond to the number of replicas that must acknowledge the request before notifying the client. Maximum availability and minimum response times are achieved by setting $W = R = 1$. Dynamo is used by Amazon to store and retrieve shopping carts data. Google's BigTable [Cha+08] and Yahoo's PNUTS [Coo+08] are another examples of non-relational databases designed to achieve high scalability. In contrast, WeaQL provides high availability by replicating the full state of the database over a set of replicas and employing weakly consistent replication among them. Furthermore, WeaQL preserves the SQL interface, which is more convenient for programmers to store and retrieve application's data when compared to the interface provided by NoSQL stores.

Li et al. proposed Gemini [Li+12], a middleware layer that supports a two-fold consistency model called RedBlue Consistency. In Gemini operations are labeled either blue or red. Blue operations do not require upfront coordination and are lazily replicated to remote sites, thus providing weak consistency. Red operations are totally ordered among them and require coordination between replicas, thus providing strong consistency. For an operation to be labeled blue, it must fulfill two conditions: *i*) the operation must be commutative with respect to all others, and *ii*) the operation must not break application's invariants. To increase the number of blue operations, each operation is decoupled into two parts: *i*) a *side-effect free* generator operation determine the changes induced by the original operation, and *ii*) a shadow operation that effectively modifies the state of the system . While Gemini offers a flexible behavior, the generation of shadow operations is not automated and demands the programmer intervention. Sieve [Li+14], presents an extension to this work, which was designed to address this problem. Sieve automatically labels and produces commutative shadow operations, therefore minimizing the developer intervention. In WeaQL, all operations are commutative by design and are always lazily replicated to remote replicas. To enforce particular application's invariants, instead of coordinating operations that might endanger such invariant, WeaQL resorts to a coordination service prior to the execution of those operations.

Clements et al. [Cle+13] highlight the importance of commutativity in the performance and scalability of a distributed system. They introduced a general rule that states the following: "whenever interface operations commute, they can be implemented in a way that scales". To exploit this property, they developed a tool that analyzes a set of operations based on the interface's specification and identifies under which conditions operations commutes. They used this tool to modify POSIX's calls in a Linux-based kernel and reported that most calls are indeed possible to implement in a commutative way. WeaQL also leverages commutativity to improve the performance of replicated databases. However, contrary to the work of Clements et al., the process of generating commutative operations is automated, consequently not requiring the programmers intervention.

Bayou is a weakly consistent replicated storage system [Ter+95]. It was designed to serve clients with unstable network conditions, as found in early mobile connections. To overcome this limitation, operations only need to contact a single replica. Updates are lazily propagated in the background, thus providing Eventual Consistency. Similar to WeaQL, Bayou automatically resolves conflicts without exposing them to the application level. However, the merge procedures must the provided by the programmer. WeaQL on the other hand, only requires programmers to specify the intended semantic to be able to implement the adequate reconciliation procedures.

ChainReaction [Alm+13] is a distributed key-value store that provides Causal Consistency. Replicas placement follow the same approach as in Dynamo. The subset of replicas responsible for managing a given object is called a chain. A client-side library forwards requests to the appropriate chain. Writes are managed by the head of the chain, which has the responsibility to propagate the updates throughout the rest of the chain; read operations are directed to the tail of the chain. When a write request reaches the tail, it is guaranteed that all the chain have already seen the update. At this point, the tail acknowledges back to the client. To minimize latency, a parameter k defines the number of replicas that must receive the update before returning to the client. While achieving Causal Consistency in read operations with low latency, write operations must wait for the update to be replicated in k replicas, therefore augmenting the perceived latency for those operations. WeaQL does not make the distinction between read and write operations. In particular, all operations can be processed by any replica in the system and complete with low latency. Furthermore, because write operations are not tunneled to a specific replica, WeaQL does not suffer from the traditional issues associated with the Master-Slave approach, such as the master replica becoming a bottleneck.

2.3.4 Summary

In this section, we have explored in detail the “strong vs weak” consistency dichotomy. In particular, we discussed the advantages and disadvantages of both approaches and scrutinized the trade-offs involved when choosing each replication strategy. To summarize, strong consistency guarantees that users observe a consistent view of the system at the cost of becoming unavailable in the presence of partitions. In contrast, weak consistency remains available in such conditions by allowing replicas to temporarily diverge, which may result in users reading stale data. We concluded this section by comparing our work with relevant distributed systems that advocate weak consistency models.

2.4 Conflict-Free Replicated Data Types

In the previous section, we discussed weakly consistent replication, a technique suitable for providing performance and availability. Unfortunately, replicating data in such context is complex because any replica can process updates without synchronizing with other replicas, which may result in conflicts. To address this challenge, so far we have discussed two different approaches, namely automatic reconciliation (e.g., last-writer-wins) and application specific procedures. We now introduce a third approach.

CRDTs [Sha+11b] are a collection of data types where all operations that modify the state are commutative by design. As a consequence, replicas can modify the state of these data types without coordination and still achieve the same final state. Furthermore, conflicts that might arise during the execution of concurrent operations are resolved inside the data type itself, hence relieving programmers from worrying about the concurrency details of their applications. For these reasons, CRDTs implement compelling features to use in the context of weakly consistent systems.

Formally, CRDTs provide Strong Eventual Consistency (SEC), which can be defined as:

Definition 2.5. *Strong Eventual Consistency [Sha+11b]: An object is Strongly Eventually Consistent if it is Eventually Consistent and* ***Strong Convergence:*** *Correct replicas that have delivered the same updates have equivalent state:* $\forall i, j : c_i = c_j \Rightarrow s_i \equiv s_j$

The additional property of SEC (when compared to Eventual Consistency) implies that CRDTs are in a consistent state after delivering the same set of operations. Note that Eventual Consistency does not require this property; indeed, it only states that after delivering the same set of updates, replicas *eventually* converge. The key insight here is that in CRDTs, the reconciliation process (if required) is done as part of the operation itself.

CRDTs expose two types of operations to programmers: query and update operations. The query operation reads the local state of the object while the update operation modifies it. Each update operation is decoupled in two phases: *i*) the first phase is executed locally and may perform some preliminary computations, and *ii*) the second phase, also known as `downstream` phase, disseminates the update asynchronously to all replicas [Sha+11a]. The replica where the client issues the operation is called the *source* replica for that operation.

CRDTs can either be operation-based or state-based. In an operation-based CRDT, the operation is replicated by the source replica and executed at all replicas (i.e., following an active replication approach as described in Section 2.2.3). In a state-based CRDT, the whole state of the object is replicated instead of the operation.

2.4.1 State-based CRDT

Formally, a state-based CRDT (or Convergent Replicated Data Type (CvRDT)) is a tuple (S, s^0, q, u, m), where S represents the state domain of the object and s^0 represents the initial state; q is the set of query operations and u the set of update operations. When an update operation is executed, the object transits from s^i to s^{i+1}

In a CvRDT, an update operation is executed exclusively by the source replica. Asynchronously, the modified object is sent to all other replicas. When a replica receives the remote state of an object, it executes a merge procedure that merges both states (i.e., the remote and local states). After this step, it is guaranteed that the object is consistent across the local and remote replicas.

2.4.2 Operation-based CRDT

An operation-based CRDT (or Commutative Replicated Data Type (CmRDT)), is a tuple (S, s^0, q, t, u, P) where S, S^0 and q preserve the same meaning as described before. In CmRDTs, an update operation is decoupled in two operations: *i*) t is a side-effect free prepare-update operation (phase 1), and *ii*) u is an effect-update operation, that effectively modifies the state of the object (phase 2). The prepare-update operation executes exclusively at the source replica and it does not modify the internal state of the CRDT. This phase is used to generate unique tags or calculate timestamps needed to enable automatic reconciliation (let us call this the operation's metadata). At the source replica, the effect-update operation is executed right after phase 1. Asynchronously, this operation is disseminated to the remote replicas along with the metadata, and is executed independently by each one.

2.4.3 CRDTs Examples

While there are more CRDTs defined in the literature (such as graphs, counters, and replicated arrays [Sha+11a]), this section will describe the data types that are most relevant to our work, namely registers (which can be seen as records in a database table) and sets (which represent database tables). All the specifications presented in the remainder of this section were originally formalized by Shapiro et al. in [Sha+11a].

LWW Register

A register is a memory cell that supports both assign (i.e., update) and value (i.e., query) operations. Each assign operation is associated with a timestamp. Non-concurrent assign operations preserve sequential semantics (i.e., the later one overwrites the earlier one), which means that assign operations do not commute. In a state-based `LWW Register`, the assign operation always succeeds on the local replica, since there is no greater timestamp than the one just calculated. The new state is eventually replicated to all replicas and merged with the remote state, by comparing the two timestamps associated with each state (i.e., by comparing the timestamp generated in the local replica with the one generated by the remote replica). Specification 1 presents the pseudo-code for this CRDT.

In an operation-based `LWW Register` (see specification 2), the assign operation is split into two operations as discussed before. During phase 1, a new timestamp is calculated; phase 2 modifies the value of the register, if and only if the new timestamp is greater than the current timestamp associated with the register.

Specification 1 LWW Register (state-based)

```
1:  payload X $x$, timestamp $t$                          ▷ $X$ : some type
2:      initial $\perp$,0
3:  update assign(X w)
4:      $x, t = w, now()$                                 ▷ Timestamp, consistent with causality
5:  query value() : X $w$
6:      let $w = x$
7:  compare (R, R') : boolean $b$
8:      let $b = (R.t \leq R'.t)$
9:  merge (R, R') : payload $R''$
10:     if $R.t \leq R'.t$
11:         $R''.x, R''.t = R'.x, R'.t$
12:     else
13:         $R''.x, R''.t = R.x, R.t$
```

Specification 2 LWW Register (operation-based)

```
1:  payload X $x$, timestamp $t$                          ▷ $X$ : some type
2:      initial $\perp$,0
3:  query value() : X $w$
4:      let $w = x$
5:  update assign(X x')
6:      atSource() $t'$
7:          let $t' = now()$                              ▷ Timestamp
8:      downstream($x'$, $t'$)                            ▷ no precondition: delivery order is empty
9:          if $t < t'$
10:             $x, t := x', t'$
```

G-Set, 2P-Set, and OR-Set

Sets are a building block for implementing many data structures such as Maps, Containers and Graphs. A traditional implementation of a Set includes three basic operations: add, remove and lookup.

A G-Set (i.e., Grow-Only Set) only supports the add and lookup operations. Since the only operation that modifies the state is the add operation (which is commutative by design), an operation-based G-Set naturally commutes. In a state-based approach (depicted in Algorithm 3), the add operation locally adds an element to the set, and the merge operation computes the union between the local and remote state.

In the 2P-Set (presented in Algorithm 4), elements may be added and removed, but never added again. The implementation is based on two G-Sets, one for added elements (S_a) and the other for removed elements (S_r). An element e is said to be in the Set if $e \in S_a \wedge e \notin S_r$. A remove operation is allowed if and only if the source replica contains the element to be removed. Hence, a remove operation must first query the set to check if the element e is present. This behavior can be captured by requiring a precondition $P : lookup(e)$.

While useful in some cases, the previous specifications do not reflect the expected

Specification 3 G-Set (state-based)

```
1:  payload set $A$
2:      initial $\emptyset$
3:  update add (element e)
4:      $A := A \cup \{e\}$
5:  query lookup (element e) : boolean $b$
6:      let $b = (e \in A)$
7:  compare (S, T) : boolean $b$
8:      let $b = (S.A \subseteq T.A)$
9:  merge (S, T) : payload $U$
10:     let $U.A = S.A \cup T.A$
```

Specification 4 2P-Set (state-based)

```
1:  payload set $A$, set $R$                          ▷ $A$: added; $R$: removed
2:      initial $\emptyset, \emptyset$
3:  query lookup (element e) : boolean $b$
4:      let $b = (e \in A \wedge e \notin R)$
5:  update add (element e)
6:      $A := A \cup \{e\}$
7:  update remove (element e)
8:      pre lookup(e)                                  ▷ precondition
9:      $R := R \cup \{e\}$
10: compare (S, T) : boolean $b$
11:     let $b = (S.A \subseteq T.A \vee S.R \subseteq T.R)$
12: merge (S, T) : payload $U$
13:     let $U.A = S.A \cup T.A$
14:     let $U.R = S.R \cup T.R$
```

behavior of a Set. The G-Set does not allow removes while the 2P-Set does not allow elements to be added again after being removed. The OR-Set more closely resembles a general implementation of a Set. A specification for an operation-based OR-Set can be found in Algorithm 5. In this variant, each element has a unique tag associated. An add operation generates a unique tag at the source replica, which inserts the pair *(element, tag)* in the set. Then, it propagates this pair to the remote replicas. Since two concurrent add operations generate distinct (and unique) tags, duplicates may be present. Hence, the lookup operation must mask duplicates. When removing an element, the set of unique tags associated with that element observed at the source replica are recorded. After a remove operation, the element is no longer in the set and the collection of tags observed during the operation is propagated to other replicas, which in their turn will remove all elements that match the tags received. In this CRDT, the add operation takes precedence over the remove operation in case both operations are executed concurrently. This is because the set of tags observed by the remove operation does not include the tag generated by the concurrent add operation.

Specification 5 OR-Set (operation-based)

```
payload set $S$                                        ▷ set of pairs $\{(e, unique-tag\ \mu, \ldots)\}$
  initial $\emptyset$
query $lookup$(element e) : boolean $b$
  let $b = (\exists_\mu : (e,\mu) \in S)$
update $add$(element e)
  atSource (e)
    let $\alpha = unique()$                            ▷ unique(): returns a unique value
  downstream $(e,\alpha)$
    $S := S \cup \{(e,\alpha)\}$
update $remove$(element e)
  atSource (e)
    pre $lookup$(e)
    let $R = \{(e,\mu) | \exists_\mu : (e,\mu) \in S\}$
  downstream $(R)$
    pre $\forall (e,\mu) \in R : add(e,\mu)$ has been delivered     ▷ Causal order suffices
    $S := S \setminus R$                               ▷ Remove pairs observed at source
```

2.4.4 Summary

In this section, we discussed in detail a relevant approach to aid the reconciliation process of conflicting operations. CRDTs transparently reconcile conflicting updates by implementing the reconciliation procedures inside the data type while guaranteeing that their state converge in all replicas, despite the order of which they execute updates. As weakly consistent replication has been gradually gained popularity as a way to improve the performance of many online systems, CRDTs have had a significant influence in their design. For instance, CRDTs are nowadays deployed in many industry systems, such as Riak [Tec15] and the online game League of Legends [Ava16], among others.

2.5 Invariants Preservation

Frequently, application's developers desire to enforce a set of properties on their data to implement particular behaviors on their applications. For instance, Twitter allows users (i.e., clients) to login in its website by providing a combination of *username+password*. As a consequence, Twitter must be able to uniquely identify each client to be capable of displaying the appropriate information associated with each client, such as private messages, news feed, etc. To achieve this, Twitter requires usernames to be unique, so whenever a client tries to login in the site, Twitter promptly knows who is the user trying to access its system. Unique usernames is a paradigmatic example of an invariant that one might want to enforce on its data. However, there are more examples of these invariants, such as restricting the values of an object to a certain value, or set of values (e.g., ensuring that the stock of an item in an shopping application is always positive), or maintaining relations between different objects (e.g., foreign-key constraints in the context of relational databases).

Defining invariants over the data objects stored in a database allow programmers to better express the envisioned semantics of their applications, and prevent erratics behaviors that might disrupt the normal utilization of a system. Typically, these invariants are encoded in the database level by resorting to integrity constraints provided by SQL-based relational databases, such as *foreign-key* constraints, *check* constraints and *unique* constraints. Alternatively, they can also be enforced at the application-level if the database does not provide these mechanisms, as in the case of NoSQL databases, making the application's logic notoriously more complex.

Preserving invariants in distributed environments remains challenging [Bal+15b], as the data objects which must obey these rules are no longer centralized, but instead replicated across a number of sites. In this context, as replicas are able to process concurrent updates, they might unwillingly lead to the violations of invariants. Returning to the Twitter example previously describe, how can Twitter ensure that usernames are globally unique if multiple replicas can process the operation of a user signing-up in their system? Unfortunately, in this case one has to either provide strong consistency semantics by centralizing updates on a single replica, or resort to expensive consensus protocols, such as the Paxos algorithm [Lam06]. However, the emergence of weak consistent semantics as a way to improve the performance of distributed systems, has increased the tension between enforcing global invariants and improved performance.

