Transport-Level Transactions: Simple Consistency for Complex Scalable Low-Latency Cloud Applications

by

Mahdi Tayarani Najaran

M. Sc., Computer Science, The University of British Columbia, 2009
M. Sc., Electrical Engineering, Sharif University of Technology, 2007
B. Sc., Electrical Engineering, Ferdowsi University of Mashhad, 2005

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Computer Science)

The University of British Columbia

(Vancouver)

August 2015

© Mahdi Tayarani Najaran, 2015
Abstract

The classical move from single-server applications to scalable cloud services is to split the application state along certain dimensions into smaller partitions independently absorbable by a separate server in terms of size and load. Maintaining data consistency in the face of operations that cross partition boundaries imposes unwanted complexity on the application. While for most applications many ideal partitioning schemes readily exist, First-Person Shooter (FPS) games and Relational Database Management Systems (RDBMS) are instances of applications whose state can’t be trivially partitioned. For any partitioning scheme there exists an FPS/RDBMS workload that results in frequent cross-partition operations.

In this thesis we propose that it is possible and effective to provide unpartitionable applications with a generic communication infrastructure that enforces strong consistency of the application’s data to simplify cross-partition communications. Using this framework the application can use a sub-optimal partitioning mechanism without having to worry about crossing boundaries. We apply our thesis to take a head-on approach at scaling our target applications. We build three scalable systems with competitive performances, used for storing data in a key/value data-store, scaling fast-paced FPS games to epic sized battles consisting of hundreds of players, and a scalable full-SQL compliant database used for storing tens of millions of items.
Preface

This dissertation is based on the following publications:


- P4. Mahdi Tayarani Najaran, Primal Wijesekera, Norman C. Hutchinson, Andrew Warfield, Distributed Locking and Indexing: In Search for Scalable Consistency, 5th Workshop on Large-Scale Distributed Systems and Middleware (Ladis), 2011 [119].

- P5. Mahdi Tayarani Najaran, Charles Krasic, Scaling Online Games with Adaptive Interest Management in the Cloud, 9th Workshop on Network and Systems Supports for Games (NetGames), 2010 [104].

Most of the work in the papers is mine as the lead investigator with my advisor, Norman C. Hutchinson, as a co-author. Shun-Yun Hu was a co-author on P1 which helped on writing and the preparation of the paper, and also helped define evaluation requirements and related work. P4 was done in collaboration with Primal
Wijesekera and his advisor Andrew Warfield. Primal helped with the implementation of the MySQL handler, distributed locking and evaluation. P5 was done with my previous advisor, Charles Krasic, as a co-author.

Some of the writing of the thesis was taken from the aforementioned papers. Specifically, Chapter 3 draws from P1, P2 & P3, Chapter 4 from P2 & P3, Chapter 5 from P1 and Chapter 6 from P4. Additionally, Chapter 6 uses text and evaluation results from the Primal’s thesis [125], from whom permission has been obtained.
# Table of Contents

Abstract .................................................. ii

Preface .................................................... iii

Table of Contents ........................................ v

List of Tables ............................................. x

List of Figures ............................................ xi

Acknowledgements ......................................... xiv

1 Introduction ............................................. 1

1.1 Online Games at Scale ................................. 1

1.2 Classical Cloud-ification ............................. 2

1.3 Partitioning the Unpartitionable ....................... 3

1.4 The Thesis ........................................... 3

1.5 CAP Revisited ......................................... 5

1.6 Contributions .......................................... 6

1.6.1 One-Round Transactions ............................ 6

1.6.2 Key/Value Store ..................................... 6

1.6.3 Scalable Spatial Publish/Subscribe .................. 7

1.6.4 Partitioned SQL ..................................... 7

1.7 Roadmap ................................................ 8
# Background & Motivation

2.1 Consistency

2.1.1 Why Strong Consistency?

2.1.2 ACID

2.2 Transport-level ACID

2.2.1 One-Round Transactions

2.2.2 Cloud Unfriendly Applications: A Different Approach

# One-Round Transactions

3.1 API

3.1.1 Client

3.1.2 Server

3.1.3 Sample Transaction

3.2 Commit Protocol

3.2.1 Lock Mode

3.2.2 Clock Mode

3.2.3 Lock vs. Clock

3.3 Durability & Failure Recovery

# Innesto

4.1 Introduction

4.2 Overview

4.2.1 Data Model & API

4.2.2 Architecture

4.2.3 Strong Consistency

4.3 Multi-Version Storage (MVS)

4.3.1 Partitioning

4.3.2 Non Structure-Modifying Operations

4.3.3 Structure-Modifying Operations

4.3.4 Partition Hierarchy

4.4 Search

4.4.1 Search Clones

4.4.2 Sparse Clones
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5</td>
<td>Extended API</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Operation Transactions</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Conditional Operations</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Extensions</td>
</tr>
<tr>
<td>4.6</td>
<td>Evaluation</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Key/Value Performance</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Load Balancing</td>
</tr>
<tr>
<td>4.6.3</td>
<td>Clone Cost</td>
</tr>
<tr>
<td>4.6.4</td>
<td>High Dimensionality</td>
</tr>
<tr>
<td>4.6.5</td>
<td>Lock vs. Clock</td>
</tr>
<tr>
<td>4.6.6</td>
<td>Summary</td>
</tr>
<tr>
<td>4.7</td>
<td>Related Work</td>
</tr>
<tr>
<td>4.7.1</td>
<td>Cloud Storage &amp; Key/Value Datastores</td>
</tr>
<tr>
<td>4.7.2</td>
<td>Transactional Datastores</td>
</tr>
<tr>
<td>4.7.3</td>
<td>Peer-to-Peer (P2P)</td>
</tr>
<tr>
<td>4.7.4</td>
<td>Distributed Transactions</td>
</tr>
<tr>
<td>4.8</td>
<td>Conclusion</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>SPEX</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>5.2</td>
<td>Virtual Environments</td>
</tr>
<tr>
<td>5.3</td>
<td>Overview</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Dynamic VE Distribution</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Consistency &amp; Fault Tolerance</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Adaptive Load Balancing</td>
</tr>
<tr>
<td>5.4</td>
<td>SPS Functionality</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Spatial Publications</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Spatial Subscriptions</td>
</tr>
<tr>
<td>5.5</td>
<td>Implementation Details</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Spatial Partitioning</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Scalable Pub/Sub</td>
</tr>
<tr>
<td>5.6</td>
<td>Discussion</td>
</tr>
<tr>
<td>5.7</td>
<td>Evaluation</td>
</tr>
</tbody>
</table>

vii
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.6.3 Transaction Handler Failure</td>
<td>106</td>
</tr>
<tr>
<td>6.7 Evaluation</td>
<td>107</td>
</tr>
<tr>
<td>6.7.1 TPC-C</td>
<td>108</td>
</tr>
<tr>
<td>6.7.2 Strong Scaling</td>
<td>109</td>
</tr>
<tr>
<td>6.7.3 Contention</td>
<td>110</td>
</tr>
<tr>
<td>6.7.4 Web Compatible Workload</td>
<td>112</td>
</tr>
<tr>
<td>6.7.5 Crash Consistency Cost</td>
<td>113</td>
</tr>
<tr>
<td>6.8 Related Work</td>
<td>114</td>
</tr>
<tr>
<td>6.8.1 Data Partitioning</td>
<td>114</td>
</tr>
<tr>
<td>6.8.2 Distributed Indexing</td>
<td>115</td>
</tr>
<tr>
<td>6.8.3 Distributed Locking</td>
<td>116</td>
</tr>
<tr>
<td>6.8.4 NoSQL</td>
<td>116</td>
</tr>
<tr>
<td>6.8.5 Scaling Storage</td>
<td>117</td>
</tr>
<tr>
<td>6.8.6 NewSQL</td>
<td>118</td>
</tr>
<tr>
<td>6.8.7 Scalable Transactional Models</td>
<td>118</td>
</tr>
<tr>
<td>6.9 Conclusion &amp; Future Work</td>
<td>119</td>
</tr>
<tr>
<td>7 Future Work: Argal</td>
<td>120</td>
</tr>
<tr>
<td>8 Conclusion</td>
<td>124</td>
</tr>
<tr>
<td>Bibliography</td>
<td>126</td>
</tr>
</tbody>
</table>
List of Tables