To address these challenges, researchers have been exploring alternative approaches to preserve global invariants while minimizing the coordination costs. Gemini [Li+12] provides weak consistent semantics whenever possible, but requires global synchronization on the operations that might violate global invariants. SIEVE [Li+14] resorts to static analysis to determine the weakest pre-conditions in which operations may be executed without resorting to coordination. Balegas et al. on the other hand, try to minimize the coordination costs by placing it outside the critical path of the operations [Bal+15a]. Bailis et al. focus on the integrity constraints present in relational databases and propose I-Confluence [Bai+14], a formal framework which determines when coordination is required to ensure the correctness of an application (i.e., to avoid the violation of invariants)

The work in this dissertation confluences in many of the previous observations. In more detail, in designing WeaQL, we also had to address this well-known tension between preserving the SQL semantics, which requires coordination among replicas, and providing weak consistency, whose performance gains come from avoiding coordination. To tackle this challenge, in this dissertation we leverage many insights and observations made previously on this topic [Bai+14; Bal+15a; Li+14], and introduce a set of lightweight mechanisms to preserve SQL integrity constraints, such as foreign-key constraints, unique constraints, and auto-increment values. These mechanisms provide a sweet spot between preserving the most important invariants of relational databases and minimizing the coordination overhead.

2.6 Summary

In this chapter, we have reviewed the topics and works found in the literature that are more relevant to the contributions made in this dissertation. In Section 2.2 we have discussed how today's applications leverage replication in order to improve their performance and availability. In this context, we have discussed how replicas coordinate their efforts, and how updates are disseminated across a set of replicated databases. In more detail, we explored synchronous and asynchronous techniques, and argued about the advantages and disadvantages of both approaches. In Section 2.3, we have examined the trade-offs that distributed systems must contemplate when advocating a consistency model. In particular, strong consistent systems are easy to reason about, but sacrifice performance and availability in the presence of network partitions. In contrast, weakly consistent systems favor availability and performance, but require an extra effort by developers to cope with conflicts and inconsistencies that might arise in this environment. In this line of work, we discussed CRDTs in Section 2.4, a relatively recent family of data types designed to seemingly handle conflicting operations, therefore making them a choice of excellence in the design and implementation of weakly consistent systems. We ended this chapter in Section 2.5, by discussing the consequences of relaxing the consistency model in relation to the preservation of global invariants in a database system. In particular, we described the inherent tension between avoiding coordination to improve performance, and enforce global invariants which by their nature may require global coordination.

Chapter 3

Solution

In this chapter, we present our solution, WeaQL, a distributed middleware layer. In Section 3.1 we start by giving an high-level overview of the solution proposed in this dissertation, followed by an outline of the goals which we aim to accomplish. In Section 3.2 we discuss the semantics that WeaQL offers to applications when compared to the traditional ACID semantics. In Section 3.3 we describe the set of SQL invariants which are relevant for the context of this dissertation. The remaining sections of this chapter present our solution in full detail.

3.1 Goals and Overview

In a nutshell, the main goal of WeaQL is to provide applications that store their state in a relational database with a more scalable alternative, improving their performance. To this end, we resort to replication of the database state and explore a trade-off that has been commonly employed in the context of storage systems for geo-replicated scenarios. In particular, we sacrifice strong consistency, instead offering a form of eventual consistency which enables us to avoid the overhead of continuous synchronization among replicas. As a result, we allow transactions that access the same data not to observe the effects of each other and subsequently reconcile their possibly conflicting side-effects. However, we would also like our system not to deviate significantly from the currently offered interface and semantics. In particular, this entails the following concrete objectives:

- to offer a language and semantics that are very close to SQL, namely preserving its main invariants, while requiring minimal coordination among a set of replicas of a standard SQL database;
- to offer multiple conflict resolution policies to developers, that should entail minimal (or none) changes to the application code;

- to transparently guarantee that all replicas converge to the same state, regardless of the order in which they execute transactions or queries issued by applications.

To accomplish these goals, WeaQL employs a set of techniques which will be described thoroughly in this chapter.

To summarize, WeaQL is a module deployed between the application and the database layer. WeaQL has a client-side component that resides on the application-side, and a component that is coupled with each instance of a replicated database.

At a high-level, an interaction of a client application with WeaQL has the following pattern. Clients issue a series of SQL queries, which are intercepted by the client-side component of WeaQL, which provides the front-end for our replication strategy and storage back-end. To ensure convergence, SQL queries that form the transaction are transformed in a way that converts them into CRDT operations that commute by design. These transformations are transparent for the application and allow the execution of most transactions without resorting to any form of coordination. The client will then commit the transformed transaction to a single replica of the system. When an application issues a commit, WeaQL first verifies if the transaction can be safely committed locally without breaking SQL invariants. While most transactions can be committed locally, some of them might lead to invariant violations when applied on remote replicas. In those cases, WeaQL resorts to a logically centralized but physically replicated coordination service prior to committing the operation. The coordination service is responsible for providing the fundamental coordination mechanisms for preserving SQL constraints which are not possible to maintain without some form of coordination. For instance, it checks if a given value can be used as a unique-key, or if it is already being used by another record in that table. When a transaction is committed in the local replica, WeaQL replicates it to other instances of the database asynchronously, following a lazy replication approach. This technique slightly departs from the traditional ACID semantics, which will be discussed in the next section.

3.2 WeaQL Semantics

The ACID properties (**A**tomicity, **C**onsistency, **I**solation, **D**urability) are a fundamental aspect of relational databases, in which developers heavily rely on to build their applications. Therefore, it is important to understand how these four principles are integrated into our solution.

The end result of our design is a system whose specification can be described as providing the Atomicity, Consistency, and Durability aspects of the ACID semantics, but where the Isolation guarantee is slightly weakened in the following way. The low-level aspect of isolation of the underlying database engine is preserved, meaning that the intermediate effects of ongoing transactions may or may not be observed by one another, depending on the isolation level defined by the programmer for the execution of those transactions. However, the high-level aspect of making transactions becoming visible to other transactions is weakened. This is because WeaQL defers updates propagation to remote replicas, which enables replicas to temporarily diverge. In this context, immediately after a transaction is committed, its effects are only seen by the replica in which the transaction was executed (i.e. the local replica). Therefore, clients accessing other replicas will not observe the effects of that transaction until the local replica propagates its effects. This also implies that two transactions that modify the same data may execute against the same database state, not observing each other's effects, leading to the need to reconcile such divergent state at a later time.

This approach raises the possibility of violating important invariants of the SQL language, which for completeness we summarize next.

3.3 SQL Invariants

The SQL language allows programmers to specify integrity constraints on individual records, a collection of records (tables), or relations among records in different tables. This mechanism eases the development of applications and is important for the continuous use of relational database nowadays. In this work, we focus on three of the most commonly used integrity constraints, which are characterized in the remainder of this section.

Unique Constraints

Unique constraints disallow the repetition of values in a column, or set of columns, across multiple records of a database table, making each row on a table distinguishable. For example, a social network developer can flag a field *username* as unique so that *username* can be used to login in the service. *Primary-keys* and *auto-increment* constraints are particular examples of this class of constraints.

Foreign-Key Constraints

Foreign-key constraints denote a relation between records of two different database tables. It is specified over one, or a set of columns in one table (which we dub *child-table*), whose values (if different from `NULL`) have to match the values on a similar number of columns in a second table (which we name *parent-table*). There is an implicit dependency expressed by this constraint: a *child-record* (i.e., a record in a child-table) cannot be inserted if the respective *parent-record* (i.e., a record in a parent-table) does not exist. Foreign-key constraints may also be defined with a set of options that specify the action to be taken by the database when parent-records are updated or deleted. The `RESTRICT` option prevents a parent-record from being deleted or updated if it has at least one child-record referencing it. The option `CASCADE` will propagate the actions made to columns belonging to the foreign-key constraint in the parent-record to all child-records. While there are other options (e.g., `SET NULL`), in this work we focused only on the two options previously mentioned because they were the most challenging to reason about in a distributed environment. Dealing with the `SET NULL` option is future work.

Foreign-key constraints are useful for tying together records across different tables. For instance, in a shopping cart application, there can be two tables stored in the database, namely *cart* and *cart_line*. The former table keeps a set of records, each of which describes the information regarding a cart, while the latter table stores a list of items associated with each cart. In this case, one could specify a foreign-key constraint between the two tables so that items can be referred back to the cart they belong to. In this example, *cart* and *cart_line* are the parent and child table, respectively.

Check Constraints

Check constraints allow developers to restrict the values for a specific column (across all records) on a database table. A paradigmatic example of the use of this constraint is to capture the fact that the stock of a product must be non-negative (i.e., $stock \geq 0$). This class of integrity constraints may also be seen as bounded counters (when applied to integer columns), which must be below or above (or both) a given set of threshold value (or values).

3.4 Merging Policies

As discussed in Section 3.1, WeaQL propagates updates asynchronously between replicas. This design choice allows the execution of concurrent operations which may conflict with each other and must be later reconciled. In this context, it is important to understand the desired behavior for handling conflicting operations. What should be the final state of the database when a replica deletes a record and, concurrently, other replica updates it? Should the final state reflect the modification of that record or should it simply reflect the delete operation? It turns out that there is not a single correct answer to this question. Different answers may be suitable for different applications scenarios. This raises the question of how should the reconciliation of conflicting operations be done. On the one hand, implementing a static reconciliation method (such as last-writer-wins [LM10]) compromises the flexibility required by many applications for handling concurrent operations. On the other hand, imposing on developers the burden of implementing their own reconciliation procedures on a per application basis leads to errors as developers have to make a non-trivial effort to re-design their code and to reason about all possible concurrent executions between all operations.

To address the above tension, we aim to find a sweet spot in this trade-off: we seek to minimize the developer intervention while at the same time provide flexibility, enabling the use of different semantics for addressing conflicting updates.

```
@DELETEWINS CREATE TABLE FOO (
@LWWINTEGER a int(10) unsigned NOT NULL,
@NOSEMANTIC b int(10) unsigned,
@LWWINTEGER c int(10) unsigned,
@DELTAINTEGER d int(10) unsigned,
@LWWINTEGER e varchar(50),
PRIMARY KEY (a),
@UPDATEWINS FOREIGN KEY (b)
REFERENCES t1(b))
ENGINE=INNODB;
```

Figure 3.1: SQL Schema Annotations Example

To this end, we require programmers to specify the intended semantic for dealing with concurrent operations that modify the state of the database using small annotations in the database schema file (DDL); these annotations are processed by WeaQL, which in its turn, will apply the adequate conflict resolution strategy (i.e. merging policies), freeing the programmer from that burden, while still providing several choices for the intended semantics.

To capture the different merging policies available to the application programmer, we have defined three different types of annotations, namely *i*) table annotations, *ii*) attributes annotations, and *iii*) foreign-key annotations.

In Figure 3.1 we illustrate how these annotations are integrated into the DDL schema file. All annotations are optional, which means that in case the programmer decides not to specify any particular semantic, WeaQL will employ what we consider the "default" behavior to address potential conflicts between transactions modifying the database state.

In the remainder of this section, we will focus exclusively on covering each type of annotation and discussing their underlying semantics. In Section 3.7 we will resort to these annotations to design the conflict resolution procedures employed by WeaQL.

3.4.1 Table Annotations

We say an update-delete conflict arises when two transactions concurrently update and delete the same record on a table.

Table annotations allow developers to specify how such conflicts should be addressed. The annotation `@UPDATEWINS` favors the update operations. This mean that when two operations respectively update and delete the same record concurrently, the update operation overrides the delete operation and, therefore, the final state of the database only reflects the update operation. In contrast, when the database table is annotated with the `@DELETEWINS` annotation, the opposite behavior is achieved. Accordingly, in this case, the final state of the database will not contain the record.

3.4.2 Attribute Annotations

This type of annotations enables WeaQL to deal with conflicts at the level of columns, in particular for transactions that update the same column in a table record or transactions that attempt to insert the same record (i.e. with the same primary-key).

To handle conflicts which arise from two concurrent updates on the same record, we follow a strategy similar to the one employed in the SIEVE system [Li+14], which offers two merging policies. On the one hand, with the annotation `@LWW*`, WeaQL follows a last-writer-wins approach, which means that only the last value applied to that column will be stored in the database. On the other hand, by using the annotation `@DELTA*`, the updates for that column are perceived by the application as an increment or decrement of a value over an original value, in which case the final state of the column will reflect the effects of all operations. As an example, this semantic might be useful for an attribute that represents the stock of a particular item in an online store, such as Amazon. With the annotation `@DELTA*`, if transactions t_1 and t_2 concurrently try to increment the stock of a given item by 3 and 5 respectively, the total delta would be the sum of both values, in this case, 8. These merging policies differ in a fundamental way. With the `@DELTA*` annotation, the effects of both updates are reflected in the state of the database, even if they were issued concurrently. In contrast, with the `@LWW*` annotation, only one of the updates is reflected in the final state of the database.

WeaQL also deals with a different type of conflict, namely those that arise upon the concurrent insertions of records with similar values on columns with a unique constraint. This is relevant as unique fields in a table may have different semantics for the application. In particular, some of these columns might only serve to identify records in tables, where the concrete value has no meaning to the application, while in other cases (for instance an username) the value itself might be used by the application for other purposes. This implies that in the first case WeaQL can address these conflicts by transforming (locally) these values in a way that ensure their uniqueness while in the later case such a technique cannot be employed and WeaQL might be required to resort to some form of coordination. To distinguish between these two cases we allow individual columns to be tagged with the `@SEMANTIC` and `@NOSEMANTIC` annotations, which indicates whether or not the values in that column are relevant to the application. Additionally, if the annotation `@SEMANTIC` is used along an auto-increment field, we consider that *ids* must be generated sequentially.

The relevance of this distinction is further discerned ahead in Section 3.6.1.

3.4.3 Foreign-Key Annotations

These annotations are similar to the table annotations described previously. However, while table annotations specify the conflict resolution policy for operations on a given table, foreign-key annotations specify the policy for dealing with concurrent delete and insert operations across two tables that have a foreign-key restriction between them.

WeaQL Annotations		
Type	**Options**	
Table Annotations	**@UPDATEWINS**	
	@DELETEWINS	
Foreign-key Annotations	**@UPDATEWINS**	
	@DELETEWINS	
	WeaQL Annotations	**SIEVE Annotations**
Attribute Annotations	**@SEMANTIC**	**@LWW***
	@NOSEMANTIC	@DELTA*

Table 3.1: WeaQL Annotations

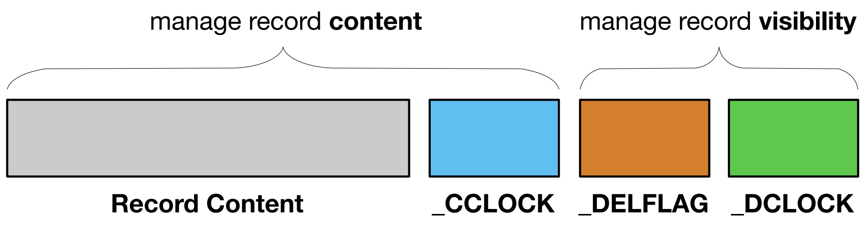


Figure 3.2: A database record in WeaQL

With the annotation `@UPDATEWINS`, if a parent-record is deleted concurrently with a child-record insertion, the insert operation prevails and WeaQL will avoid the deletion of the parent-record and effectively insert the new child-record. The `@DELETEWINS` annotations specify the opposite behavior, meaning that the child-record becomes deleted (similar to the case where the insertion of the child-record is serialized before the delete of the parent-record).

3.4.4 Summary

In this section, we have detailed the different merging policies in which programmers can rely on to reconcile conflicting operations in their applications. Table 3.1 summarizes all available annotations in WeaQL, and we highlighted the default behavior for the cases where no annotation is provided.

In the next section, we will cover a fundamental aspect that every replicated database must pursue which is state convergence.

3.5 State Convergence

The key mechanism which WeaQL uses to achieve state convergence is commutativity. WeaQL leverages commutativity by ensuring that all transactions committed to the database are commutative by design. In particular, WeaQL ensures this property by applying a set of transformation over the SQL statements issued by the application in a way that makes each transaction globally commute with all others.

Because each transformed SQL transaction is commutative, replicas can execute operations in different orders and still reach the same final state. This avoids the necessity of determining a global order of operations and allows WeaQL to propagate transactions to remote replicas asynchronously. Unfortunately, this approach also promotes conflicts, since two operations may concurrently update the same record. Hence, in order to ensure a consistent state among replicas, it is imperative for WeaQL to be able to detect these conflicts and rollback certain operations. Note that here we do not use the word rollback in the traditional sense that one usually associates with the operation of a relational database. We do not *actually* rollback transactions by issuing a rollback command to the database. Instead, its *effects* are undone through the effects of transactions whose SQL statements are transformed.

To achieve this, we enrich the database schema with metadata. The metadata consists of three extra columns for every table, depicted in figure 3.2. The **_cclock** and the **_dclock** fields are vector clocks that keep track of the logical time when each record was modified or deleted, respectively. The **_delflag** field is a boolean flag that indicates whether the record is deleted or not.

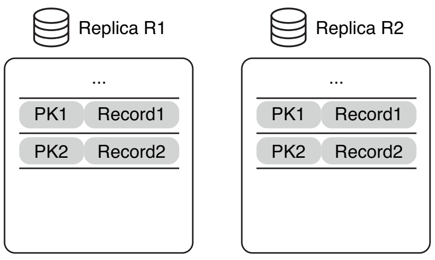
These extra columns are useful in the following way. Imagine two replicas R_1 and R_2 concurrently updating a record in a specific column tagged with the annotation `@LWW*`. In this scenario, because we follow a last-writer-wins strategy, one of the updates must be discarded. Hence, we use the value of **_cclock** to decide which update should be discarded. In particular, we compare both clocks and decide in favor of the update with the highest clock value.

An important aspect of this technique is that it is completely transparent to the application. What this means is that the application have no knowledge about the extra columns and it does not need to change the queries it uses to retrieve and store data. WeaQL automatically adds the metadata columns in an automatic preprocessing of the database schema.

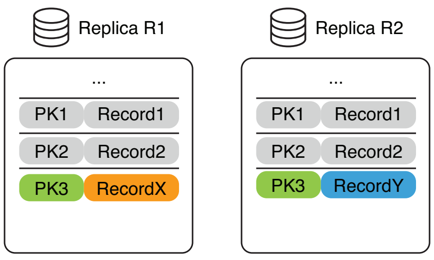
To summarize, in WeaQL every interaction with the database (i.e. transaction) is commutative. We do not propose any novel technique. Instead, we leverage mechanisms that have been widely used by the NoSQL community and apply the same insights to the context of relational databases, in particular, commutativity and weakly consistent replication. Unfortunately, this approach endangers the properties of SQL invariants, a topic which will be discussed in the next section.

3.6 Preserving SQL Invariants

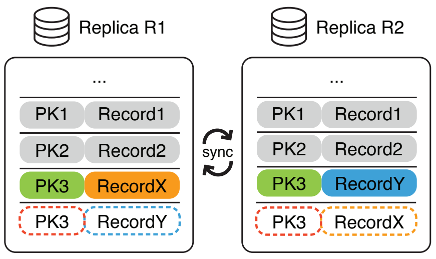
To preserve SQL invariants, traditional solutions resort to some form of coordination mechanisms. Multi-Master architectures rely on the virtues that group communication



(a) Initial state



(b) Concurrent insertion



(c) Error when replicating operation

Figure 3.3: State divergence on concurrent insertion of unique-keys

primitives provide to determine a total order of operations. On the other hand, Master-Slave solutions serialize all updates using a master replica.

As a result of employing weakly consistent replication in WeaQL, SQL invariants must be carefully handled so its original semantics can be preserved. In our work, we focus on three of the most commonly adopted integrity constraints, namely the ones discussed in section 3.3. WeaQL specifically targets this set of constraints as they are widely used across many applications deployed on top of relational databases.

In the remainder of this section, we discuss each of these SQL invariants in more detail. We start by observing how a naive coordination-free execution of operations might violate the semantics underlying each integrity constraint. We then continue by presenting how WeaQL effectively protects SQL integrity constraints while minimizing coordination between replicas.

3.6.1 Unique Constraints

When two operations try to insert two tuples with the same value in the unique-key, only one of the operations should succeed. Figure 3.3 shows a potential situation that might surface when two concurrent insertions on the same table are executed in different replicas without coordination. Figure 3.3a reflects the initial states of both replicas, which are in a consistent state. In figure 3.3b both replicas independently, but more important, concurrently without resorting to any form of coordination, insert a tuple with the same

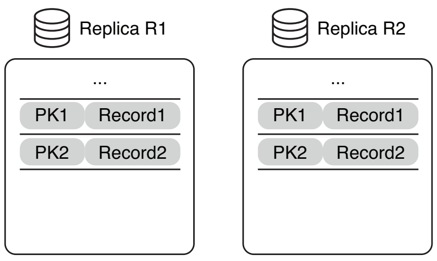
Ids Generation			
	Replica 1	**Replica 2**	**Replica 3**
Ids	7	8	9
	10	11	12
	13	14	15
	16	17	18

Table 3.2: *Ids* generation in WeaQL

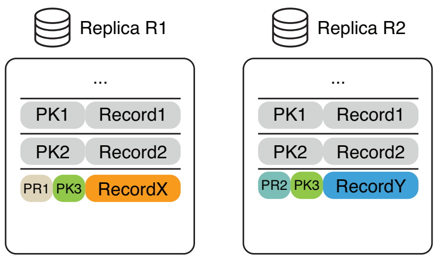
unique-key, but different values for the remaining columns. Because the operations are executed without coordination in each replica, each of the operations is initially successful. The problem arises when the operations are propagated, in the background, to the remote replicas, as portrayed in Figure 3.3c. Both replicas will fail to execute the operation received from the remote replica because of the unique constraint, leading to a divergent state (which will become permanent). Clearly, if the operations did not use the same value for the unique-key this problem would never arise. Therefore, the question that one should ask is the following: does the unique value has a relevant semantic for the application? If the unique value has no semantic meaning for the application it means that its sole purpose is to uniquely identify each record. In such cases, we consider that it is acceptable to manipulate the value of each unique-key, as long as we preserve its original semantics (i.e. that the values remain unique). Identifier columns (*ids*) are a common example where the values themselves have no meaning besides identifying each record. In this case, it is possible to generate a globally unique *id* without requiring replicas to coordinate. Table 3.2 describes how WeaQL handles such cases. Initially, replicas start by reading the current maximum value for this unique-key. Subsequently, each replica adds its own *id* to the maximum value. For instance, if the maximum value was 6, replica R_1 would get 7 and replica R_2 8. To generate globally unique numbers, each replica keeps increasing its value by a constant delta equal to the number of replicas in the system.

In the case where the unique constraint refers to `string` fields, we can transform the value by appending a prefix with the *id* of the replica where the operation was issued. We illustrate this behavior in Figure 3.4. Again, we start in a consistent state in Figure 3.4a. In Figure 3.4b, two concurrent inserts with the same value for the unique-key are executed. but in this case, before inserting the tuple in the database, WeaQL appends the replica's prefix to the unique value. When the operations are replicated, both will execute successfully and the state of both replicas will converge as shown in Figure 3.4c.

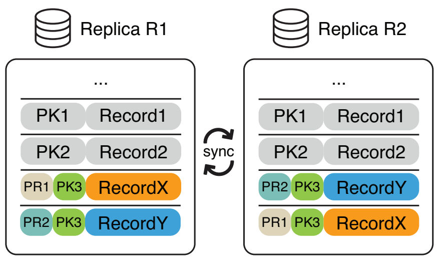
There are however cases where the value of the unique-keys has a semantic meaning. A database table that contains the usernames of all registered users in the system is a particularly relevant example. In this case, we cannot employ the strategy described above as it would change the behavior of the application in a way that would be noticeable by its



(a) Initial state



(b) Concurrent insertion



(c) State convergence on concurrent insertion of unique-keys

Figure 3.4: Concurrent insertion of unique-keys in WeaQL

users. In particular, if a user registers himself in the system with a specific username, his expectation is to use that same username to execute particular operations such as login in the system. Appending a prefix in such a case would not allow the application to retain its originally intended behavior. Therefore, to preserve the integrity constraint in this situation one should either not allow concurrent operations or, alternatively, coordinate such operations. WeaQL follows the latter approach. However, instead of coordinating the whole operation across all replicas in the system, WeaQL issues a lightweight Remote Procedure Call (RPC) to the coordination service asking for permission to use the specific unique value. We have already described how WeaQL distinguishes between both cases in Section 3.4.2, namely by tagging the database field with the appropriate annotation.

In Figure 3.5, we present a diagram which summarizes how WeaQL handles this class of constraints. Green and red boxes represent coordination-free and coordinated techniques, respectively.

3.6.2 Foreign-Key Constraints

Preserving a *parent-child* relation without coordination is challenging. First, the cascade behavior described in Section 3.3 depends on the state of the replica where the operation is executed, which means that the operation has the potential to be non-deterministic

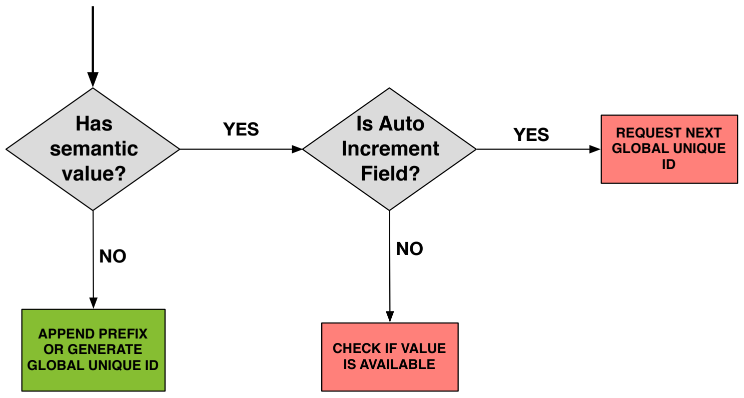

Has semantic value?
YES
Is Auto Increment Field?
YES
REQUEST NEXT GLOBAL UNIQUE ID
NO
APPEND PREFIX OR GENERATE GLOBAL UNIQUE ID
NO
CHECK IF VALUE IS AVAILABLE

Figure 3.5: How WeaQL manages Unique constraints

(and non-determinism is a well-known source for state divergence [Cle+13; Li+12; Li+14; Sha+11b]). Second, two replicas may concurrently manipulate the two tables in a foreign-key relationship and originate inconsistencies, for instance when one replica deletes a parent-record and other inserts a new child-record.

To tackle these challenges, we opted to incorporate the implementation of the foreign-key constraint directly in WeaQL. To do so, we removed those commands from the database schema (in the automatic preprocessing phase of the database schema) and instead, implemented the corresponding behavior of the foreign-key constraint in WeaQL. By doing so, we are able to capture the side-effects of the operations and apply them in a deterministic, state-independent way across all replicas. As an example, whenever necessary, WeaQL piggybacks parent-records information along with operations creating or updating child-records to ensure that the parent-child relation is not broken. In particular, if one annotates the database schema with the tag `@UPDATEWINS` for a given foreign-key constraint, whenever a child-record is inserted, we piggyback the primary-key of its parent-record. This enables WeaQL to bring the adequate parent-record back into the database if a concurrent operation had deleted it.

In the end, managing this class of constraint can be achieved without requiring coordination in any of the possible concurrent scenarios. Later on, in Section 3.7, we describe in more detail how WeaQL effectively enforces and protects this class of constraint.

3.6.3 Check Constraints

As discussed in Section 3.3, a check constraint may also be seen as a bounded counter above or below a certain threshold. Implementing bounded counters without coordination can be partially achieved [Bal+15b]. Consider the constraint `CHECK stock` $\geq$ `0`. In this case, operations that increment the stock field will never break the invariant. In contrast, two concurrent operations trying to decrease the stock value might lead the stock value below zero, therefore violating the constraint. WeaQL makes this important distinction and only employs coordination in the latter case. The coordination service is

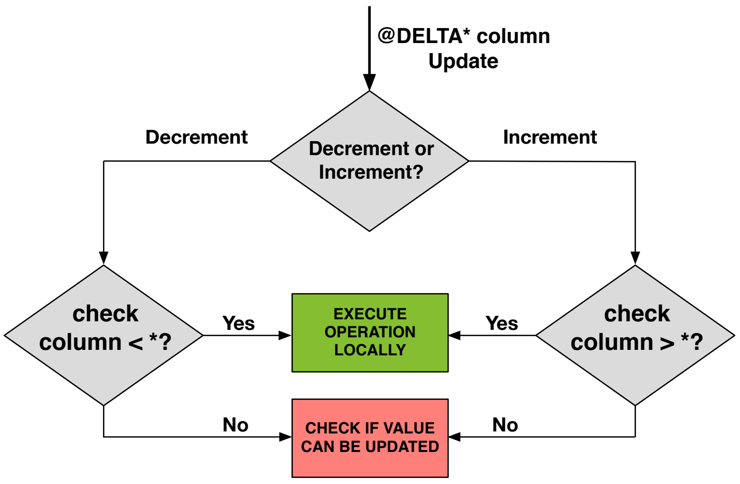


Figure 3.6: How WeaQL manages Check constraints

contacted to request permission to decrement (or increment if the constraint is defined with ≤) a specified value where such a constraint exist.

Note that the above discussion is only valid if the columns covered by a check constraint are tagged with `@DELTA*` annotation. If the annotation is `@LWW*`, it is trivial to evaluate the value and decide whether or not it complies the check constraint.

Figure 3.6 summarizes the behavior of WeaQL on preserving this class of constraints, when a table column is covered with a check constraint.

3.6.4 Coordination Service

While in most cases it is possible to protect invariants without resorting to any form of coordination, we have identified particular cases where coordination must be employed to preserve some SQL invariants. To this end, WeaQL resorts to a replicated coordination entity prior to committing transactions that might endanger such SQL invariants.

To summarize, with WeaQL coordination is required for preserving invariants in three different scenarios: *i*) generating sequential *ids* for auto-increment fields, *ii*) inserting unique-keys with semantic value, and *iii*) decrementing or incrementing a table field covered by a check constraint of the type ≥ or ≤, respectively.

The coordinator is an external service which is contacted by WeaQL prior to committing a transaction, which by its nature requires coordination. This step is a very lightweight process and requires a single RPC to complete. At the end, if the transaction is allowed to commit, it is guaranteed that all the invariants involved in the transaction will be preserved. In contrast, if the transaction is not allowed to commit the client application is notified of the error.

This coordination service must keep track of some information to make the decision about whether or not a transaction can be safely committed. In particular, the coordinator keeps track of the following information. For every unique constraint associated with a column with semantic value, the coordinator keeps tracks of all currently used values. Therefore, when an operation tries to insert a record with a particular value for that column, all that the coordinator must do is to check if that value is already being used. If not, the new value is added to the list of currently used values and the transaction is allowed to proceed. To generate sequential *ids* for an auto-increment constraint, the coordinator manages an integer value which is atomically incremented. The coordinator maintains different values for different auto-increment constraints, hence they can be independently incremented. As to ensure check constraints, the coordinator does the following. First, it keeps track of the threshold associated with the constraint. For instance, if the check constraint is defined as `check stock > 0` the threshold value would be 0. Then, for every record which must comply this constraint, the coordinator monitors the current value for that particular column. As an example, if a transaction tries to decrement the stock of a given record by 3, the coordinator checks the current value for that record to decide if the operation can proceed. In this case, the operation could only proceed if the current value was ≥ 4, in which case it would be updated to the new value.

When the requirements of a transaction cannot be satisfied, the coordinator forces WeaQL to abort the transaction.