Table 2.1 Possible anomalies of inconsistent systems [110]. . . . . . . . 10
Table 2.2 Order of operations $a$, $b$ and $c$ observable by every thread under different consistency models (assuming $b$ causally depends on $a$). 10
Table 3.1 Description of message fields used in the commit protocol. . . 25
Table 4.1 Innesto’s API which includes basic key/value operations plus search. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
Table 4.2 Innesto’s latency and storage requirements for storing $N$ items with different configurations. . . . . . . . . . . . . . . . . . . 41
Table 4.3 YCSB workload specification in terms of operation mixtures and access distribution. By default, each item has ten attributes with a total size of 1KB. . . . . . . . . . . . . . . . . . . . . . . . . 43
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Transaction commit protocol.</td>
<td>24</td>
</tr>
<tr>
<td>4.1</td>
<td>Logical architecture of Innesto components. Each component is fully distributed.</td>
<td>33</td>
</tr>
<tr>
<td>4.2</td>
<td>Spatial partitioning and distribution of a table. The root partition covers the entire key space and each partition keeps references to child/parent partitions. Only leaf (gray) partitions store the actual data. Partitions are distributed between servers for load balancing.</td>
<td>35</td>
</tr>
<tr>
<td>4.3</td>
<td>Search Clones are used to partition data based on secondary attributes. Items are grouped into attribute groups to better support range search. Each $K_i$ is a full copy of an item.</td>
<td>38</td>
</tr>
<tr>
<td>4.4</td>
<td>Average throughput for YCSB benchmark. The first three (red) workloads are write heavy, the middle three (gray) are read heavy, and the right most workload (black) is search heavy.</td>
<td>44</td>
</tr>
<tr>
<td>4.5</td>
<td>Temporal throughput in Load phase of the YCSB benchmark. Notice how HyperDex’s performance decreases with dataset size while Cassandra and Innesto maintain a steady throughput.</td>
<td>46</td>
</tr>
<tr>
<td>4.6</td>
<td>Average Innesto throughput for YCSB benchmark with different configurations.</td>
<td>47</td>
</tr>
<tr>
<td>4.7</td>
<td>Average throughput for write-only workload of inserting 5 million items with varying number of data dimensionality.</td>
<td>48</td>
</tr>
</tbody>
</table>
Figure 4.8  Average throughput for YCSB benchmark for Innesto in lock mode and clock mode. Compared to lock mode, clock mode has on average 31.8% lower throughput.  

Figure 4.9  Average throughput of Innesto in lock mode and clock mode using the hotspot distribution. The x-axis shows the probability of accessing a hot item. 

Figure 4.10  Average throughput of Innesto in lock mode and clock mode using the hotspot distribution with a fixed access probability of 95% with varying number of hot items. 

Figure 5.1  Basic architecture of SPEX. All communications are done using one-round transactions. 

Figure 5.2  (left) Full partitioning vs. (right) Progressive partitioning. 

Figure 5.3  Distributed dynamic partition hierarchy. 

Figure 5.4  99th percentile SPEX processing delays in the testbed. 

Figure 5.5  99th percentile SPEX processing delay in the testbed with different update rates at map size $1024 \times 1024$. 

Figure 5.6  SPEX host and actor average bandwidth in the testbed. 

Figure 5.7  SPEX accuracy in the testbed. 

Figure 5.8  End-to-end delay across the continent in EC2. 

Figure 5.9  SPEX performance over period of 1 hour of run time with random waypoints. 

Figure 5.10  SPEX performance over 1 hour of run time with VE size of 256x256 and different numbers of hotspots. 

Figure 5.11  Cost of using one-round transactions for global synchronization instead of simple message passing. 

Figure 6.1  Conventional workflow in a RDBMS. 

Figure 6.2  General architecture of iEngine. 

Figure 6.3  Distributed storage layer architecture. 

Figure 6.4  MySQL architecture.
| Figure 6.5 | Range lock example. The numbers represent the reference count of each fragment. Note that the partitions and locks could have any number of dimensions. | 102 |
| Figure 6.6 | Red are existing range locks and gray are pending range lock requests (left) Order in which range lock requests arrive (right) Equivalent lock dependency graph. | 104 |
| Figure 6.7 | Strong scaling of iEngine. | 109 |
| Figure 6.8 | User contention. | 111 |
| Figure 6.9 | Web compatible workload. | 112 |
| Figure 6.10 | Crash consistency. | 113 |
| Figure 7.1 | Architecture of Argal. | 122 |
I would like to thank my advisor Dr. Norman C. Hutchinson for his support and guidance throughout the course of developing this thesis. His feedback and careful eye for detail proved very helpful.

Special thanks to my wife Sara for her support and motivation, without which I might have never reached the end.

Thanks to all collaborators, project members and NSS friends.
Chapter 1

Introduction

1.1 Online Games at Scale

Online games are a billion dollar industry which provide experiences ranging from simple pass-time puzzles to million-dollar professional tournaments [16]. Depending on the genre of game, scaling online games has different interpretations. Player vs. Environment (PvE) games focus on the interactions between the player and the game in the form of puzzles, tasks or quests. Players can only interact with each other asynchronously at the meta-game level.\(^1\) Player vs. Player (PvP) games on the other hand allow synchronous interactions between multiple players.\(^2\) Each game instance is a head-to-head challenge between a fixed number of players. In both types of PvE and PvP the service host acts as a communications medium between the players, match making, and storing state. Scaling such games entails supporting more instances of the game to achieve higher online user counts.

First-person shooter (FPS) games are a popular form of PvP where each player observes a virtual world from the eyes of an avatar while interacting with other players in real-time. FPS games have the potential to fulfill the excitement and thrill of participating in epic battles, ones that we can currently only observe in

---

\(^1\)Farmville [18] is a game where players individually build their own farm and can send/receive gifts and favors from their friends on social networks.

\(^2\)Clash of Clans [12] is a fort-building game where players can attack another player’s fort or have to defend themselves from attacks. Dota2 [15] is a team-based game where each fixed-size team of players has to destroy the other team’s stronghold.
movies such as Avatar, Lord of the Rings and Star Wars. Scaling FPS games to allow more players to participate in the same game instance has become an active area of research.

1.2 Classical Cloud-ification

In the era of cloud computing, the cloud has emerged as the dominant solution for large-scale on-line services and big data applications. With scalability in mind, the cloud has presented an appealing hosting environment for pay-as-you-go, always-available on-line services. The first version of our work found commercial virtualized cloud environments (Amazon EC2) [3] to be suitable for hosting FPS games at such scale with a very appealing model of gaming-as-a-service [104].

The classical move from single-server applications to the cloud is to split the application state along certain dimensions into smaller partitions absorbable by each cloud server in terms of size and load. Each partition is independently managed by a single server. As long as each operation can be contained in a single partition, data consistency may be enforced by its owner and race conditions be safely resolved. Thus the key feature of the partitioning criteria are being ones that result in the least cross-partition operations. Enforcing consistency on cross-partition operations requires following explicit protocols between the servers and adds unwanted complexity to the application.

For most applications many of such criteria readily exist. Unfortunately, FPS games are not like most applications. Players have a virtual position inside the game but the range of effect they can cause is arbitrarily defined by the weapon they carry. Thus for any partitioning criteria there exists a weapon that results in cross-partition operations. Additionally, unlike video data which is encoded once and served to users without modification, the state of the game is frequently and regularly modified by the players. Compared to social media applications, FPS games have a much higher write/read ratio in their workload. Players need to observe an almost consistent view of the game over wide area network (WAN) latencies which start to diverge as latency increases [67], losing the game appeal. The interactions between the players is fast and unpredictable and the system cannot know a-priori what a player will do next. Since the game is essentially a graphical
representation of state, data inconsistencies translate into bizarre visual artifacts and can’t be silently swept under the carpet.