3.6.5 Summary

In this section, we have discussed how WeaQL is able to preserve the natural semantics of SQL invariants with minimal or no coordination. Instead of looking at each invariant as a blind rule to which the database records must comply, we extract meaningful information regarding each invariant through the use of annotations. This enables us to gain a deeper understanding of each invariant and, in some cases, we were able to explore alternatives solutions that guarantee the enforcement of invariants while avoiding coordination.

In the following section, we will scrutinize the SQL transformations used by WeaQL, an aspect that has been referred to since the beginning of this chapter but one which has not yet been covered in detail up to now.

3.7 SQL Queries Transformations

SQL transformations are a transversal mechanism in WeaQL. First, they effectively implement the foreign-key constraint semantics. Second, they provide state convergence by leveraging the metadata columns discussed in Section 3.5. Third, they are also relevant to implement the semantics associated with the annotations described in Section 3.4. For instance, if the programmer specifies a `@UPDATEWINS` policy over a database table, when two replicas concurrently delete and update the same record, the transformations employed by WeaQL must ensure the appropriate behavior, in this case by guaranteeing that after both operations are executed, the record is still in the database with the appropriate final value.

A SQL transformation maps a single SQL query (i.e. the original query issued by the application) to a sequence of SQL queries (i.e. transformed queries), generated by WeaQL in an automatic fashion. The transformed queries are then executed in the database in place of the original query. To be able to produce the correct and expected behavior, operations (i.e., transactions) in WeaQL must be executed respecting its causal history. As an example, imagine three replicas R_1, R_2, and R_3. R_1 inserts a record in the database and propagates the operation to R_2 and R_3. After inserting the record in the database, R_2 issues an update on that same record and propagates the changes to its peers R_1 and R_3. If, for some reason (e.g., network congestion), R_3 receives the update operation before the insert, it must wait for the insert operation before installing the update because of the causal dependency between both operations. Indeed, if R_3 promptly executed the update, it would have no consequence since that record had not yet been inserted.

We developed SQL transformations for the insert, update, and delete queries, which are discussed in full detail next.

Insert Record Query

We start by presenting the transformation associated with the insert query, which is the most straightforward and easy to understand. The pseudo-code for this transformation is depicted in algorithm 6.

Specification 6 SQL Transformations: insert record

```
def insert(Record record, VectorClock txnClock):
    let $table = record.table$          ▷ the table in which this record will be inserted

    $record.\_cclock = txnClock$
    $record.\_cclock = txnClock$
    $record.\_delflag = false$

    $table.put(record)$
```

First, the record is augmented with the information regarding the metadata columns. In particular, we assign to **_cclock** and **_dclock** a unique vector clock generated for this transaction, and set the **_delflag** to *false*. Second, a typical insert query is executed to effectively insert the record in the database table.

Delete Record Query

WeaQL does not actually deletes database records. Instead, we use the **_delfag** column to emulate the delete operation. While deleted records remain in the database, they are filtered in the context of select queries. For instance, when performing a select query, only the records with $\_delflag = false$ are retrieved from the database, therefore preserving the natural semantics of the select query. Furthermore, if the original delete query contains a range query (e.g. `delete from foo where bar > 10`), we make it deterministic by querying the database to retrieve the affected tuples *a priori*.

Specification 7 SQL Transformations: delete record

```
 1: def delete(Record record, VectorClock txnClock):
 2:   let policy = record.table.policy          ▷ the policy for this database table
 3:
 4:   if policy = @UPDATEWINS :
 5:     if (txnClock > record._cclock)
 6:       record._delflag = true
 7:   else if policy = @DELETEWINS :
 8:     if (txnClock > record._cclock OR txnClock || record._cclock)
 9:       record._delflag = true
10:
11:   record._dclock = merge(txnClock, record._dclock) ▷ merge: take the greatest value
      for each clock entry
```

Algorithm 7 presents the pseudo-code for the delete operation, which consists of setting the **_delflag** to *true* under specific circumstances. On the one hand, if the policy for this table is @UPDATEWINS, we only "delete" the record if the timestamp associated with this transaction is greater than **_cclock** (line 6), which means that no concurrent update for that particular record was issued. On the other hand, if the policy is @DELETEWINS, this delete operation should prevail even if some concurrent operation also updated this record. This behavior is reflected from lines 7 to 9. In either case, we end this transformation by merging the transaction vector clock with the current **_dclock** for that record, to indicate that the state of this record already reflects the operations previously applied plus the current delete.

Update Record Query

Algorithm 8 outlines the pseudo-code for the SQL transformation developed for dealing with an update query. The variable *values*, that is a parameter of the algorithm, maps each table column with the respective value of this record (excluding the metadata columns). Note that even if an update query only updates a single column, this parameter will include both the non-updated column values and the updated values (i.e., the old unmodified columns/values and the updated columns/values).

An update query is transformed as follows. First, we replace the old content of the record with the values contained in *values*, if the assigned logical timestamp *txnClock* is greater than **_cclock** (line 6). This comparison is a deterministic process which depends **only** on the values of both clocks. This ensures that in every case the content (i.e. the actual values for each column stored in the database) of this record will converge in all replicas, even when concurrent updates are issued over different replicas of the database. For instance, if two replicas R_1 and R_2 concurrently execute two updates to the same

Specification 8 SQL Transformations: update record

```
 1: def update(Record record, Map<Column,Value> values, VectorClock txnClock):
 2:     let $policy = record.table.policy$                ▷ the policy for this database table
 3:
 4:     if ($txnClock > record.\_cclock$)
 5:         for ($Entry < Column, Value > entry : values$)
 6:             $record.column = value$
 7:
 8:     $record.\_cclock = merge(txnClock, record.\_cclock)$  ▷ merge: take the greatest value
        for each clock entry
 9:
10:     if ($policy$ = @UPDATEWINS):
11:         if ($txnClock \parallel record.\_dclock$)
12:             $record.\_delflag = false$          ▷ set record visible, in case some concurrent
        operation deleted it
13:             $record.\_dclock = merge(txnClock, record.\_dclock)$
```

`@LWW*` column of a particular record, only one of the updates will be reflected, in this case, the update whose transaction timestamp is greater. After updating the values (if required), we merge both clocks to indicate that the state of this record already reflects the operations previously applied plus the current update. At the end, if the semantic policy for this table is specified as `@UPDATEWINS`, we additionally check if the **_dclock** of this record is incomparable (i.e. concurrent) with $txnClock$, in which case it would point to the existence of a concurrent delete query, that previously made the record invisible (i.e. deleted). If so, we need to prevail the effects of the update by explicitly resetting **_delflag** to $false$ so that the subsequent queries will observe this record again. This last step is crucial for the following reason. Imagine two replicas R_1 and R_2 concurrently executing an update and delete operation, respectively. After executing the delete operation, when R_2 receives the update operation from R_1, it will first install the update on this record (line 6). However, this record remains deleted (i.e., $\_delfag = true$) because of the delete operation previously executed. However, by explicitly resetting the **_delflag** to $false$ (lines 10 to 13), the effects of the delete operation are canceled. In the end, after the execution of both operations, the previously deleted record will be visible (i.e., not deleted) and in a consistent state when compared with R_1.

Delete Parent Record Query

Operations on records which must guarantee a foreign-key constraint require special treatment. This is because of the implicit relation between parent-records and child-records. On the one hand, child-records cannot be "orphan" upon their insertion. On the other hand, deleting a parent-record may prompt side-effects such as deleting all matching child-records. Hence, we have implemented specific SQL transformations for the operations that manipulate records on each side of the foreign-key relation as to preserve the original semantics, namely when deleting a parent-record and inserting a child-record.

The transformation for deleting a parent-record is shown in algorithm 9. First, we

Specification 9 SQL Transformations: delete parent-record

```
1:  def delete(Record record, List<Records> childs, VectorClock txnClock):
2:    let $policy = record.table.policy$                ▷ the policy for this database table
3:    let $fkPolicy = record.table.fkPolicy$    ▷ the foreign-key properties for this record
4:
5:    if $(fkPolicy.option = RESTRICT\ AND\ childs.size > 0)$
6:      $return;$                                      ▷ abort and notify client application
7:
8:    if $policy =$ @UPDATEWINS :
9:      if $(txnClock > record.\_cclock)$
10:       $record.\_delflag = true$
11:   else if $policy =$ @DELETEWINS :
12:     if $(txnClock > record.\_cclock$ OR $txnClock \parallel record.\_cclock)$
13:       $record.\_delflag = true$
14:
15:   $record.\_dclock = merge(txnClock, record.\_dclock)$
16:
17:   if $(fkPolicy.option = CASCADE)$
18:     $delete(childs)$
19:     if $(fkPolicy.policy =$ @DELETEWINS$)$
20:       $for(child : record.getChilds())$
21:         if $(child.\_cclock \parallel txnClock)$
22:           $delete(child)$
```

verify if the parent-record can be deleted (line 5). This might not be possible when the foreign-key is defined with the option RESTRICT and there are child-records associated with this parent-record. In such case, we abort the operation and notify the client (line 6). The next steps of the transformation (up to line 15) are similar to the "normal" delete record transformation. The difference is that in this case, under some circumstances, deleting a parent-record will cause the associated child-records to be deleted as well. Hence, after deleting the parent-record, we check if the option defined for the foreign-key constraint is CASCADE. If so, we delete all the child-records associated with the current parent-record (line 18). To make this process deterministic, the list of child-records to be affected is retrieved *a priori* by querying the database, allowing this operation to be state-independent, even when executing against different states (e.g. when executing in the remote replicas). Finally, if the foreign-key is tagged with the annotation @DELETEWINS, the transformation must additionally delete all child-records that might have been inserted concurrently. This step, depicted from lines 19 to 22, is the only one which depends on the state of the replica executing the operation. However, this is justified because if some operation actually inserted a child-record, the local replica where the delete was issued in the first place has not yet seen this operation. Hence, we issue a query to explicitly delete all child-records that were inserted concurrently. To summarize, when lines 19 to 22 are executed in the local replica (i.e. the replica which executed the delete in the first place) no records will be affected. However, when those lines are executed in the remote replicas, they will capture all the child-records that might have been inserted concurrently.

Insert Child Record Query

The transformation to insert a child-record is depicted in algorithm 10.

Specification 10 SQL Transformations: insert child-record

```
 1: def insert(Record record, VectorClock txnClock):
 2:     let $table = record.table$
 3:     let $fkPolicy = record.fkPolicy$          ▷ the foreign-key properties for this record
 4:     $record.\_cclock = txnClock$
 5:     $record.\_dclock = txnClock$
 6:     $record.\_delflag = true$                  ▷ insert the child row hidden
 7:
 8:     $table.put(record)$
 9:
10:     if $fkPolicy =$ @UPDATEWINS :
11:         let $parentRecord = record.parent$
12:         $parentRecord.\_delflag = false$       ▷ set parent visible
13:         $parentRecord.\_cclock = merge(txnClock, parentRecord.\_cclock)$
14:
15:     if $(parentRecord.\_delflag = false)$
16:         $record.\_delflag = false$             ▷ set child row visible iff parent is visible
```

We start by augmenting the record with the metadata columns. However, in this case, we assign *true* to the **_delflag**, which means that in the beginning, this record will be inserted as a deleted record. Then, the effective behavior of the transformation of this query will depend on the annotation associated with the foreign-key defined for the affected record.

If the annotation is `@UPDATEWINS`, it means that even if some concurrent operation deletes the parent-record, the parent and child records should be visible in the database after both operations are executed. Hence, in this case, we explicitly set the **_delflag** of the parent-record to *false* (line 12), to reverse some potential delete parent operation that might have been executed in another replica. Moreover, we update the value of **_cclock** of the parent-record (line 13) to simulate an update on the parent-record, therefore preventing concurrent operations from deleting it.

Finally, we end the transformation by conditionally setting the **_delflag** of the child-record to *false* (lines 15 to 16). In particular, the child-record will only be visible if the parent-record is also visible, therefore retaining the parent-child relation in all cases.

The intuition underlying this transformation is the following. If the annotation is `@UPDATEWINS`, by explicitly resetting the **_delflag** of the parent-record, all concurrent parent-record deletes are reverted, and therefore, the condition on line 15 will always be *true*. Hence, the child-record will be visible in every possible case, which complies with the semantic associated with the annotation `@UPDATEWINS`. In contrast, if the annotation is `@DELETEWINS`, the child-record will be deleted if a concurrent parent-record is executed, because the delete parent operation will delete all child-records that were inserted concurrently. As an example, let us imagine two replicas R_1 and R_2 concurrently inserting a child-record and deleting a parent-record, with a foreign-key tagged with `@DELETEWINS`. At first, R_1 successfully inserts the new child-record in the database. When R_1 receives the delete parent operation, the newly inserted child-record will be deleted because the operation was concurrent with the delete (line 22 of algorithm 9). When R_2 receives the insert operation, after its execution, the child-record will not be visible because its parent-record is not visible. Therefore, after the execution of the two operations, the parent and child records will be in a deleted state in every replica of the database.

3.8 Summary

In this chapter, we presented the key mechanisms which WeaQL employs to achieve the goals delineated in section 3.1. Table 3.3 summarizes what we have discussed so far and maps each mechanism with the corresponding objective. In section 3.4 we presented schema annotations which allow developers to specify the desired semantics for reconciling conflicting operations. In section 3.5 we detailed how WeaQL leverages metadata columns to ensure state-convergence. In section 3.6 we discussed how SQL transformations allow WeaQL (in some cases) to protect invariants without resorting to coordination. Finally, we presented the SQL transformations algorithms employed by our solution in section 3.7.

In the next chapter, we will present the internal structure of WeaQL, describe its core components, give implementation details that are relevant for a clear understanding of our solution, and cover practical challenges that had to be tackled in this work.

WeaQL Summary	
Goal	**Mechanism**
Multiple Merging Policies	DDL Annotations SQL Transformations
Preserve SQL Invariants	SQL Transformations Coordination Service
State Convergence	SQL Transformations Metadata

Table 3.3: WeaQL Summary

Chapter 4

Implementation

This chapter discusses the implementation details of our solution, described in Chapter 3. In Section 4.1 we focus our attention on the internal architecture of WeaQL. Sections 4.2 and 4.3 comprises relevant aspects of our implementation essential for a clear understanding of our solution.

4.1 WeaQL Architecture

We now present the internal architecture of WeaQL. Throughout this section, we will describe each core component of WeaQL, their purpose and how they interconnect with each other.

4.1.1 Overview

An high-level overview of our system is depicted in Figure 4.1. At its core, WeaQL has two central components. The Proxy is a library that is coupled together with the application program. The Replicator component has an instance tied together with each replica of the database. External to the core of WeaQL (but still an important component of our solution) is Zookeeper [Hun+10], which we leverage as the coordination service which WeaQL resorts to preserve some of SQL integrity constraints.

With WeaQL, when an application opens a connection to the database (via JDBC), WeaQL transparently intercepts this process and spawns a dedicated proxy instance, which in practice replaces the original JDBC connection by exposing the same interface as the original JDBC. Each proxy instance is bound to a single replicator instance during its lifetime. This is because we want to preserve the read-my-writes property [Ter13] between transactions issued by a single client. If we were to allow a proxy to switch between replicas, we could not ensure this property since both replicas may not have

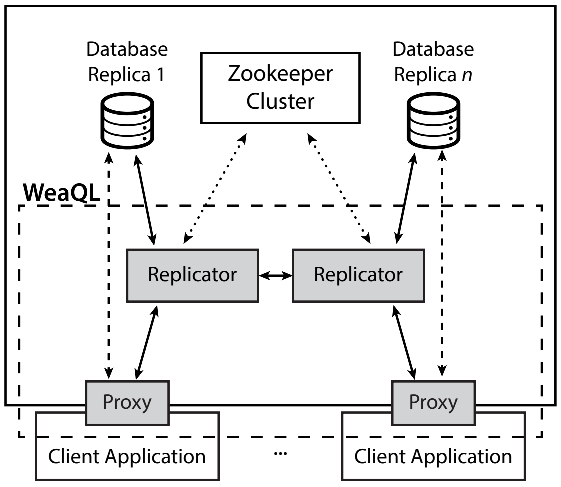


Figure 4.1: WeaQL architecture. Dashed arrows represent communication between Proxies and database replicas for executing transactions and making transactions commutative. Solid arrows indicate communication for committing and replicating transactions. Dotted arrows illustrate communication between Zookeeper and Replicator for coordinating conflicting transactions.

had the time to synchronize their state and hence be missing the effects of some of the transactions, previously completed by the client.