1.3 Partitioning the Unpartitionable

Without an optimal partitioning scheme, an epic-scale FPS game is faced with one of two options. First, partition the players between the servers and limit them to their current partition. This will eliminate the need for cross-partition operations, but the limited feature set is visible to the players and reduces the quality of the virtual experience. Players can only see a subset of other players and interactions are only possible with objects and players in the current partition.

The second option is to partition the state to provide a rich virtual experience and deal with frequent cross-partition operations and constantly changing partitions. Related work in this area have successfully achieved scalability via the first option (e.g., [33]) despite its limitations. Currently, any other attempts to circumvent interaction limitations by using the second option have resulted in relaxing data consistency down to eventual-consistency. Considering how players invest a significant amount of time and money in a game, we consider eventual not good enough. Inconsistencies can directly be mapped to monetary losses. In this thesis we take the second option but aim to provide strong consistency.

1.4 The Thesis

An application node is a processing unit in charge of a partition of the application state which could be anything from a single virtualized core to a large multi-processor computing cluster. No matter what the application nodes are, there exists a scale at which a single node is no longer sufficient and multiple nodes are required. It is up to the application to now deal with sharing state across partitions in a consistent manner. The nodes need to be aware of the existence of others when performing their computations and need to somehow communicate their events.

In distributed systems the transport layer serves as the lowest level in the application stack. Applications use different forms of communications between different components in the form of remote procedure calls (RPC). An RPC may use standard requests, e.g., HTTP, REST [79], SOAP [37], or proprietary protocols,
e.g., RTMP [32], Thrift [47], protocol buffers [121]. Regardless, these are designed for point-to-point communications and provide no means to deal with one-to-many communications or synchronization. As a result, higher layers in the application stack are forced to deal with complicated data consistency scenarios when the application logic mandates one-to-many communications or there is concurrency on the data.

This thesis proposes that it is possible and effective to provide unpartitionable applications with a generic communication infrastructure that enforces strong consistency of the application’s data to simplify cross-partition communications. Our approach to providing data consistency is to replace the normal transport with one that explicitly deals with data consistency. Instead of a simple message passing API, the unit of communication is the one-round transaction. Our transactions are one-round since they provide the same abstraction as one round of message passing from source to destination, and they are a form of transaction as they allow atomically grouping and delivering messages to multiple recipients, always keeping the data in a consistent state. All subsequent components of the application are then built in this framework while having the freedom to fallback to normal message passing when consistency is not required. Using this framework the application can use a sub-optimal partitioning mechanism without having to worry about crossing boundaries. This removes an excessive amount of complexity from the end application which no longer has to deal with enforcing isolation, concurrency, and fault tolerance and recovery.

One-round transactions provide a simple programming model which transforms the non-trivial task of protocol design into a much simpler task of data structure design [49]. In essence, each message to be passed to other nodes only needs to consider what parts of the application’s state it is touching and what parts of the state were used to compute the message. This single guideline is generic as it does not require knowledge of other components which may be carrying out conflicting operations. It eliminates the need for the programmer to enumerate all possible permutations of actions on a message type basis.

Frameworks for consistent distributed objects date back to as long as the ISIS system [58]. Yet a key factor to consider in the applicability of a framework is the amount of effort required to migrate existing complex applications to it. Some
well-known single-node applications such as MySQL \[28\] cannot be trivially converted to adopt a different design architecture.

Modifying an existing application to use the one-round transaction framework is as simple as changing the transport layer without touching the rest of the application. One-round transactions will merely operate as the original transport would've done with no change to network performance complexity. Using one-round transactions for isolation and consistency requires some extra help from the application to interpret the contents of each message and adds at most an extra network round-trip time to each message. Providing this information is a simpler task than having to re-architect the application’s data model. The performance overhead is also predictably bound. Hence, we consider migrating to one-round transactions to be simpler than using other frameworks. Our framework has allowed us to take a direct approach at building systems which were already too complicated to try and deal with consistency and scalability at the same time.

1.5 CAP Revisited

A system simultaneously seeking scalability and consistency immediately brings the CAP theorem \[84\] to mind. At a first glance, the CAP theorem says distributed applications must choose two out of: consistency (C), availability (A), and the ability to tolerate network partitions (P). This rationale has been used to justify the decision to rank consistency last, making relaxing data consistency down to eventual-consistency in favor of availability a generally acceptable approach.

A deeper look at the CAP theorem shows this trade-off to be greatly misunderstood \[9\]. Partitions in the network are unavoidable, but rarely happen. Also, availability is deemed to be binary, yet in reality there is a spread of availability from all to nothing. The key insight here is that the CAP theorem holds in the micro scale, but taking a system as a whole, there doesn’t have to be a complete loss of availability at macro scale in the face of partitions. Thus a system may be consistent with some form of availability at all times, even during partitions \[9\]. CAP doesn’t contradict our design goals and we deem this model of operation to be a reasonable trade-off for our framework.
1.6 Contributions

1.6.1 One-Round Transactions

Using one-round transactions to build consistent distributed systems was inspired by Sinfonia [49]. However, Sinfonia provides limited functionality for the back-end servers (memnodes), forcing computation on the front-end. This makes computations such as read-modify-write to be very inefficient. We present three different implementations of one-round transactions in Chapter 3, each suitable for a different purpose. Lock-based isolation was inspired by Sinfonia [49] but provides improvements to the API for better performance, and was developed independently from Granola [72] from which we take clock-based synchronization. One-round transactions serve as the basis for the rest of our applications.

1.6.2 Key/Value Store

Much of the success of large-scale Internet services can be attributed to key/value stores which serve as the backbone of cloud platforms. The same common structures of web services which use stateless front-end servers that indirectly communicate through a scalable key/value store may be applied to FPS games. However, using existing key/value stores in our target application demonstrates that they fall short on one or more required features.

Data retrieval patterns of FPS games include a large number of multi-dimensional range queries which look for objects in the virtual environment within certain areas. Most key/value stores don’t provide such a functionality. This means the gaming system would have to implement its own overlay to maintain such information which will be inefficient compared to a natively supported search capability. The dominant data model supported by key/value stores, i.e., eventual consistency, does not satisfy our design goals. Finally, multi-operation functions caused by the game workload are inevitable as we do not intend to limit the scope of effect of the players.

Considering we found no key/value store to simultaneously provide all of our required features, we implemented our own in Chapter 4. Innesto [116,117] is a strongly consistent distributed key/value store. Innesto uses one-round transactions.
to provide a *data partition* abstraction with range keys. Partitions are automatically distributed to balance load, while allowing different partitioning schemes of the same data to co-exist for improved data retrieval. As part of its API, Innesto provides searching for data on multiple attributes, supports atomic grouping of key-value operations, and provides an extensible API to implement custom application-defined operations.

### 1.6.3 Scalable Spatial Publish/Subscribe

On its own, Innesto provides half the functionality needed by an FPS game, namely, searching for objects with specific criteria. It lacks the other half: tracking them once they’re found. In Chapter 5, SPEX [103] combines spatial partitioning provided by Innesto with a hierarchy-based publish/subscribe mechanism to support spatial publish/subscribe (SPS) [86].

SPS is the minimal abstraction that allows decoupling game logic from the infrastructure hosting it [86]. In essence SPS requires the ability to publish or subscribe to any region of a virtual environment. By definition, SPS can’t tolerate being limited to a single partition. With the aid of one-round transactions, SPEX ensures data consistency is preserved for interactions that span more than a single object. SPEX is the first fully distributed strongly consistent SPS system and can scale to hundreds of players in a single virtual environment. In our experiments we have reached a scale of 750 players as permitted by our hardware resources.

### 1.6.4 Partitioned SQL

Another type of application whose data isn’t trivially partitioned are Relational Database Management Systems (RDBMS). The tight integration between the components of a traditional RDBMS makes it extremely hard to distribute. This has often required redesigning the entire RDBMS and resulted in sacrificing parts of its SQL capabilities [13, 29, 43, 51]. SQL transactions could potentially touch any table in arbitrary order. Once again, the application has to either limit what the transactions can touch to a single partition, or support SQL to its full extent and deal with the consequences. As before, we take the latter approach.

Chapter 6 presents iEngine [119, 125] which is a full-SQL compliant distributed
RDBMS. Data is stored inside Innesto for efficient indexing and multi-attribute range queries. An enhanced version of Innesto designed for distributed locking is used for isolating database transactions using one-round transactions. Query planning and transaction execution is done using unmodified centralized RDBMSs (e.g., mySQL) which are given access to the data through a standard storage interface. While Innesto scales out to provide higher storage throughput, iEngine enables a new dimension of scalability by allowing the application to plug in any number of RDBMS engines for increased data processing throughput.

1.7 Roadmap

We deem data consistency in distributed systems as a problem that originates at the transport level and escalates up through the application stack. When the stack is deep enough, consistency becomes too big a problem to deal with its complexity. We provide a background on data consistency in Chapter 2. Our solution is to provide a means of dealing with consistency at the transport level with one-round transactions which allow turning consistency on and off, explained in Chapter 3. While one-round transactions aren’t anything new, we use them to take a head-on approach at scaling applications which can’t be trivially partitioned. Innesto is the component that uses one-round transactions to abstract away partitions and boundaries, relieving upper layers from having to deal with crossing them (Chapter 4). In Chapter 5 SPEX uses this to safely implement area publications and subscriptions, knowing Innesto will correctly handle ordering of events. Finally, in Chapter 6 iEngine takes advantage of the boundary-less abstraction provided by Innesto to give a single-node RDBMS the familiar feel of being in control of the data it stores.

The two applications built in this thesis each demonstrate a different key feature of one-round transactions. The techniques used can more or less be applied to situations where more than a single processing node is needed. iEngine represents the simplicity of porting a single-node application to a distributed setting with minimal changes. The composability of transactions allowed us to glue together operations such as committing a transaction and write-ahead logging. SPEX presents how strong consistency can easily be enforced in an unpartitionable application such as FPS games.
Chapter 2

Background & Motivation

2.1 Consistency

In distributed systems, multiple threads of execution operate on the state of the application. The threads could belong to different processes on different machines. The data they touch may also be hosted on other machines, rendering low level synchronization primitives such as locks and semaphores useless. A thread has to take one or more steps to complete each logical application task. We call the group of operations done by a single thread to complete each task a transaction. A transaction may involve any or all of reading data, performing computations and writing data in arbitrary order.

With multiple concurrent transactions operating on the same data, based on how the transactions are executed, different data consistency models are possible. The data consistency model defines what different concurrent transactions observe while executing. For example, if multiple different transactions read the same recently modified data item, will they all see the same newest value or will some transactions observe the old value? If a single transaction reads an item multiple times, will it consistently observe the same value or could it possibly see different values as a result of execution of other transactions? Finally, if two transactions modify the same data item at the same time, which one will win (i.e., make the final modification)? Table 2.1 lists a few anomalies possible in concurrent distributed systems.
Table 2.1: Possible anomalies of inconsistent systems [110].

<table>
<thead>
<tr>
<th>Anomaly</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dirty read</td>
<td>A transaction reads an update made by another unfinished transaction.</td>
</tr>
<tr>
<td>Non-repeatable read</td>
<td>A transaction performs multiple reads on the same data but gets different</td>
</tr>
<tr>
<td></td>
<td>results due to concurrent modifications.</td>
</tr>
<tr>
<td>Lost update</td>
<td>Two transactions make concurrent updates to the same data and the results</td>
</tr>
<tr>
<td></td>
<td>of one are overwritten by the other.</td>
</tr>
<tr>
<td>Conflicting fork</td>
<td>Two concurrent transactions make conflicting updates causing the state</td>
</tr>
<tr>
<td></td>
<td>to fork and require application logic to resolve.</td>
</tr>
</tbody>
</table>

Table 2.2: Order of operations $a$, $b$ and $c$ observable by every thread under different consistency models (assuming $b$ causally depends on $a$).

<table>
<thead>
<tr>
<th>Consistency Model</th>
<th>Possible Observations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eventual</td>
<td>Any of $a \rightarrow b \rightarrow c$ or $b \rightarrow c \rightarrow a$ or $c \rightarrow a \rightarrow b$ or $a \rightarrow c \rightarrow b$ or $c \rightarrow b \rightarrow a$ or $b \rightarrow a \rightarrow c$</td>
</tr>
<tr>
<td>Causal</td>
<td>Any of: $a \rightarrow b \rightarrow c$ or $a \rightarrow c \rightarrow b$ or $c \rightarrow a \rightarrow b$</td>
</tr>
<tr>
<td>Sequential</td>
<td>One of: $a \rightarrow b \rightarrow c$ or $b \rightarrow c \rightarrow a$ or $c \rightarrow a \rightarrow b$ or $a \rightarrow c \rightarrow b$ or $c \rightarrow b \rightarrow a$ or $b \rightarrow a \rightarrow c$</td>
</tr>
<tr>
<td>Serializable</td>
<td>$a \rightarrow b \rightarrow c$</td>
</tr>
</tbody>
</table>

Various consistency models exist which are defined by how different threads in a system observe the results of operations executed by others. We present a simple example in which three operations $a$, $b$ and $c$ happen in a distributed system around the same time and in that exact order. Also assume $b$ causally depends on $a$, i.e., it has read the result of $a$ to perform its computations. Table 2.2 presents the possible observable orderings under different consistency models.

In the weakest form, with eventual consistency [123], any ordering is observable for the operations. This means some thread might observe a computation (i.e., $b$) using some computation it hasn’t seen yet (i.e., $a$). In a stronger model, causal
consistency [11] preserves the order of causally dependent operations. Hence, no thread ever observes \( b \) before observing \( a \). However, it is up to the application to ensure dependencies are correctly conveyed to the component that provides causal consistency. With sequential consistency [35], all threads observe the exact same ordering of events globally, but it might differ from the actual ordering that happened in real life. Only with serializability [36] (strong consistency) it is the case that all threads observe the exact same order in which the events actually occurred.

### 2.1.1 Why Strong Consistency?

From a programmer’s perspective using a distributed system, strong consistency is easy to deal with. It hides away the complexities of the distributed system as if the system is single threaded. The developer only needs to focus on the task at hand without requiring in-depth knowledge of implementation details of the system. Invariants are easily maintained without requiring full knowledge of what they are.

Without consistency guarantees, extra effort is required to ensure the application functions properly. This requires detailed a-priori knowledge about the application state and how it is modified. Regardless of the transport protocol used, requests in the system may be re-ordered at any point. A write to a data item followed by a read by the same thread on the exact same item may be executed out of order. As a result, the thread may not read what it just wrote.

Even if single event orderings are preserved, there is still no guarantee about orderings of groups of operations (transactions). Numerous interleavings of operations are possible which could result in drastically different results for the exact same transactions. There is no way for the developer to reason about concurrency. For example, when a read-modify-write transaction is executing, there is no way to ensure the read data hasn’t changed while the modify and write operations were in progress.

Such problems have forced cloud providers to enhance their component APIs with system support for dealing with some of these limitations. Amazon’s SimpleDB [6], an eventually consistent datastore, added a consistent read operation which returns a result that reflects all writes that received a successful response. They also added conditional put and conditional delete. These operations only ex-
execute to completion if the data item has the specified expected value (the condition) or else they abort [7]. This helps protect items from concurrent modifications by rejecting modifications that were based on stale values.

However, making each component in a distributed system consistent on its own will not solve everything. A transaction that needs to operate on multiple data items simultaneously still faces the concurrent modification and interleaving problems. It has no control on the ordering of operations on different machines. A conditional operation on one data item can ensure its value hasn’t changed, but since the execution time of the other operations on the other machines is unknown, it can’t be sure the other values haven’t been modified.

Finally, failures can happen at any time. Even if individual operations are durably protected from failures, if the thread executing a transaction fails midway, it leaves the consistent components in an inconsistent state.

2.1.2 ACID

Using transactions to build services is nothing new. For decades RDBMSs have used ACID [2] transactions to provide strong consistency:

- **Atomicity** Operations in a transaction execute as *all or nothing*. A transaction only completes if all of its operations succeed. No intermediate state is ever observable in the system.

- **Consistency** A transaction takes the state of the system from one consistent state to another. No kind of system failures should result in inconsistent state.