A transaction in WeaQL can be decomposed in two (potentially three) phases. The proxy is responsible for executing the first phase, which corresponds to the execution of the transaction in a private *sandbox* environment. The second phase corresponds to the coordination process, which may not be required for every transaction. The third and last step is handled by the replicator and corresponds to the execution of the updates over persistent storage (immediately on the local replica and asynchronously on the remote replicas). In fact, a golden rule that epitomizes WeaQL is the following. Proxies write to temporary memory, replicators write to persistent storage. The three steps mentioned above will be described in more detail throughout this chapter.

We implemented WeaQL in a total of 32k lines of Java code. We reused SIEVE's source code [Li+14] and integrated it directly in WeaQL. In particular, SIEVE already provided mechanisms to parse the table and attributes annotations from the database schema file. We extended that functionality so the full length of annotations used by WeaQL could be captured.

4.1.2 Proxy

The proxy module consists of a customized JDBC driver, which has two main purposes. First, it orchestrates the execution of transactions in the local sandbox, and second, it is responsible for mapping the original SQL queries into their commutative form (i.e. using the SQL transformations described in section 3.7). In practice, the sandbox environment consists of temporary tables which mimic the original structure denoted by the database schema. For instance, if an application contains a table *user* with 2 columns, each sandbox will contain a table $tmp*\_user$ with the same columns, where * corresponds to the identifier of the proxy instance. Note that the sandbox does not hold a copy of the data, but only the tables' structure. In fact, prior to the beginning of a transaction, all the tables in the sandbox are emptied.

To get a clear picture of how the proxy works, we now explain the sequence of steps that are performed, since an application starts a transaction until it issues the commit operation. A transaction starts when the application issues the first SQL query. Each SQL query is intercepted by the proxy, which has two tasks. First, it executes the query in the sandbox and returns the result back to the client. In this step, the original query is executed with a minor caveat. Because deleted records are kept in the database, we must filter these records so they are not part of any result seen by the client application. To do so, we append the condition $\_delflag = false$ to every select query in order to exclude deleted records from the result sets exposed to the application.

The motivation for executing transactions in the sandbox first lies in the fact that we want to minimize the coordination overhead. For instance, consider a transaction that inserts multiple records one by one in a given table *user*. In this particular case, WeaQL would have to contact the coordination service to ensure that usernames are unique. Because we cannot predict what will be the queries issued by the application, WeaQL would need to contact the coordination service prior to each insert, issuing *n* RPCs, where *n* equals the number of inserted records. However, by executing transactions first in the sandbox, this number is lowered to 1. This is because transactions are executed in the sandbox using an optimistic approach, and, at commit time, a single RPC is issued by the replicator with a list of all usernames that the transaction intends to insert. As another example, if a transaction tries to insert a record with an auto-increment column whose values must be sequential, the proxy assigns a local temporary *id* to avoid immediate coordination, and at commit time, the replicator replaces this value with the definitive value after contacting the coordination service.

During the execution of transactions the proxy keeps track of all the changes that the execution of the transaction will produce. Then, after receiving the commit command from the application, the proxy generates the commutative SQL statements and ships them to the replicator. Because some key information regarding these SQL statements resides in the replicator (for instance the transaction's vector clock), the proxy uses placeholders whenever needed, which are later replaced by the replicator prior to committing the transaction.

4.1.3 Replicator

The Replicator module implements the following semantics: *i*) committing transactions in the local database, *ii*) handling all interactions with the Zookeeper cluster, and *iii*) communicating with other replicators to exchange transactions logs.

In more detail, upon receiving a request to commit a transaction from the `proxy`, the `replicator` performs a lightweight check to verify whether the transaction requires coordination or if it can be safely executed locally without coordination. In the case where coordination is required, the `replicator` issues an RPC to the coordination service, which either allows the transaction to commit or not. If the transaction is not allowed to commit, the `proxy` is notified of the error and the transaction aborts. This might happen, for instance, when a transaction tries to insert a record with a value for a unique-key that is already being used.

If the coordination request succeeds, the `replicator` does the following. First, it generates a new unique vector clock to be associated with the transaction. Second, he replaces all the placeholders left by the `proxy` in the list of `SQL` statements with the appropriate values. Third, it effectively commits the transaction in persistent storage. If the commit succeeds, the transaction log is sent to all its peers (i.e. other `replicators`) in a lazy fashion and the `proxy` is notified of the success of the transaction.

When a `replicator` instance receives a transaction log from another `replicator`, it first checks if the transaction can be executed without violating causality (i.e. if that replica has received all transactions from remote replicas that the incoming transaction may depend on, according to each entry of the vector clock). If so, the transaction is executed. Otherwise, the transaction is added to a queue of pending transactions. A collection of background threads are then responsible for emptying this queue periodically by executing pending transactions whose causal dependencies have already been met locally.

The number of threads that are allocated to executing transactions from other `replicators` can be tuned depending on the applications needs. In particular, to maximize throughput, we can allocate a single thread to this task (which can even run at a lower priority and include some sleep time), thus prioritizing the execution of local pending requests. In contrast, if the application prefers to reduce the staleness of the data, it can make use of more threads in order to reduce the time it takes to incorporate transactions from remote replicas.

4.1.4 Zookeeper Extension

The Zookeeper is a centralized coordination service [Hun+10], which enables clients to implement sophisticated synchronization protocols. In its core, the Zookeeper uses *znodes*, which resembles files and folders in a typical file system, that can be created, deleted, updated, and so on via the interface Zookeeper exposes to its clients.

Extensible Zookeeper (EZK) [Dis+15] allows the Zookeeper [Hun+10] coordination service to be augmented with client code (extensions), which are dynamically loaded and executed in Zookeeper replicas. In particular, EZK allows custom code to execute in response to each original Zookeeper API call. One of the main advantages of EZK is that it allows clients to batch multiple API calls in the same RPC while ensuring its atomic execution. This drastically differs from Zookeeper, in which each API call requires an RPC. This property enables clients to execute complex operations with very low latency.

Our coordination service is implemented as a Zookeeper extension by creating, updating, and removing *znodes* that represent the contended resources. For instance, each value already used by a unique-key constraint is represented by a *znode*. When our coordination service receives a request from the `Replicator`, it only has to check if the corresponding *znode* exists. Similarly, sequential *ids* are generated by atomically incrementing the value of the *znode* that corresponds to the auto-increment field.

4.2 Conflicts Detection and Lost Updates

As discussed in Section 3.5, each database record is equipped with a vector clock (**_cclock**) which tracks the logical time in which each record was last modified. In particular, this vector clock always reflects the logical time when the last column was updated, which means that WeaQL is not able to distinguish between updates across different columns of a single record. More specifically, given a record and a vector clock, WeaQL is not able to recognize which column was last updated. For this reason, we say that the conflict detection is made at the record level. Achieving a more fine-grained conflict detection is possible. For instance, if we were to use a logical clock for each table column, we could keep track of changes in individual columns. However, this approach would also consume more storage, at least $2n$ columns per table, where n equals the number of original table columns. For this reason, we considered a record-level conflict detection a good compromise between spatial complexity and the granularity of conflicts detection.

The approach that WeaQL follows to deal with conflicting operations raises the possibility of losing updates during the process of reconciling concurrent operations. As an example, imagine two concurrent operations on the same record modifying columns c_1 and c_2 respectively, both with the annotation `@LWW*`. Since the vector clocks of both operations are incomparable (i.e. transactions are concurrent), WeaQL will only install one of the updates, namely the one whose vector clock value is greater. This decision is deterministic and is achieved by comparing each entry of the vector lock. For instance, if operation op_1 has a vector clock $vc_1 = [5,2,3]$ and op_2 has a vector clock $vc_2 = [4,2,4]$, despite the fact that these clocks are incomparable, op_1 will prevail since by looking at the first clock entry we verify that $5 > 4$. Therefore, the effects of operation op_2 will be lost. Note however that we explicitly specified that the columns updated were tagged with the `@LWW*` annotation. Indeed, this analysis only makes sense for columns where its merging policies are last-writer-wins. If the updated columns were tagged with `@DELTA*`, no updates would be lost since updates on these columns are seen as increments or decrements of a specified *delta* value, which can be easily merged as discussed in Section 3.4.2.

The above discussion unveils a fundamental difference on how the state of each replica progresses in WeaQL when compared with classic, strong consistent solutions. In these systems, all replicas orderly progress through the same sequence of internal states. In contrast, this is not true in WeaQL. This can be easily understood by revisiting the previous example. In particular, the state of the replica that issued op_1 in the first place will never reflect the effects of op_2. However, the replica that issued operation op_2 will temporarily reflect its effects, until op_1 is propagated and executed in this replica. Hence, despite the fact that the state of both replicas eventually converges, they did not go through the same sequence of states during their execution.

4.3 Stored Procedures

To efficiently support the implementation of the SQL transformations described in Section 3.7, we implemented a set of stored-procedures whose goals are to compare vector clocks. While internally all these functions compare the values of two vector clocks, they output different values. As an example, we have implemented a function `isConcurrentOrGreaterClock(oldClock,newClock)` which returns *true* if *newClock* is concurrent or greater then *oldClock*, and *false* otherwise. However, another function `clockIsGreater(oldClock,newClock)` returns *true* only if *newClock* dominates *oldClock* and *false* otherwise.

These procedures have an active role in the SQL transformations performed by WeaQL. In particular, their output values are used as boolean flags which are appended to the *where* clause of many SQL statements in order to cancel their effects. For instance, let u_1 (issued by replica r_1) and u_2 (issued by replica r_2) be two concurrent updates on the same record. Because these operations are trying to update a `@LWW*` column, the final state of the record will only reflect one of the updates (let it be u_1). Therefore, when r_1 receives u_2 from r_2, it must not install the respective update. To do so, we append the `clockIsGreater` function to the *where* clause of the update, being the first argument **_cclock** and the second argument the clock associated with the operation. In effect, the operation query would look similar to:

```
UPDATE * WHERE * AND clockIsGreater(_cclock,txnClock)
```

where * represents the content of the original query (e.g. columns updated and the original *where* clause). We now resume the previous example. After r_1 executes u_1, the **_cclock** is updated with the vector clock value associated with that operation. Then, when this replica receives the u_2 from r_2, because the vector clock of u_1 is greater than the clock of u_2, the output of executing the function `clockIsGreater(_cclock,txnClock)` will be *false*. Therefore, the SQL query would look similar to:

```
UPDATE * WHERE * AND FALSE
```

Consequently, this update would not be in fact executed, which was the desired behavior. In contrast, when r_2 receives the operation from r_1, the update will be effectively executed because the function will output *true* instead of *false*. Note, however, that in this example, we replaced the call of the function by its output value for clarity. Indeed, what is issued to the database is the function's call itself. It is the database's engine responsibility to execute the function and calculate the output value.

To summarize, all the comparisons of vector clocks required by the SQL transformations described in Section 3.7 are achieved by calls to stored procedures implemented in the database engine. While originally these procedures are not installed in the databases servers used by applications, we have developed a small program which automatically installs them, therefore removing the intervention of programmers in this process.

4.4 Summary

In this chapter, we shifted the focus to the implementation details of WeaQL. We started by describing each core component of WeaQL and its main purposes. We then materialized our coordination service, in particular by building on previous work and developing our own Zookeeper extension. We concluded by exploring relevant implementation details, more specifically how WeaQL handles conflicts detection and how stored procedures aids in the implementation of the SQL transformations.

Chapter 5

Evaluation

In this chapter we experimentally evaluate WeaQL. Our experimental evaluation focus on trying to answer the following questions:

- What is the overhead introduced by WeaQL?
- What are the performance benefits of WeaQL when compared with state of the art solutions for replicating relational databases?
- How does WeaQL performance vary with the number of replicas in the system?
- What is the overhead imposed by the use of the coordination service?

The remainder of this chapter is organized as follows. We start by describing the experimental setup in which all experiments were conducted. Then, we report the results of the experiments and discuss the results. We conclude by summarizing the most relevant aspects of the evaluation and by drawing some additional conclusions. Throughout this chapter, we will use the words "transaction" and "operation" interchangeably.

5.1 Experimental Setup

All experiments were conducted in a local cluster with a maximum total of 10 machines. We used MySQL 5.6 as storage engine and Java SE 7 to run WeaQL. The machines running database instances (which varied between one and five) had a 2 Quad-Core AMD Opteron 2376 2.3 GHz processor, 16GB of RAM and 2 Gigabit NICs. For experiments with WeaQL, we deployed a `replicator` instance co-located with each database instance, and deployed a Zookeeper cluster with 3 instances. Clients were spread across 5 dedicated machines to avoid client-side bottlenecks.

The experiments were performed by having a variable set of clients connect via a JDBC driver to the database instances, and continuously executing transactions during 60 seconds, after an initial ramp-up time of 20 seconds. We call *run* to a single execution of $20 + 60$ seconds experiment with a fixed number of clients. All the experiments were conducted by executing three independent runs, and all the plots presented throughout this chapter reflect the average values of the three runs. Before running each experiment, we loaded the full database into memory to eliminate potential I/O bottlenecks. Every experiment was executed over a clean version of the database so that the evolution of the database state did not impact the measured results.

5.1.1 Baselines

We compared WeaQL with two solutions for replicating relational databases across a set of nodes, namely MySQL-Cluster [MyS15b] and Galera-Cluster [Gal15]. We chose these systems for two main reasons. First, they are two prominent solutions used in the industry community, and second, their approaches to replicate the state diverge in fundamental ways, therefore creating the opportunity to compare our work with two different solutions.

MySQL-Cluster follows a shard-based technique, which means that each replica of the system only holds a portion of the data. In this configuration, operations may access data located on remote nodes, in which cases MySQL-Cluster resorts to the 2PC protocol to ensure that all nodes involved in the transaction agree on the commit decision.

Galera-Cluster on the other hand, uses a multi-master full replication approach to replicate the state among a set of nodes. Since every node of the system holds a full copy of the data, Galera-Cluster waives the need of executing distributed transactions. However, because two replicas may concurrently access the same data, Galera-Cluster resorts to group communication primitives to determine a total order of operations and ensure that all replicas evolve through the same sequence of internal states. While the process of agreeing on the order of execution of operations is synchronous, the process of installing updates in remote replicas is asynchronous.

5.1.2 Benchmark

We evaluated WeaQL using the TPC-C benchmark [Cou15]. The TPC-C benchmark in an online transaction processing (OLTP) benchmark which simulates an e-commerce environment. Users interact with the database by executing a set of different transactions, which range from read-only to update-intensive transactions. In particular, the TPC-C standard defines five different transactions types, which are summarized next. The *order-stat* transactions is a mid-weight read-only transaction which checks the status of an order for a particular customer. The *delivery* transaction is a read-write operation that processes a batch of new orders yet to be delivered. The *stock-level* transaction is a heavy read-only transaction that examines the current stock level of items that have

Workloads Specification		
Transaction	**Workload 1**	**Workload 2**
New Order	45	40
Payment	43	0
Delivery	4	0
Stock Level	4	30
Order Stat	4	30

Table 5.1: Workloads Specification

been recently purchased. The *payment* operation is a light-weight read-write transaction that updates the customer's balance to simulate the payment of a new order. Finally, the *new-order* operation completes a new purchase by creating a new order and inserting multiple purchased items.