- **Isolation** Different concurrent transactions are isolated from each other as if they were executed serially, i.e., one after another. The effects of an incomplete transaction should not interfere with another transaction’s computations.

- **Durability** The effects of completed transactions will persist through any kind of outages and failures.
Unlike locks, transactions are composable. Two different operations built using transactions can be combined together. By using the same transaction they provide correct concurrency isolation. Durability simplifies application development. All the developer has to do is ensure important state is stored and modified using transactions. No extra failure scenarios have to be considered.

Database transactions are a powerful tool, yet their implementation is extremely complicated. Distributing them is an even more challenging task. It is theoretically impossible to globally serialize concurrent database transactions in a lock-free manner. Scalable RDBMS that provide full-fledged ACID transactions only scale up by adding more expensive hardware with immense licensing costs.

RDBMS solutions offered by cloud providers [4] are still single-node servers. To scale out, an application has to use multiple different instances of the RDBMS, each one for a different purpose. Consequently, the transactions live in different worlds, each used for building different individual services. They can’t be used together to build the entire system. Such limitations have forced applications to use a hybrid architecture of using transactions in the back-end for important application tasks and resort to eventual consistency in the front-end [25].

2.2 Transport-level ACID

The benefits of ACID transactions are too important to ignore when building distributed systems. Thus we take a different approach. Instead of trying to distribute traditional ACID database transactions, we simplify them down to one-round transactions provided at the transport level. The rationale is that a transport capable of providing some form of ACID transactions at scale with reasonable performance could serve as the foundation of scalable large-scale applications.

2.2.1 One-Round Transactions

Database transactions can have multiple rounds of reading and writing different data items between start and commit. The multiple rounds make it impossible to serialize them in a lock-free manner as there is no generic way to predict what the transaction will do next. One-round transactions only allow a single round of communication initiated by commit. There is no limitation on the number of items
that can be read or written by the transaction. The only limit is that they need to be known at commit time.

Compared to using an ordinary transport, the application using one-round transactions is exposed to the same API and message receiving handler (invoked on receipt of each message). What’s added is another handler, invoked before the receiving handler, used for checking consistency requirements. The application can use this handler to define its consistency rules, or simply ignore it to forfeit consistency provided by one-round transactions.

The simplification of transactions provides these benefits:

- **Less-lock/Lock-less serialization** In the worst case, committing a one-round transaction takes 2 network round trip times (RTT) (plus a durable log time or extra RTT for replication) to complete. If locking is used to deal with concurrency, locks will generally be held for a short time, providing better performance under contention than long-lived locks.

  One-round transactions can also be isolated by using clock vectors for lock-free serialization. Without locks, none of the overheads of synchronized execution, lock failures and random back-off are present. Thus one-round transactions can perform well under extreme contention (see Chapters 3 and 4).

- **Message-passing fallback** While transactions may be a suitable solution for certain components of a system, not all components might be interested in the extra overhead of the commit protocol and durability. Fortunately, the commit protocol executes in a single RTT under certain conditions. The application is given the freedom to choose the right balance of performance, durability and consistency.

### 2.2.2 Cloud Unfriendly Applications: A Different Approach

Our new transport based on one-round transactions provides consistency with the desired level of performance. This allows us to use one-round transactions to implement the needed components for our target applications.
Online FPS Games

FPS games are a popular genre where the player observes a virtual environment (VE) from the eyes of an avatar they control. Online FPS games use frequent message passing between different replicas of the VE to synchronize what the players observe. For example, in a server/client architecture, all players send their commands to the server and the server keeps them all up to date with the latest state of the game via frequent update messages. Previous studies have found the enjoyability of the game to be very sensitive to end-to-end latencies. This is the delay between when one player takes an action until the time it is observed by other players in the game. Latencies of 150ms represent the border of enjoyability [67]. Anything more than that will drastically degrade the quality of the game.

Online FPS games present an interesting class of application. The state of the VE is shared between all players in the game. It consists of the objects in the VE, including player avatars. The actions of a player are translated into interactions between the avatar of the player and its target objects, such as other players. Players are continuously moving and interacting with the VE during the game. As a result, the workload of an online FPS game is heavily write-oriented. This is in contrast to other types of online application whose workloads usually have a high read/write ratio (sometimes as high as 30 to 1 [50]).

The game is sensitive to end-to-end latencies. In many scenarios traditional techniques for dealing with concurrency, e.g., locks, and data retrieval, e.g., range queries, will not provide the desired performance. The players controlling avatars are manipulating the objects remotely, usually over WAN distances. What a player observes when taking an action will differ from the latest state of the VE on the server due to propagation delay. Inconsistencies need to be dealt with correctly and fairly.

The players have unpredictable interactions with other objects. Unlike most online applications that can serve users from the nearest replica, geographic proximity has no meaning in online FPS games. Even in the VE players don’t necessarily interact with players virtually closest to them. Various factors contribute to how players select their targets [55]. Interactions are also restricted based on system imposed policies, such as constraints in a physics simulation system. The
interactions are not only unpredictable, but also vary over time as the active battle grounds shift [99]. This prevents a clean partitioning scheme for the state of the VE.

Online gaming is a billion dollar industry. Players invest months of their time building large armies to live the thrill and excitement of participating in battle. World of Warcraft [46] has reported over 5 million monthly subscribers [124]. EveOnline [17] has recently reported over $300,000 USD of in-game ships and equipment destroyed in a single 21-hour battle [39].

Existing state-of-the-art systems do not yet live up to the full potential of epic-scale gaming (see Chapter 5 for a survey). Sharded systems, e.g., World of Warcraft, limit the interactions of players to a subset of others. Partitioned systems, e.g., Second Life [33], limit the interactions of players to their immediate surroundings in the VE. Systems that suffer from neither, e.g., EveOnline, sacrifice consistency. The players currently face one of two options. Either participate in a large yet inconsistent VE and risk their assets on good faith, hoping they’ll end up on the good side of concurrency conflicts. Or choose to settle for a smaller more limited battle with fewer players. Overcoming these limitations will not only broaden the horizon of FPS gaming entertainment, but would also make way for large scale sports activities and social exhibits.

Beyond games, the model of the system is generic enough to be extended to other domains. Imagine replacing the game client with a browser, game objects with user profiles, and physics restrictions with friendship connections to form a real-time social networking service, one that instead of dealing with a soul-less wall, users will more dynamically interact with their friends. Thus solutions presented for epic-scale FPS games may be used for other purposes.

Databases

For decades databases have served as the data storage component of computer systems and applications. Years of research have perfected the tasks of data layout, query planning and synchronization while providing SQL as the universal language for accessing the data. SQL simplifies interactions with the data as it allows a very broad set of possible questions to be asked [45]. It has enabled an ecosystem
of management and operator tools to help design, monitor, inspect, explore, and build applications for enterprises. A SQL programmer can reuse their API and UI knowledge across multiple back-end systems which reduces application development time.

Traditional relational databases, including those from Oracle and IBM, are built with a share-everything centralized topology. This results in tight coupling of internal components. This doesn’t mean they don’t scale. But rather, when it comes to scalability they scale up. Throughput is only improved by adding more expensive hardware with immense licensing costs, as opposed to scaling out by combining multiple cheaper hardware units to achieve the same result.

Finding distributed alternatives to mathematically proven properties of centralized SQL engines is a non-trivial task. Systems with similar goals have ended up sacrificing some part of SQL along the way (see Chapter 6). We provide consistency at the transport level, therefore we take a different approach in trying to scale out a database. We leave the engine intact, but allow simultaneous engines to co-exist in the system and take care of correct synchronization ourselves. As a result, we provide full SQL while allowing a new dimension of scalability.
Chapter 3

One-Round Transactions

One-round transactions use two-phase commit to communicate with multiple servers in parallel. They are designed to complete in the shortest amount of time theoretically possible, which is two network round-trip times (RTTs)\(^1\) [49\textsuperscript{72}]. The term one-round [72] is the fundamental property that distinguishes them from traditional database transactions which allow reading and modifying data in multiple rounds of communication during execution. A one-round transaction has 2 key properties: (1) all data items referenced by it should be known when execution starts, and (2) execution will run to completion, i.e., will not abort for application defined reasons or any form of failures. Any number of transactions can run simultaneously for increased throughput. Under certain conditions where a transaction only reads data from the servers or only touches data stored on a single server, one-round transactions can commit in a single RTT.

Applications can extend the one-round transaction framework to implement any kind of custom transactions. The application uses the client library to create, populate and commit transactions which internally communicates with the servers storing the data. Servers only respond to requests that follow the commit protocol. The application has to implement its own server handlers to deal with the operations it uses. The framework handles committing each transaction to completion.

\(^1\)Assuming each application message is sent to the receiver in one RTT which is generally true in steady state communications under normal network conditions
Listing 3.1: Transaction Client API

```c
// Create a new transaction.
Tx newTx();

// Add a new item with id 'dataId' on server 'serverId' to the transaction.
// 'op' specifies the app-defined operation that should be performed on the
// data item. 'readOnly' specifies if 'op' will modify the item or not.
Tx addItem(
    Tx tx, Id serverId, Id dataId, Operation op, Data data, bool readOnly);

// Try to commit the transaction and return commit success result. Fills
// 'result' with items returned by the transaction.
bool commitTx(Tx tx, out List result);
```

A lock-free one which uses clock vectors, each useful under different scenarios. As part of the commit protocol, state modifications are also protected from failures. In the event of a failure, a separate tool known as the recovery manager is required to clean up half-committed transactions. Parts of the data may become unavailable due to failures and while recovery is in progress, but it will never be in an inconsistent state. In the remainder of this chapter we present details of one-round transactions. For simplicity, from now on we use the word transaction when referring to one-round transactions.

### 3.1 API

An application trying to use transactions is exposed to a client library API and a server interface. The client API is used for creating and committing transactions. The server interface has to be overloaded to provide the necessary functionality of the server.

#### 3.1.1 Client

The client API can be found in Listing 3.1. Each client library instance is assigned a unique client identifier. When a transaction is created, it is assigned a globally unique transaction identifier. The transaction id is produced as the concatenation of the client id and a monotonically increasing counter.

A data item is referenced by a tuple specifying the server id which stores the item and the item’s unique id. Unlike distributed hash tables [112] where the server
id of an item is inferred from its id and the network topology, we allow items to freely migrate between servers for load balancing purposes. The application can use transactions to perform an operation on any item by simply adding it to their items. When adding an item, the application also has to specify if it might be modifying it. This information is used to improve the commit performance of the transaction.

Commit either fails or succeeds, returning a custom list of messages provided by the servers in response to the transaction. During commit transient problems (such as lock conflicts and server failures) are handled internally. Commit failures only happen when the transaction encounters a permanent problem which cannot be resolved without help from the application, e.g., concurrently modified data. The result list of the transaction includes reasons as to why commit failed upon failure, or computation results in the case of success.

### 3.1.2 Server

The server functionality is extended by overloading two handlers as part of the server interface (Listing 3.2). The handlers are invoked for each transaction and give the application full control over them.

The first handler, `prepare`, asks the application to validate the assumptions of the transaction to ensure it can proceed. This involves checking the existence of items referenced by the transaction and blocking the assumptions from changing until the transaction is done when required. At the end, the application decides if the transaction can proceed, has failed due to a permanent problem, or encountered a temporary problem and should be retried.

The second handler, `finalize`, is invoked once the final outcome of a transaction has been decided with regard to all the involved servers. The handler notifies the application to proceed with applying the changes of the transaction to the state or to rollback and abort.

### 3.1.3 Sample Transaction

Listing 3.3 illustrates a sample transaction that operates on three items stored on two different servers. The transaction performs the custom defined operations
Listing 3.2: Transaction Server Upcalls

```c
VoteType prepare(int64 txId, List items, out List result);
```

// Check to ensure the transaction is operating on valid 'items'. Return
// INVALID if any of the items are invalid, FAIL if there is a concurrent conflict
// with another transaction. Else return OK and locks the item if necessary.
// Anything that should be returned are stored inside 'result'.

// Finalize execution of the transaction to either proceed or abort.
```c
finalize(int64 txId, List items, bool proceed);
```

Listing 3.3: Sample Transaction

```c
tx = new tx();
tx_add_item(tx, server1, item1, WRITE, 'Data to store', false);
tx_add_item(tx, server2, item2, READ, Null, true);
tx_add_item(tx, server2, item3, COMPARE, 'Value to compare to', true);
List result;
if commit(tx, result) == true:
    // Transaction succeeded.
    // result[item2] contains contents read by the transaction.
else:
    // Transaction failed.
    if result[item2] == INVALID or result[item3] == INVALID:
        // 'item2' or 'item3' no longer exist.
    else:
        // Compare failed on 'item3' which has a different value.
```

`WRITE` to write the given data to the item, `READ` to read the contents of the item
and `COMPARE` to compare the provided value against the current value of the item.

Once commit is initiated, the outcome of the transaction is returned. This trans-
action can only fail under two scenarios. Either one or more of the referenced items
do not exist, or the comparison of the third item failed against the latest value on the
server. Each failure would trigger a specific application response and can be differ-
entiated from each other by using the result messages returned from the servers. If
the transaction commits successfully the modifications, in this case writing to the
item, are applied and the result will include valid data read from the servers.

The server-side implementation of the sample transaction using locks can be
found in Listing 3.4. In the prepare upcall, the server first checks for invalid item
references and locking conflicts. Then it proceeds to lock items to ensure no other
concurrent transaction can invalidate this transaction’s assumptions. Finally, re-
quested data is piggy-backed to the client. Locks are released in the finalize upcall
Listing 3.4: Server-side Implementation of the Sample Transaction with Locks

VoteType prepare(int64 txId, List items, out List result) {
    // Check for invalid items or lock conflicts.
    foreach (item in items):
        if not item in stored_items:
            result[item] = INVALID;
            return FAIL;
    
    if locked(item):
        return ABORT;

    // Lock items for this transaction and perform preliminary actions.
    foreach (item, op in items):
        lock(item, txId);

        switch op:
            case WRITE:
                break;
            case READ:
                result[item] = stored_items[item];
                break;
            case COMPARE:
                if stored_items[item] != item:
                    return FAIL;
                break;

    return OK;
}

finalize(int64 txId, List items, bool proceed) {
    if proceed == true:
        foreach (item, op in items):
            switch op:
                case WRITE:
                    stored_items[item] = item;
                    break;
                case READ:
                    break;
                case COMPARE:
                    break;

    // Unlock items if locked for this transaction.
    foreach (item, op in items):
        if txId == lock_owner(item):
            unlock(item, txId);
}
and if the outcome is to proceed, modifications are made to the stored data.

### 3.2 Commit Protocol

The client trying to commit a one-round transaction is the *Coordinator*, and each server whose id is included in the items of the transaction is a *Participant*. To commit, the Coordinator sends the transaction to each Participant and each Participant votes to either proceed or not. The outcome of the transaction is then decided by the aggregation of all the votes; if all Participants vote to proceed, the outcome is to proceed, else the outcome is to abort. Once decided, the Participants finalize execution of the transaction. If the final outcome is to proceed, modifications requested by the transaction are applied to the stored data. Otherwise, they are simply discarded.

In cases where the vote can be inferred by all Participants without an explicit message from the Coordinator, the commit protocol completes in a single RTT. Specifically, when a transaction only touches data on a single server, or the transaction only performs read operations, there is no need for the extra round of communication.

At any given time multiple transactions may be executing concurrently which should be isolated from each other to prevent race conditions. Isolation happens using two different methods that vary in the details of voting and when a transaction is run by each Participant and the upcalls are dispatched. However, each server could be servicing both kinds of transactions at the same time. Figure 3.1 illustrates the messages exchanged during commit, and Table 3.1 contains the description of the messages.

#### 3.2.1 Lock Mode

In this mode, logical locks are used for isolation (Figure 3.1a). The core voting and locking protocols used in lock mode are similar to Sinfonia [49] with the addition of custom operations via server upcalls.