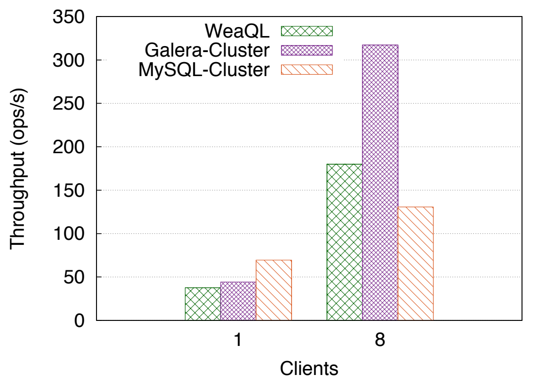
In TPC-C, items are distributed across multiple warehouses, which serve ten different districts each. The TPC-C standard specifies that *ids* generated for each *new-order* must be assigned sequentially, on a per district basis. This invariant is not directly encoded by any `SQL` integrity constraint; instead, it is maintained by placing a select before each insertion for querying the next available *id*. As a result, this application-level invariant cannot be directly handled by our system. To make TPC-C run with WeaQL, we slightly modified the TPC-C code so that the previously mentioned constraint was changed from the application-level logic to a single `SQL` constraint. In particular, we set this column as an auto-increment attribute and tagged it with the annotation `@SEMANTIC`. This way WeaQL will generate sequential *ids* to each new-order. Note however that this modification increases contention since our system must generate sequential *ids* for all *new-order* transactions, in contrast to Galera-Cluster and MySQL-Cluster which generate sequential *ids* only on a per district basis.

5.1.3 Workloads

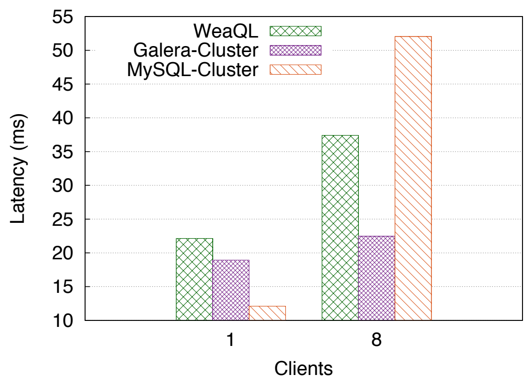
To cover a wide range of application scenarios, we have defined two distinct workloads with different read/write ratios, which are presented in Table 5.1. The first workload conforms to the one specified by the TPC-C standard, which consists of 92% of read-write transactions (Workload 1). We have also defined a more light-weight workload where read-only transactions dominate the transaction space. In this workload (Workload 2) 70% of the transactions are read-only transactions.

5.2 WeaQL Overhead

WeaQL is a middleware layer deployed between the application and the database engine. Therefore, it inherently introduces some overhead since operations must pass through



(a) Measured Throughput

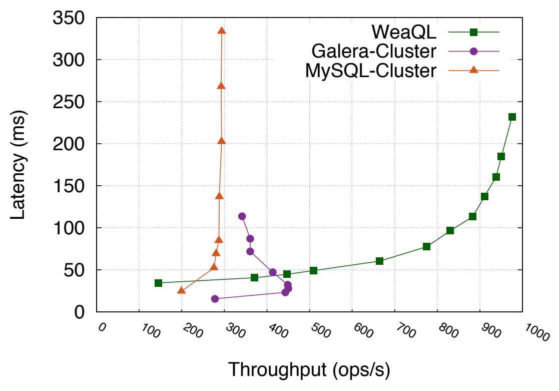


(b) Average Latency

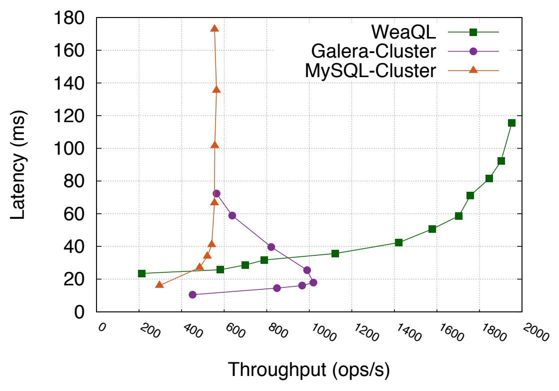
Figure 5.1: WeaQL Overhead for a single replica experiment

WeaQL instead of access directly the database engine. To measure the overhead imposed by our solution, we deployed a single replica and executed an experiment with one and eight clients issuing operations using Workload 1. In this configuration, the results obtained accurately reflect the performance of each solution for a single replica, without having interference produced by the replication mechanisms.

Figure 5.1 presents the throughput and average latency obtained for this experiment. Focusing on the results with one client in Figure 5.1a, we observe that WeaQL is outperformed both by Galera-Cluster and MySQL-Cluster. In particular, Galera-Cluster yields 1.18*x* the throughput in relation to WeaQL while MySQL-Cluster surpasses WeaQL by a factor of 1.85*x*. The average latency depicted in Figure 5.1b is also coherent with these results, showing incrementally higher latencies for WeaQL, Galera-Cluster and MySQL-Cluster. When the number of clients is increased, the results obtained for WeaQL and Galera-Cluster remained consistent with our expectation. In more detail, in the configuration with eight clients, Galera-Cluster yielded 1.76*x* the performance of WeaQL and experienced 0.6*x* the latency observed in WeaQL. The results obtained with MySQL-Cluster were unforeseen in two different ways. First, we did not expect the performance of MySQL-Cluster to be so high in the configuration with one client. Second, the sudden performance loss observed between the experiments with one and eight clients was also surprising. Even though we are not able to explain these results, we can speculate that this behavior is related to the database engine used by MySQL-Cluster. While both WeaQL and Galera-Cluster in their core use the InnoDB storage engine [MyS16b], MySQL-Cluster on the other hand resorts to the NDB storage engine [MyS16a]. Nevertheless, the low performance of MySQL-Cluster with multiple clients will again become noticeable further ahead on the experimental results reported in Section 5.4.



(a) Workload 1



(b) Workload 2

Figure 5.2: Latency-Throughput with 3 replicas

5.3 Performance Evaluation

In this section, we want to evaluate the following factors. First, what is the raw performance of WeaQL in terms of throughput (i.e., operations per second) when compared to Galera-Cluster and MySQL-Cluster. Second, how different system loads impacted the latency of transactions and overall throughput. We evaluated these aspects of the systems using a different number of replicas to understand how the different replication approaches affected these metrics.

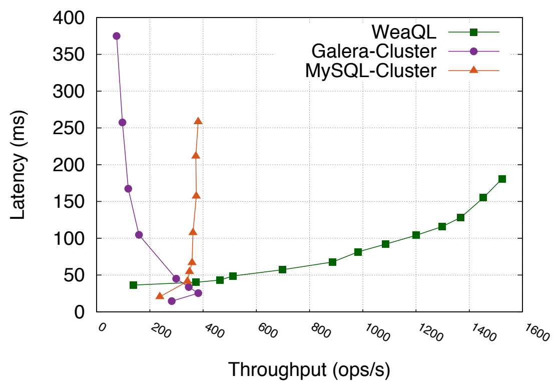
We conducted two experiments, with three and five replicas. In each experiment, we increased the load on the system by increasing the number of clients issuing operations concurrently. Each experiment was executed with the two different workloads described previously. We measured the number of operations successfully executed, and the average latency experienced by the end clients.

Figure 5.2 reports the results obtained from experiments with three replicas. In these plots, the $x-axis$ refers to the number of successful operations while the $y-axis$ represents the average latency of the operations. The marked dots along the curves represent the different loads imposed on the system, in this case, the number of connected clients issuing operations. Ideally, the curves would be horizontal ones, which means that increasing the number of outstanding clients is translated into a higher throughput without affecting the latency experienced by them.

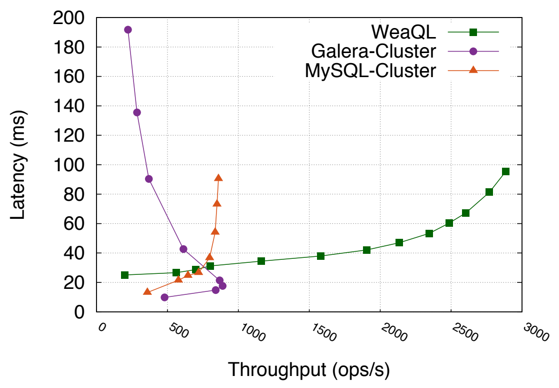
The first noteworthy observation is that for a low number of users (marked dots in the beginning of the lines), Galera-Cluster and MySQL-Cluster outperform WeaQL, both in Workload 1 and Workload 2. This is coherent with our expectation and is justified by two factors. First, the overhead penalty inherent to WeaQL (as discussed in section 5.2) is translated into lower throughput per replica in WeaQL. Second, for a reduced number of clients, the contention derived from the concurrent access to the same records is not high enough to severely compromise the performance of Galera-Cluster and MySQL-Cluster.

Taking as example the run with five clients (first marked dot on the curves), Galera-Cluster achieves 1.9*x* the throughput when compared with WeaQL. On the other hand, MySQL-Cluster yields 1.4*x* the performance in relation to WeaQL, a lower value when compared with Galera-Cluster. This latter result seems to contradict the results obtained in section 5.2, where MySQL-Cluster yielded better performance (for the experiment with one client). However, it is sustained by the fact that MySQL-Cluster may be required to execute transactions involving multiple nodes of the system (in contrast to Galera-Cluster where each node hosts a full replica of the data), which increases the experienced latency and degrades performance.

When we increase the number of clients, WeaQL was able to achieve a much higher throughput while degrading the perceived latency at a much slower rate when compared both to Galera-Cluster and MySQL-Cluster. Note that for WeaQL, we had to further increase the load on the system to be able to saturate it, which is why the line referent to WeaQL has more marked dots. There are a couple of reasons that justifies the results obtained. On the one hand, by increasing the number of clients issuing operations, the chances that those clients concurrently modify the same data also increase. This fact is the most determinant aspect that justifies the behavior observed for Galera-Cluster. In particular, in Galera-Cluster transaction execution follows an optimistic approach. At commit time, the transaction's write-set is broadcast to the entire cluster which then must certificate it's validity by checking possible conflicts. If this certification process fails, the transaction aborts. As increasing the number of clients leads to a higher number of conflicts (for instance when two concurrent transactions issued to different replicas modify the same record), the number of transactions that fail to pass the certification process also increases. Consequentially, the abort rate increases which result in a loss of throughput. On the other hand, the performance degradation observed in MySQL-Cluster is mostly due to the fact that a transaction that spans multiple nodes requires the execution of the 2PC protocol to decide whether or not it is successfully committed. In this context, the perceived latency of each transaction is dominated by the slowest node to acknowledge the commit. While a good partition schema of the data might reduce the chances of requiring distributed transactions, MySQL-Cluster does not actively tries to minimize the need to use 2PC. In particular, the data partition schema employed by MySQL-Cluster is based on a *hash* function applied in the primary-key of each record, making this approach completely "blind" to the application scenario. In contrast, WeaQL does not require any system-wide decision (besides the ones enforced by the coordination service) to be able to successfully commit a transaction. Furthermore, even if two transactions modify the same record concurrently, instead of aborting the transaction, WeaQL is able to resolve the associated conflict by following the reconciliation policy specified by the programmer. Hence, in WeaQL, increasing the number of clients does not lead to a higher number of aborts, allowing WeaQL to achieve much higher throughput. In the end, when compared with Galera-Cluster, WeaQL yields 2.2*x* the peak throughput for the Workload 1 and 1.9*x* the peak throughput for Workload 2 (figure 5.2b). As to



(a) Workload 1



(b) Workload 2

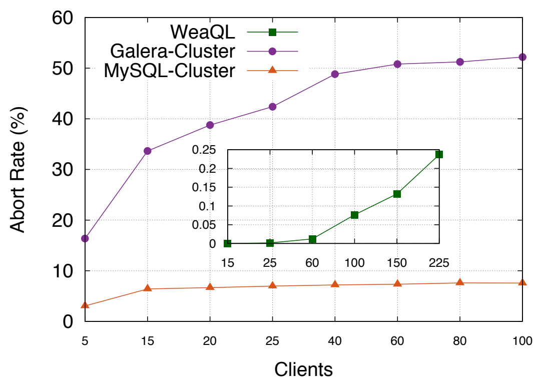
Figure 5.3: Latency-Throughput with 5 replicas

MySQL-Cluster, WeaQL achieves 3.3*x* the peak throughput for Workload 1 and 3.5*x* for Workload 2. The comparison between the results obtained for both workloads also unveils an interesting difference. More specifically, the lower gains in throughput of WeaQL when compared with Galera-Cluster for Workload 2 is due to the fact that read-only transactions are more prominent. Consequently, the decrease in the number of conflicts leads to a lower number of aborted transactions for Galera-Cluster. However, the inverse behavior is observed with MySQL-Cluster. Indeed, the increased read ratio of Workload 2 benefits more WeaQL than MySQL-Cluster. This is because in MySQL-Cluster, read-only transactions may still require accessing data on remote nodes. Another aspect that differentiates MySQL-Cluster from Galera-Cluster is the fact that increasing the number of clients does not significantly deteriorates the throughput of MySQL-Cluster, which contrasts with the behavior exhibited by Galera-Cluster. This can be observed in both workloads after both solutions achieve their peak throughput. At that point, adding more clients in Galera-Cluster results in a significant loss of throughput, while in MySQL-Cluster it results in a fast increase in latency and a less pronounced loss of throughput. We now move on to the experiment with five replicas, whose results are depicted in Figure 5.3. In a general way, the discussion above and consequent conclusions are also reflected in the results of these experiments. However, the higher number of nodes in the system impact Galera-Cluster and MySQL-Cluster in different ways. In the case of Galera-Cluster, more replicas in the system is translated into more deadlocks and concurrent accesses on the same data, and therefore, lower performance. Indeed, Galera-Cluster achieved better performance in the scenario with three replicas than in this scenario. This fact exposes the scalability issues associated with group communication protocols (a topic which will be discussed in more depth in section 5.4). MySQL-Cluster on the other hand, is able to achieve better throughput when compared to the three replicas scenario. In this configuration, despite the probability of transactions accessing data on remote nodes increases (due to the greater number of shards present in the system), each replica must process fewer operations since the load in the system is distributed among a larger number of processes. This allowed MySQL-Cluster to slightly increase its peak throughout when compared with the previously discussed scenario. The results obtained show that when compared with Galera-Cluster, WeaQL yields 4.0*x* the peak throughput for the Workload 1 and 3.3*x* the peak throughput for Workload 2 (figure 5.3b). As to MySQL-Cluster, WeaQL achieves 3.9*x* the peak throughput for Workload 1 and 3.4*x* for Workload 2.

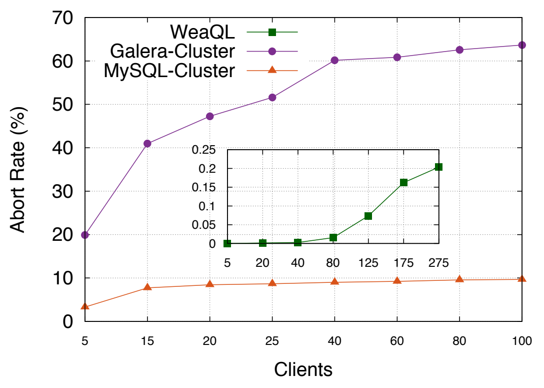
5.3.1 Abort Rates

To support the conclusions drawn in the previous section, and further understand the observed results, we have also investigated the abort rates for Workload 1 associated with each solution. This is an important aspect to assess as some of our conclusions, particularly regarding the observed performance of Galera-Cluster, were justified by the high number of aborted transactions derived from the high contention on resources. Indeed, contention have a very different impact in WeaQL in regard to both baselines. On the one hand, as discussed in Section 5.3, Galera-Cluster executes transactions based on the local state of each replica, requiring no cluster-wide synchronization in this step. At commit time, however, if the certification process fails, the transaction aborts and the whole effort of executing the transaction is wasted. On the other hand, MySQL-Cluster follows a pessimistic lock-based approach. In this context, a transaction may abort for instance due to a deadlock that arises during its execution. In such case, the transaction is immediately aborted, i.e., sooner when compared to Galera-Cluster. For this reason, aborts in Galera-Cluster have a greater negative impact on performance when compared with MySQL-Cluster because more work is wasted. WeaQL contrasts with both approaches in the following way. First, conflicting transactions are not aborted, instead, they are reconciled. Second, each transaction is first executed in its own sandbox. Then, at commit time, the side-effects of each transaction are applied as a batch in the main storage, drastically reducing the time transactions require (exclusive) access to the shared database objects. As a consequence, aborts in WeaQL are virtually nonexistent.