The Coordinator sends a *PREPARE* message along with `txId`, `items` and `ro` to the participants which invokes the *prepare* upcall. Each participant validates all items referenced in `items` to exist and to be unlocked. If there are invalid items
Figure 3.1: Transaction commit protocol.
Table 3.1: Description of message fields used in the commit protocol.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>txId</td>
<td>Globally unique transaction Id.</td>
</tr>
<tr>
<td>items</td>
<td>List of items in the transaction intended for the recipient Participant. Each item has a serverId specifying the target Participant, itemId uniquely specifying the target item, operation defining the custom operation that needs to be done, and readOnly flag indicating if operation will modify the item’s content or not.</td>
</tr>
<tr>
<td>ro</td>
<td>Flag specifying if the transaction only consists of read-only items.</td>
</tr>
<tr>
<td>vote</td>
<td>The vote of a participant which could either be: OK, FAIL or RETRY.</td>
</tr>
<tr>
<td>results</td>
<td>List of messages sent from the Participants to the Coordinator.</td>
</tr>
<tr>
<td>outcome</td>
<td>Boolean flag describing the outcome of commit.</td>
</tr>
<tr>
<td>timestamp</td>
<td>Virtual time at which the transaction is executed by all Participants.</td>
</tr>
<tr>
<td>clientTs</td>
<td>Virtual clock maintained by each client.</td>
</tr>
<tr>
<td>itemTs</td>
<td>Virtual clock maintained by a server per item.</td>
</tr>
<tr>
<td>proposedTs</td>
<td>Virtual execution time of a transaction proposed by a Participant.</td>
</tr>
</tbody>
</table>

referenced, vote is to FAIL, else if there is a locking conflict, vote is to RETRY. Otherwise, items are locked and read only operations are executed with their results stored in results. If any operation fails, vote is to FAIL, else the Participant votes OK.

Once the Coordinator receives all votes, it can decide the outcome of the transaction. Commit fails if any Participant votes FAIL or RETRY. The latter case being necessary for preventing deadlocks for which commit is retried after waiting a random amount of time with exponentially increasing value. Only if all Participants vote OK will commit succeed. The Coordinator sends a FINALIZE message with txId and outcome to the Participants (invoking the finalize upcall), and notifies the application unless commit has to be retried. In the finalize upcall, write items are applied and locks are released.

3.2.2 Clock Mode

In this mode, inconsistencies caused by race conditions are avoided by having all servers execute transactions in the exact same serial order. To achieve serializabil-
ity, transactions are executed on all servers using a globally unique order which is decided using clock vectors \[72\]. Hence, there is no need to lock data items. The clock vector protocols were taken from Granola \[72\] where synchronization granularity was increased to per-item clocks for improved performance.

In clock mode, a transaction commits at a specific \textit{timestamp}, a virtual execution time. Each client maintains its own clock, \textit{clientTs}, as the highest timestamp of any committed transaction it has observed so far. Upon commit, the Coordinator sends a \textit{PREPARE} message along with \textit{txId}, \textit{items}, \textit{ro} and \textit{clientTs}. The participants invoke the upcall and the application returns \textit{vote}. Additionally, each Participant proposes a timestamp \textit{proposedTs} for the transaction as \( \text{MAX(clientTs, } \forall \text{itemTs)} + 1 \).

In the base protocol (Figure 3.1b), instead of contacting the Coordinator with their votes, the Participants directly send \textit{txId}, \textit{vote} and \textit{proposedTs} to each other. Each time a Participant receives a vote, it sets \textit{proposedTs} to the maximum of what it has and what it receives. This ensures all Participants will eventually agree on the same final timestamp for the transaction.

Per item, Participants maintain a sorted list of transactions in increasing order of proposed timestamps. \textit{TxId} is used as the tie breaker. Transactions are then executed in-order. A participant never invokes the \textit{finalize} upcall on an item until all votes have been received. This ensures the exact same execution ordering of dependent transactions by all Participants. Once a Participant finishes executing a transaction, it sends a notification to the Coordinator which includes \textit{txId} and \textit{timestamp}. When the Coordinator receives completion notification from all Participants, it updates it \textit{clientTs} to be greater or equal to \textit{timestamp}.

The base protocol in clock mode has the same complexity as lock mode, i.e., 1.5 RTTs, but the message complexity grows quadratically with the number of Participants. This causes trouble in cases where a transaction may touch items on many servers. The modified commit protocol in Figure 3.1c offloads deciding the timestamp of the transaction to the Coordinator with an added extra half RTT (same as lock mode).
3.2.3 Lock vs. Clock

Each of the different isolation modes has an advantage over the other in certain scenarios. If multiple transactions simultaneously access the same data items, clock mode avoids three sources of latency present in lock mode. First, as there are no lock conflicts, commit never has to retry the transaction. This avoids extra RTTs for the retry and saves wasted CPU and bandwidth resources. It also doesn’t create extra contention on other items in the transaction which have to constantly be locked and freed due to locking failures on another server. Second, as a result of no retries, there is no exponential back-off and idle periods for the transaction. Third, execution of operations on different servers isn’t synchronized, avoiding wait periods for other servers to catch up.

Clock mode performs better under extreme contention compared to lock mode, but imposes limitations on the types of operations that may be performed when the transaction touches multiple items. In lock mode all items are locked before voting. This means they will not change until the transaction is done executing. In clock mode, there is a window of time between when the vote is issued until when the transaction is done where the items may be modified. Extra care should be taken by the application developer when designing their operations to ensure no consistency requirements are violated. They should be aware of this limitation when choosing the right isolation mode for their application.

We found a hybrid of lock-clock to be very useful, in which isolation is done using clock vectors but we still use locks for certain items to ensure correctness. We use this technique in Innesto (Chapter 4). The insight here is that for performance reasons data items aren’t locked by using clock mode, while item containers (partitions) are locked for preventing items from disappearing during commit.

3.3 Durability & Failure Recovery

As part of the one-round transaction commit protocol, a Participant makes a stable log of important information about the transaction before issuing its vote. Logging can be either done synchronously to disk or by replicating the transaction to backup servers to avoid disk access. In lock mode, if vote is OK a Participant logs txId, list of Participants, and items that don’t have their readOnly flag set (i.e., write items).
In clock mode, it additionally logs $proposedTs$ as well.

If a Participant fails, its dedicated log can be used by a new server to reconstruct the failed Participant. If the Coordinator fails, a Participant can directly communicate with other Participants to finalize lingering transactions. While recovery is in progress, parts of the data affected by the failures will become unavailable. This is the price paid for consistency based on the CAP theorem [84]. Beyond this point we can assume data modified via transactions is fully reliable, recoverable and always consistent.
Chapter 4

Innesto

4.1 Introduction

Key/value datastores serve as the backbone of scalable services and host hundreds of millions of data items and simultaneously service millions of clients. Current key/value datastores, such as Dynamo [76], Cassandra [94] and BigTable [65], scale across a large number of servers and provide high availability guarantees under extreme failure scenarios, such as massive network outages or hardware failures [76]. Relaxing the data model to provide eventual consistency through a simplified API that allows accessing data via put, get or remove individually have contributed as major features enabling such scalability and high availability.

While key/value stores are essential to online cloud-based applications, trading out everything in favour of scalability and availability does not present an appealing solution for all applications dealing with big data. This has caused a general lack of interest from enterprises in the NoSQL paradigm [113]. The problem is caused by three important limitations in the design of traditional key/value stores:

- **Data consistency** The eventual consistency data model implies that in a distributed system if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value [123]. However, there is a window of time after an item is updated that different reads on the same item will return different values. This pushes unwanted complexity up
to the application. The developers have a hard time reasoning about data freshness uncertainty which could lead to correctness issues in application logic.

- **API** The narrow API to put, get and remove a data item based on a unique key is insufficient for an application to freely express how data should be retrieved. Applications require accessing the data based on secondary attributes. This forces developers to use various techniques when modelling their data to be able to retrieve items by means other than their key [38]. To this end, commercial cloud service providers have developed their own languages which provide a means of finding data using queries [10, 21].