Figure 5.4 depicts the abort rates observed in all three systems for the experiments with three and five replicas. In the case of WeaQL, we had to include a separate plot with a different scale, (placed in the center of each main plot), to be able to notice the curves referent to its abort rates. Nevertheless, the results are consistent with our expectation. In more detail, we observe that the abort rates experienced by Galera-Cluster are consistently higher with the number of clients issuing operations to the system. This observation supports our claim in section 5.3, where we pointed out that the poor performance of Galera-Cluster was mainly derived from the contention of resources, which resulted in many conflicting transactions. Another interesting observation is that MySQL-Cluster does not experience a dramatic increase in the abort rate as the system is targeted by an increasingly load. Indeed, this behavior is coherent with the results obtained for



(a) Three replica experiment



(b) Five replica experiment

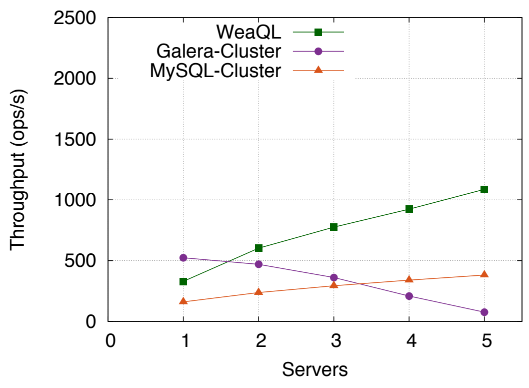
Figure 5.4: Abort Rates Comparison

MySQL-Cluster in Figures 5.2a and 5.3a, where increasing the load in the system had a low impact on the system throughput but high impact on the latencies experienced by the clients. Finally, as discussed previously, we observe that WeaQL experiences close to 0% abort rates. In fact, aborts in WeaQL only happen when *i*) the coordination service enforces the abort to preserve SQL integrity constraints, or *ii*) a deadlock is found during the execution of the updates in main storage. In the case of the TPC-C benchmark, the coordination service never aborts a transaction because it is the one responsible for distributing the *new-order ids*. Furthermore, deadlocks are very rare in WeaQL because we leverage the calls addBatch() and executeBatch() [Ora16] (provided by the standard JDBC specification) to execute transactions in main storage, which makes this process very efficient.

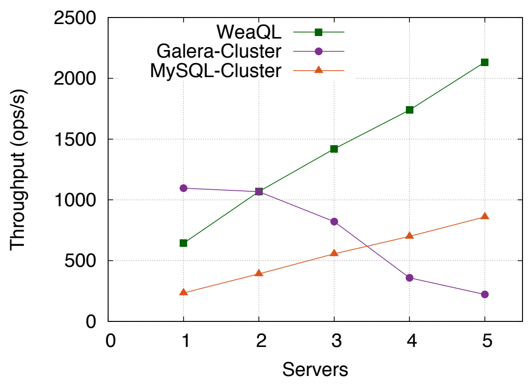
5.4 Scalability Evaluation

We now shift the focus to study how WeaQL scales when more replicas are added to the system. This is an important aspect to measure because when applications need to improve their performance, a popular solution is to add additional resources to the system. By evaluating the scalability of the system, we can predict what are the performance benefits that applications might extract from the addition of new resources. To compare the scalability achieved by WeaQL when compared with Galera-Cluster and MySQL-Cluster, we have ran experiments with up to five database replicas. In each experiment, we have increased the number of clients proportionally to the number of replicas in the system. For instance, for experiments with one replica, we used 20 clients, for experiments with two replicas we resorted to 40 clients, and so on, until 100 clients in the five replicas setup.

Figure 5.5 reports the results obtained for the two different workloads. The first noticeable observation is that for a single replica experiment, the results obtained are consistent



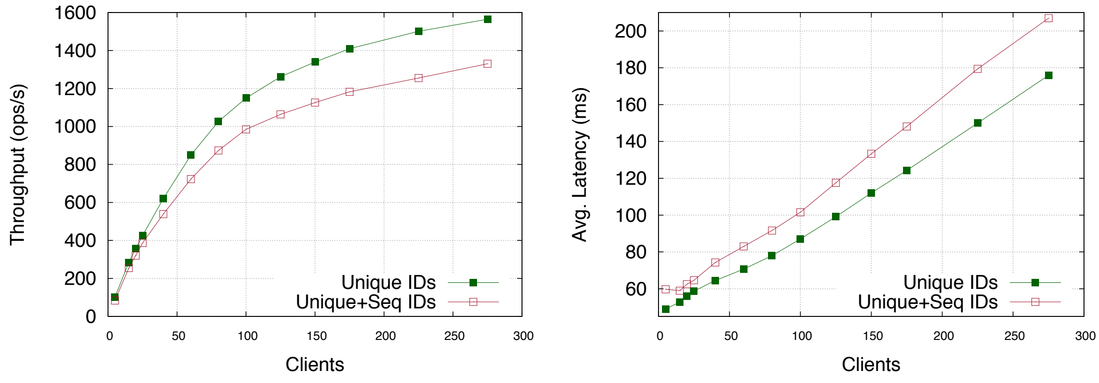
(a) Workload 1



(b) Workload 2

Figure 5.5: Scalability Comparison

with our intuition. In particular, we observe that Galera-Cluster yields the best performance when compared to WeaQL and MySQL-Cluster, while WeaQL outperforms MySQL-Cluster. This latter result is coherent with the results that were presented previously in section 5.2. Despite Galera-Cluster outperforming WeaQL and MySQL-Custer for a single replica system on both workloads (see Figure 5.5a and Figure 5.5b), the results also show that adding new resources to Galera-Cluster results in a loss of performance, even for Workload 2, where read-only transactions dominate the workload. This result is reasonable and the rationale for this behavior is two-fold. The first reason was already pointed out in section 5.3 and is associated with the fact that increasing the number of replicas/clients also leads to an increase in contention. Second, because Galera-Cluster is a full replicated database, each transaction must be installed in each replica of the system. Therefore, when the number of replicas increases, the work done by each one also increases proportionally. In contrast, MySQL-Cluster does not suffer from this handicap. In fact, the sharding approach employed by MySQL-Cluster achieves the opposite effect. In particular, by increasing the number of shards (i.e., replicas) the application's requests become more distributed across all the shards and, therefore, each replica effectively has to process a lower number of transactions. WeaQL on the other hand also experiences the drawbacks of full replication as it has to execute every transaction in every replica of the system. However, in contrast to Galera-Cluster, WeaQL mitigates this aspect by employing weakly consistent replication, and therefore prioritizing the execution of local transactions while deferring the execution of remote updates to background threads. This allows WeaQL to exhibit better scalability when compared with Galera-Cluster and MySQL-Cluster. For WeaQL, the results obtained shows a scalability factor of 3.3*x* for Workload 1 and 3.2*x* for Workload 2 in the five replica setup. In other words, by increasing the load in the system and the number of resources (i.e., replicas) by a factor of 5, WeaQL was able to extract 3.3*x* and 3.2*x* the performance when compared to a single replica setup. In contrast, MySQL-Cluster achieves a considerably lower scalability factor,



(a) Throughput comparison

(b) Latency comparison

Figure 5.6: Performance comparison between coordination-required and coordination-free operations

particularly with Workload 1, yielding only 2.4*x* the performance of the one replica setup, while achieving a 3.7*x* factor with Workload 2.

5.5 Coordination Overhead

In all experiments realized so far, *new-order* transactions generated sequential *ids*, as described in the TPC-C specification. For WeaQL, this required each *new-order* transaction to retrieve the next available *id* by contacting the coordination service. We now proceed to gauge the overhead introduced by this mechanism. In more detail, our goal is to measure the impact on the latency and throughput imposed by the coordination mechanism employed by WeaQL. To evaluate these factors, we made the following experiment. We set the annotation associated with the *id* column in the *orders* table to `@NOSEMATIC` so that the *ids* generated by the *new-order* transaction no longer need to be sequential and could be generated locally by WeaQL. In this context, *new-order* transactions no longer required coordination. We deployed a five replicas cluster and executed multiple runs with different numbers of clients issuing operations. For this experiment we used a workload which consisted only of *new-order* transactions. We ran both configurations of the TPC-C benchmark (the standard one, and the one which does not require *ids* to be sequential), and compared the results.

Figure 5.6 presents the throughput and latency comparison between the executions of both configurations of TPC-C. As expected, the configuration of TPC-C that does not require coordination yields better performance and lower latencies than the original specification. For a load of five clients, the latency increase introduced by the use of the coordination service is roughly of 22%, from an average of 48.9ms to 59.7ms per transaction, while the throughput decreases by 18%, from 102 ops/s to 84 ops/s. At the maximum load of the experiment (275 clients), the *coordination-required* configuration

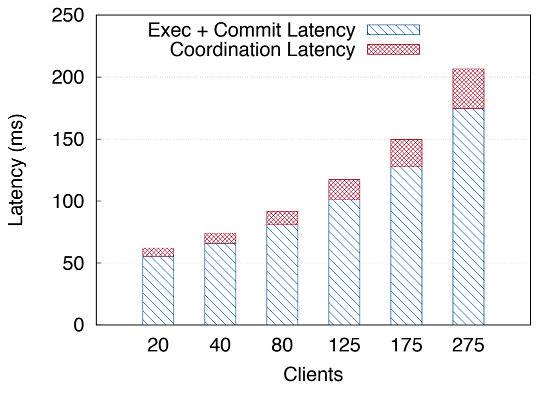


Figure 5.7: Experienced Latency Split in WeaQL

yielded 15% less throughput than the coordination-free counterpart, while experiencing an 18% increase in latency. Indeed, the performance variation between both experiments exhibited a small amplitude for the different loads injected in the system. In particular, the increase in latency observed by the coordination-required experiment fluctuated between 11% and 22% when compared to the coordination-free experiment, while the throughput varied between 82% and 91%.

Adding more clients to the system naturally results in an increase of the average latency of each transaction, which is clearly reflected in Figure 5.6b. The source of this behavior is two-fold. On the one hand, increasing the load in the system requires WeaQL to process more client's requests. On the other hand, it also increases the number of concurrent accesses that the coordination service must handle. To further investigate how each of these aspects contributed to the increase in latency (in the coordination-required experiment), we analyzed its origin in more detail. In particular, we split the latency of each transaction into two groups: *i*) execution and commit latency, and *ii*) coordination latency. This allowed us to accurately discern how much the coordination process contributed to the observed latency.

Figure 5.7 depicts this situation, where the total latency is distributed by the two distinct groups previously described. The results obtained reveal that it is the *execution + commit* time that most contribute to the increase of latency experienced by the clients. As an example, consider the results obtained for the experiment with 175 and 275 clients. The average latency with 175 clients was roughly 150ms, being 128ms associated with the *execution + commit* of the transactions, and 22ms from the coordination process. By analyzing the values obtained for the experiment with 275 clients, we observed an average of 206ms per transaction, 174ms relative to the *execution + commit* of the transaction and 32ms introduced by the use of the coordination process. These results show that the increase of the load in the system had a low impact on the time spent contacting the coordination service when compared to the time it took to execute and commit the operation. Indeed, in raw values, we notice that the coordination process only took an additional 10ms in the experiment with 275 clients, while the *execution + commit* experienced an additional of 46*ms* when compared to the experiment with 175 clients. To conclude, we can speculate that WeaQL will saturate faster then the coordination service, which means that the coordination service is not a significative bottleneck in our solution.

5.6 Summary

In this chapter, we have evaluated WeaQL against two popular solutions to replicate the state of relational databases among a set of replicas. We started by assessing the overhead imposed by WeaQL in Section 5.2. In particular, the results obtained have shown that WeaQL imposes a modest overhead when compared with Galera-Cluster, while the comparison with MySQL-Cluster was somehow inconclusive due to the fact that internal aspects of the database supporting this system appear to have significant overhead when multiple clients access the system. In Section 5.3 we have evaluated WeaQL in two distinct configurations, and under different system loads. We have shown that WeaQL is able to achieve much better performance in terms of peak throughput across all tested configurations, in relation to both of the used baselines. The benefits of lazy replication were also explored in Section 5.4. More specifically, we have shown that WeaQL is able to scale better than the competing alternatives (particularly under write-intensive workloads), by avoiding continuous synchronization among replicas. Finally, in Section 5.5, we have evaluated the costs of protecting particular SQL invariants by resorting to coordination, and shown that this process imposes acceptable overhead in terms of the latency experienced by the end user and overall throughout of the system.

Chapter 6

Conclusion

The fast improvement of the Internet's infrastructure around the globe enabled the emergence of Internet services which handle massive amounts of information and requests from clients. To meet the strict requirements of these applications (e.g., availability and performance), the backbone infrastructure which supports these services is replicated across multiple clusters. For decades, relational database systems had been the choice of excellence of applications to store and retrieve their data. The semantics provided by the SQL language established a rich, well-founded programming environment which allowed programmers to focus on the development of their applications while eschewing the need to handle low-level details such as concurrency accesses to their data or the atomic execution of a set of operations. However, scaling relational databases to meet the needs of this new era of applications has proved challenging, as relational databases were not devised to be explored in a distributed setting.

In this dissertation, we explored an alternative approach to improve the performance and scalability of relational databases. Motivated by the fact many applications already leverage weakly consistent replication for improved performance, we exploited this observation in the context of relational databases. To our knowledge, our work is the first to combine weakly consistent replication without enforcing a total order of transactions execution. We achieved this by ensuring that transactions are commutative by design, which allowed us to avoid upfront coordination, a mechanism so far required to ensure that replicas converged to the same final state. In addition, we also empowered our solution with lightweight mechanisms which preserve many SQL integrity constraints, which we believe to be fundamental to developers to express their application's semantics. We have materialized these ideas in WeaQL, a middleware replication layer that offers weakly consistent semantics on top of a set of replicated relational databases.

The benefits of our solution were demonstrated in Chapter 5. In more detail, Section 5.3 showed that WeaQL yielded better performance at lower latencies for a variety number of replicas and clients, when compared with two state-of-the-art solutions for replicating relational databases. In Section 5.4 we have shown that our solution is able to scale better than both baselines. As a trade-off, WeaQL sacrifices data consistency, enabling replicas to temporarily diverge, therefore allowing users to read stale data. In this context, we do not claim that our solution fits all application's scenarios. Indeed, applications that deal with sensible information and want to preserve strong consistency semantics must resort to solutions that enforce synchronous replication across all replicas.

Nevertheless, the work proposed in the context of this dissertation offers a new point in the design space of scalable database replication, where state-convergence is achieved by design and coordination is limited to the operations that endanger SQL invariants.

6.1 Limitations and Future Work

As a prototype, WeaQL currently misses some features that applications may require to fully implement their functionalities. For instance, WeaQL prototype does not allow primary-keys to be altered (i.e., they are immutable). This is because we use primary-keys to ensure that our SQL transformations always target the intended records. However, this is not a fundamental issue. We could extend WeaQL by adding a new immutable metadata column, as a way to always be able to deterministically refer to the records initially desired. Furthermore, since each *delete* operation does not effectively erase the record from the database, the dataset keeps growing in size. Our implementation does not yet handles garbage collection of deleted records, but one simple solution is to have a background thread periodically looking for deleted records and erasing them.

As future work, one promising path is to provide a formal proof that our SQL transformations effectively ensure state-convergence. While we spent much time working on these transformations and testing all possible interleaving of operations, a formal proof would constitute a great contribution to the research community and could inspire the emergence of new CRDTs, with practical use in the context of relational databases. Furthermore, it would be interesting to explore a more dynamic behavior in the replication process employed by WeaQL. In more detail, our solution currently propagates transaction's logs to remote replicas right after a transaction is successfully committed on the local replica. We could change this behavior by propagating batches of transactions logs instead of one at a time, to extract even more performance of the system at the cost of increasing the staleness of data. In contrast, we could also augment WeaQL to enforce that the number of pending transactions to be applied from remote replicas did not exceed a certain threshold, by temporarily refusing to process requests from clients.

Bibliography

[Agu+15] M. K. Aguilera, J. B. Leners, and M. Walfish. “Yesquel: Scalable Sql Storage for Web Applications”. In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP ’15. Monterey, California: ACM, 2015, pp. 245–262. isbn: 978-1-4503-3834-9. doi: 10.1145/2815400.2815413. url: http://doi.acm.org/10.1145/2815400.2815413.

[Alm+13] S. Almeida, J. a. Leitão, and L. Rodrigues. “ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication”. In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys ’13. New York, NY, USA: ACM, 2013, pp. 85–98. isbn: 978-1-4503-1994-2. doi: 10.1145/2465351.2465361. url: http://doi.acm.org/10.1145/2465351.2465361.

[Amz+03] C. Amza, A. L. Cox, and W. Zwaenepoel. “Conflict-aware Scheduling for Dynamic Content Applications”. In: *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*. Berkeley, CA, USA: USENIX Association, 2003, pp. 6–6.

[Ava16] H. Availability. *How League Of Legends Scaled Chat To 70 Million Players*. Mar. 2016. url: http://highscalability.com/blog/2014/10/13/how-league-of-legends-scaled-chat-to-70-million-players-it-t.html.

[Bai15] P. Bailis. *Linearizability versus Serializability*. Feb. 2015. url: http://www.bailis.org/blog/linearizability-versus-serializability/.

[BG13] P. Bailis and A. Ghodsi. “Eventual Consistency Today: Limitations, Extensions, and Beyond”. In: *Queue* 11.3 (Mar. 2013), 20:20–20:32. issn: 1542-7730. doi: 10.1145/2460276.2462076. url: http://doi.acm.org/10.1145/2460276.2462076.

[Bai+13] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. “HAT, Not CAP: Towards Highly Available Transactions”. In: *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*. Berkeley, CA, USA: USENIX Association, 2013, pp. 24–24.

[Bai+14] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. “Coordination-Avoidance in Database Systems”. In: *CoRR* abs/1402.2237 (2014).

[Bal+15a] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro. "Putting Consistency Back into Eventual Consistency". In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. Bordeaux, France: ACM, 2015, 6:1–6:16. ISBN: 978-1-4503-3238-5. DOI: 10.1145/2741948.2741972. URL: http://doi.acm.org/10.1145/2741948.2741972.

[Bal+15b] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro. "Towards Fast Invariant Preservation in Geo-replicated Systems". In: *SIGOPS Oper. Syst. Rev.* 49.1 (Jan. 2015), pp. 121–125. ISSN: 0163-5980. DOI: 10.1145/2723872.2723889. URL: http://doi.acm.org/10.1145/2723872.2723889.

[BT11] D. Bermbach and S. Tai. "Eventual Consistency: How Soon is Eventual? An Evaluation of Amazon S3's Consistency Behavior". In: *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*. MW4SOC '11. Lisbon, Portugal: ACM, 2011, 1:1–1:6. ISBN: 978-1-4503-1067-3. DOI: 10.1145/2093185.2093186. URL: http://doi.acm.org/10.1145/2093185.2093186.

[BG81] P. A. Bernstein and N. Goodman. "Concurrency Control in Distributed Database Systems". In: *ACM Comput. Surv.* 13.2 (June 1981), pp. 185–221. ISSN: 0360-0300. DOI: 10.1145/356842.356846. URL: http://doi.acm.org/10.1145/356842.356846.

[Ber+86] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986. ISBN: 0-201-10715-5.

[Bor+11] M. A. Bornea, O. Hodson, S. Elnikety, and A. Fekete. "One-copy Serializability with Snapshot Isolation Under the Hood". In: *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 625–636.

[Cat11] R. Cattell. "Scalable SQL and NoSQL Data Stores". In: *SIGMOD Rec.* 39.4 (May 2011), pp. 12–27. ISSN: 0163-5808. DOI: 10.1145/1978915.1978919. URL: http://doi.acm.org/10.1145/1978915.1978919.

[Cec+08] E. Cecchet, G. Candea, and A. Ailamaki. "Middleware-based Database Replication: The Gaps Between Theory and Practice". In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. ACM, 2008, pp. 739–752.

[Cha+14] P. Chairunnanda, K. Daudjee, and M. T. Özsu. "ConfluxDB: Multi-Master Replication for Partitioned Snapshot Isolation Databases". In: *PVLDB* 7.11 (2014), pp. 947–958.

[Cha+08] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. "Bigtable: A Distributed Storage System for Structured Data". In: *ACM Transactions on Computer Systems* 26.2 (June 2008), pp. 4–26.

[Cle+13] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. "The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. New York, NY, USA: ACM, 2013, pp. 1–17.

[Coo+08] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. "PNUTS: Yahoo!'s Hosted Data Serving Platform". In: *Proceedings of the VLDB Endowment* 1.2 (Aug. 2008), pp. 1277–1288.

[Cor+12] J. C. Corbett et al. "Spanner: Google's Globally-distributed Database". In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2012, pp. 251–264.

[Cou15] T. P. P. Council. *TPC-C Benchmark*. Feb. 2015. URL: http://www.tpc.org/tpcc/.

[DS06] K. Daudjee and K. Salem. "Lazy Database Replication with Snapshot Isolation". In: *Proceedings of the 32th International Conference on Very Large Data Bases*. VLDB Endowment, 2006, pp. 715–726.

[DeC+07] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: Amazon's Highly Available Key-value Store". In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. ACM, 2007, pp. 205–220.

[Dis+15] T. Distler, C. Bahn, A. Bessani, F. Fischer, and F. Junqueira. "Extensible Distributed Coordination". In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. Bordeaux, France: ACM, 2015, 10:1–10:16. ISBN: 978-1-4503-3238-5. DOI: 10.1145/2741948.2741954. URL: http://doi.acm.org/10.1145/2741948.2741954.

[Dro15] Dropbox. *Dropbox*. Jan. 2015. URL: https://www.dropbox.com/developers.

[EL14] M. M. Elbushra and J. Lindström. "Eventual Consistent Databases: State of the Art". In: *Open Journal of Databases (OJDB)* 1.1 (2014), pp. 26–41. ISSN: 2199-3459. URL: http://www.ronpub.com/publications/OJDB-v1i1n03_Elbushra.pdf.

[Fac13] Q. Facebook Ericsson. *A Focus on Efficiency*. Tech. rep. 2013.

[Fit04] B. Fitzpatrick. “Distributed Caching with Memcached”. In: *Linux J.* 2004.124 (Aug. 2004), pp. 5–. ISSN: 1075-3583. URL: http://dl.acm.org/citation.cfm?id=1012889.1012894.

[Gal15] Galera. *Galera Cluster*. June 2015. URL: http://galeracluster.com/.

[GL02] S. Gilbert and N. Lynch. “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services”. In: *SIGACT News* 33.2 (June 2002), pp. 51–59.

[Gra+96] J. Gray, P. Helland, P. O’Neil, and D. Shasha. “The Dangers of Replication and a Solution”. In: *SIGMOD Record* 25.2 (June 1996), pp. 173–182.

[Hal15] C. Hale. *You Can’t Sacrifice Partition Tolerance*. Feb. 2015. URL: http://codahale.com/you-cant-sacrifice-partition-tolerance/.

[HM14] A. Hammadi and L. Mhamdi. “Review: A Survey on Architectures and Energy Efficiency in Data Center Networks”. In: *Comput. Commun.* 40 (Mar. 2014), pp. 1–21. ISSN: 0140-3664. DOI: 10.1016/j.comcom.2013.11.005. URL: http://dx.doi.org/10.1016/j.comcom.2013.11.005.

[HW90] M. P. Herlihy and J. M. Wing. “Linearizability: A Correctness Condition for Concurrent Objects”. In: *ACM Trans. Program. Lang. Syst.* 12.3 (July 1990), pp. 463–492. ISSN: 0164-0925. DOI: 10.1145/78969.78972. URL: http://doi.acm.org/10.1145/78969.78972.

[Hun+10] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. “ZooKeeper: Wait-free Coordination for Internet-scale Systems”. In: *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*. USENIXATC’10. Boston, MA: USENIX Association, 2010, pp. 11–11. URL: http://dl.acm.org/citation.cfm?id=1855840.1855851.

[Inc15] G. Inc. *Google Drive*. Feb. 2015. URL: https://developers.google.com/drive/.

[KA00] B. Kemme and G. Alonso. “A New Approach to Developing and Implementing Eager Database Replication Protocols”. In: *ACM Trans. Database Syst.* 25.3 (Sept. 2000), pp. 333–379.

[Kra+09] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. “Consistency Rationing in the Cloud: Pay Only when It Matters”. In: *Proceedings of the VLDB Endowment* 2.1 (Aug. 2009), pp. 253–264.

[Lad+92] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. “Providing High Availability Using Lazy Replication”. In: *ACM Trans. Comput. Syst.* 10.4 (Nov. 1992), pp. 360–391. ISSN: 0734-2071. DOI: 10.1145/138873.138877. URL: http://doi.acm.org/10.1145/138873.138877.

[LM10] A. Lakshman and P. Malik. “Cassandra: A Decentralized Structured Storage System”. In: *SIGOPS Oper. Syst. Rev.* 44.2 (Apr. 2010), pp. 35–40.

[Lam78] L. Lamport. "Time, clocks, and the ordering of events in a distributed system". In: *Communications of the ACM* 21.7 (July 1978), pp. 558–565.

[Lam06] L. Lamport. "Fast Paxos". In: *Distributed Computing* 19.2 (2006), pp. 79–103. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=64624.

[Li+12] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. "Making Geo-replicated Systems Fast As Possible, Consistent when Necessary". In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI'12. USENIX Association, 2012, pp. 265–278. ISBN: 978-1-931971-96-6. URL: http://dl.acm.org/citation.cfm?id=2387880.2387906.

[Li+14] C. Li, J. a. Leitão, A. Clement, N. Preguiça, R. Rodrigues, and V. Vafeiadis. "Automating the Choice of Consistency Levels in Replicated Systems". In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. USENIX ATC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 281–292. ISBN: 978-1-931971-10-2. URL: http://dl.acm.org/citation.cfm?id=2643634.2643664.

[Lin+05] Y. Lin, B. Kemme, M. Pati n o Mart ’ınez, and R. Jiménez-Peris. "Middleware Based Data Replication Providing Snapshot Isolation". In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2005, pp. 419–430.

[Llo+11] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. "Don'T Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. New York, NY, USA: ACM, 2011, pp. 401–416. ISBN: 978-1-4503-0977-6. DOI: 10.1145/2043556.2043593. URL: http://doi.acm.org/10.1145/2043556.2043593.

[Llo+13] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. "Stronger Semantics for Low-Latency Geo-Replicated Storage". In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. 2013, pp. 313–328.

[Mah+11] P. Mahajan, L. Alvisi, and M. Dahlin. *Consistency, Availability, Convergence*. Tech. rep. TR-11-22. Computer Science Department, University of Texas at Austin, May 2011.

[Mon] MongoDB. *MongoDB*. https://www.mongodb.org/.

[MyS15a] MySQL. *MySQL 5.6 Documentation*. Jan. 2015. URL: http://dev.mysql.com/doc/refman/5.6/en/index.html.

[MyS15b] MySQL. *MySQL Cluster*. June 2015. URL: https://www.mysql.com/products/cluster/.

[MyS16a] MySQL. *NDB Storage Engine*. Mar. 2016. URL: http://dev.mysql.com/doc/refman/5.7/en/mysql-cluster.html.

[MyS16b] MySQL. *The InnoDB Storage Engine*. Mar. 2016. URL: http://dev.mysql.com/doc/refman/5.7/en/innodb-storage-engine.html.

[Ora15] Oracle. *Oracle Database 11g*. Jan. 2015. URL: https://docs.oracle.com/cd/E11882_01/nav/portal_4.htm.

[Ora16] Oracle. *JDBC Statement Interface*. Mar. 2016. URL: https://docs.oracle.com/javase/7/docs/api/java/sql/Statement.html#addBatch(java.lang.String).

[Ora] *Oracle Unveils Big Memory Machines With New SPARC M6 Processors for In-Memory Applications*. http://www.oracle.com/us/corporate/features/sparc-m6/index.html. accessed Nov,2015.

[Pim] S. Pimentel. *The Return of ACID in the Design of NoSQL Databases*. http://www.methodsandtools.com/archive/acidnosqldatabase.php.

[PA04] C. Plattner and G. Alonso. “Ganymed: Scalable Replication for Transactional Web Applications”. In: *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*. New York, NY, USA: Springer-Verlag New York, Inc., 2004, pp. 155–174.

[Pla+08] C. Plattner, G. Alonso, and M. T. O zsu. “Extending DBMSs with Satellite Databases”. In: *The VLDB Journal* 17.4 (July 2008), pp. 657–682.

[Pos15] PostgreSQL. *PostgreSQL 9.2 Documentation*. Jan. 2015. URL: postgresql.org/docs/9.2/static/.

[Puj+10] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez. “The Little Engine(s) That Could: Scaling Online Social Networks”. In: *Proceedings of the ACM SIGCOMM 2010 Conference*. SIGCOMM ’10. New Delhi, India: ACM, 2010, pp. 375–386. ISBN: 978-1-4503-0201-2. DOI: 10.1145/1851182.1851227. URL: http://doi.acm.org/10.1145/1851182.1851227.

[Ren+08] R. van Renesse, D. Dumitriu, V. Gough, and C. Thomas. “Efficient Reconciliation and Flow Control for Anti-entropy Protocols”. In: *Proceedings of the 2Nd Workshop on Large-Scale Distributed Systems and Middleware*. New York, NY, USA: ACM, 2008, 6:1–6:7.

[SS05] Y. Saito and M. Shapiro. “Optimistic Replication”. In: *ACM Computing Surveys* 37.1 (Mar. 2005), pp. 42–81.

[Sal+11] T.-I. Salomie, I. E. Subasu, J. Giceva, and G. Alonso. “Database engines on multicores, why parallelize when you can distribute?” In: *Proceedings of the Sixth Conference on Computer Systems*. ACM Press, 2011, pp. 17–30.

[Sch90] F. B. Schneider. “Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial”. In: *ACM Comput. Surv.* 22.4 (Dec. 1990), pp. 299–319. ISSN: 0360-0300. DOI: 10.1145/98163.98167. URL: http://doi.acm.org/10.1145/98163.98167.

[SB09] E. Schurman and J. Brutlag. *Performance Related Changes and their User Impact. Presented at Velocity Web Performance and Operations Conference*. http://slideplayer.com/slide/1402419/. 2009.

[Sha+11a] M. Shapiro, N. Preguica, C. Baquero, and M. Zawirski. *A comprehensive study of convergent and commutative replicated data types*. Tech. rep. RR-7506. INRIA, 2011.

[Sha+11b] M. Shapiro, N. Preguica, C. Baquero, and M. Zawirski. “Conflict-free Replicated Data Types”. In: *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*. 2011, pp. 386–400.

[Shu+13] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. “F1: A Distributed SQL Database That Scales”. In: *VLDB*. 2013.

[Gor] “State of the Art in Database Replication”. In: (Jan. 2007), pp. 1–64.

[Tec15] B. Technologies. *Riak*. Jan. 2015. URL: http://basho.com/riak/.

[Ter+95] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. “Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System”. In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. New York, NY, USA: ACM, 1995, pp. 172–182.

[Ter13] D. Terry. “Replicated Data Consistency Explained Through Baseball”. In: *Commun. ACM* 56.12 (Dec. 2013), pp. 82–89. ISSN: 0001-0782. DOI: 10.1145/2500500. URL: http://doi.acm.org/10.1145/2500500.

[Ter+13] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. “Consistency-based service level agreements for cloud storage”. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM Press, 2013, pp. 309–324.

[Vog08] W. Vogels. “Eventually Consistent”. In: *Queue* 6.6 (Oct. 2008), pp. 14–19. ISSN: 1542-7730. DOI: 10.1145/1466443.1466448. URL: http://doi.acm.org/10.1145/1466443.1466448.

[Vog09] W. Vogels. “Eventually consistent”. In: *Commun. ACM* 52.1 (2009), pp. 40–44. DOI: 10.1145/1435417.1435432. URL: http://doi.acm.org/10.1145/1435417.1435432.

[Gom] *Why Web Performance Matters : Is Your Site Driving Customers Away ?* http://www.mcrinc.com/Documents/Newsletters/201110_why_web_performance_matters.pdf. 2011.

[Wie+00] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. “Understanding Replication in Databases and Distributed Systems”. In: 2000, pp. 264–274.