- **Concurrency** ACID (i.e., Atomicity, Consistency, Isolation, Durability) properties mandate that read/write operations may be grouped into transactions. Operations in a transaction are executed together while taking the data from one valid state to another. The overall outcome of executing multiple concurrent transactions would appear as if they were executed serially one by one and no form of system failures should violate these properties. In a system that allows multiple concurrent read/write operations on data, system support is required to correctly deal with concurrency. This feature is missing from the basic key/value API of traditional systems. Without some form of transactions (no matter how primitive), application developers can’t deal with race conditions and concurrency.

In related work, numerous improvements have been made to amend the so-called first generation key/value datastores by extending their API to include simple range searches [69], support conditional operations as a primitive form of transactions [6, 105], or provide a stronger data consistency model [108]. However, the largely neglected missing feature from the first generation is a high degree of expressibility when dealing with data retrieval. In the age of big data, recommendation systems process massive amounts of data through numerous different attributes to provide per-user customizations. Such applications need to be able

---

1Interestingly, when given the choice, developers opt for a semi-relational storage with bad write performance (Megastore) over a scalable eventually-consistent key/value datastore with superior performance (BigTable) [71].
to freely reference data items via various attributes in the form of multi-attribute range searches, which is why they are heavily coupled with the relational SQL data model due to its flexible queries. There are simply no direct mappings from such well understood OLTP tasks to the basic key/value API, calling for a new generation of key/value datastores. HyperDex [78] marks the birth of second-generation datastores which natively supports search as part of its API. However, HyperDex is not architected for data with high dimensionality.

Our goal is to address the multi-attribute range search issue on highly dimensional data in a scalable fashion to help bridge the gap between the feature set of traditional key/value stores and the performance limitations of relational databases. We thus present Innesto, a searchable key/value datastore for data with high dimensionality. Innesto overcomes all three limitations of traditional key/value stores. It provides a strong data consistency model. Thus, application developers can assume Innesto is a single server single threaded storage system, while in fact it is fully distributed and scalable. The API is richer than that in traditional key/value stores; Innesto supports multi-attribute range search on any number of secondary attributes of data items. It also provides ACID transactions for grouping operations to deal with concurrency.

Innesto uses a combination of old and new techniques. Efficient range search is supported by using spatial partitioning [31] and maintaining a hierarchy [24]. Various different types of range search on different attributes are supported by using a set of parallel clones of the data [78], where each clone is optimized for a special set of range requests. Internally, Innesto uses high-performance one-round transactions to consistently update the data and its clones in parallel. One-round transactions execute client requests in a consistent and fault-tolerant manner, even in the face of group server failures. Concurrent operations are isolated from each other by either using short-lived locks on data items or through clock vectors [72], depending on the access patterns of the application. To avoid the imbalance due to spatial partitioning, Innesto dynamically partitions the data and performs online load balancing by restructuring partitions. Each operation is carefully designed to execute in the least amount of time possible. Multi-versioning is used to reduce the overhead of supporting parallel clones to efficiently answer more types of queries.

We evaluate our implementation of Innesto using an industrial cloud bench-
Table 4.1: Innesto’s API which includes basic key/value operations plus search.

<table>
<thead>
<tr>
<th>Op</th>
<th>Args</th>
<th>Return</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>put</td>
<td>table, key, attribs, value</td>
<td>-</td>
<td>Insert value associated with key</td>
</tr>
<tr>
<td>get</td>
<td>table, key</td>
<td>value</td>
<td>Get value associated with key</td>
</tr>
<tr>
<td>remove</td>
<td>table, key</td>
<td>-</td>
<td>Remove value associated with key</td>
</tr>
<tr>
<td>search</td>
<td>table, req_attribs</td>
<td>list{value}</td>
<td>Get all values satisfying req_attribs</td>
</tr>
</tbody>
</table>

mark and compare its performance with Cassandra [94], a scalable widely used key/value datastore, and show it has competitive performance while offering a myriad of extra features. Our results also show Innesto outperforms HyperDex [78] in every aspect. In Section 4.2 we present an overview of Innesto. Sections 4.3 & 4.4 elaborate on how key/value operations and multi-attribute search are executed, while Section 4.5 presents Innesto’s extended API. Evaluation results are in 4.6 and a review of related work can be found in Section 4.7. We conclude in Section 4.8.

4.2 Overview

4.2.1 Data Model & API

Innesto is a key/value datastore designed to extend the feature set of a normal datastore while maintaining scalability. The goal is to allow new types of applications that cannot tolerate the limitations of key/value stores to migrate to the cloud, while existing applications can be extended using the superior features offered by Innesto.

A data item has an immutable key, a set of typed secondary attributes and a value with arbitrary size. Innesto uses a table abstraction to logically group data items together using an application-defined schema. The schema sets the number and types of the secondary attributes. An attribute can have any data type that has a definable strict ordering of values. This abstraction allows for an effortless migration from existing systems that rely on traditional databases to Innesto, while other applications can use a single table representative of the global namespace.
Figure 4.1: Logical architecture of Innesto components. Each component is fully distributed.

offered by normal key/value datastores.

Table 4.1 presents Innesto’s API. The basic API of a datastore uses an item’s key to put, get and remove the item in a specific table. For put, the item’s secondary attributes and its value should also be provided. Innesto’s API also supports search on secondary attributes. The user specifies constraints on the secondary attributes, such as an exact value or a given range, and Innesto will return a list of items with attributes satisfying the constraints. For example, a table storing items based on the physical $X$, $Y$, and $Z$ locations can search for items within a specific physical region, or even items that are at most a specific distance away from a given point in the 3D space.

4.2.2 Architecture

Figure 4.1 illustrates the logical components of Innesto. Each component is fully distributed between Innesto servers which can simultaneously act as multiple different components. Client requests are served by the Proxy which can scale up and down by spawning new Proxies to accommodate current demand. Each proxy
is responsible for identifying the set of servers that need to be contacted to complete each operation. It then communicates with the rest of the components using one-round transactions and ensures each operation runs to completion, after which it notifies the client. Data items are stored in the Multi-Version Storage (MVS) component and a number of Search Clones which are also distributed between all servers to accommodate load. MVS is the main repository for storing data, while the Search Clones are used for efficiently executing search requests. Each clone is optimized for a specific type of search and requests are routed to the most suitable clone. We discuss MVS and Search Clones in Sections 4.3 & 4.4.

4.2.3 Strong Consistency

Using one-round transactions to execute key/value operations allows Innesto to provide ACID (atomicity, consistency, isolation and durability), the strongest possible level of data consistency. This means when the execution of an operation completes, any other subsequent operations completed after it will reflect its results. For example, search returns a consistent snapshot of items, no matter what clone it contacts.

4.3 Multi-Version Storage (MVS)

4.3.1 Partitioning

MVS is a distributed storage system for data items. A full copy of each item is stored using its key. Data is distributed by spatially partitioning the key space. Each data item is versioned, where the version number is incremented each time the item is modified with write operations (put and remove). When reading an item (get), an explicit version of the item can be requested or the latest version is returned. Adjustable parameters define the number of versions to keep and time to live of each older version, which keep the overall overhead of multi-versioning bounded.

Each table is distributed between servers into partitions as illustrated in Figure 4.2. Note that only single dimensional partitioning is depicted, however, Innesto supports partitioning on any number of dimensions. A partition has specific bounds.
Figure 4.2: Spatial partitioning and distribution of a table. The root partition covers the entire key space and each partition keeps references to child/parent partitions. Only leaf (gray) partitions store the actual data. Partitions are distributed between servers for load balancing.

and can only store items with keys that fall within its bounds. Each table starts as a single partition covering the entire key space. As the load of a partition increases, it is split into smaller partitions. The new child partitions each cover smaller parts of the parent’s key space and are placed on different servers to balance load. Underutilized partitions are merged to reduce fragmentation. Each partition has a globally unique ID with no strict server affinity. It can freely migrate between servers at any time. This allows us to decouple partitions from physical machines for better load balancing. Partitions keep references to their parent and children. Thus by knowing how to find the root partition of a table, all other partitions of the table can be subsequently located.

4.3.2 Non Structure-Modifying Operations

Normal key/value operations don’t directly change the structure of a table (i.e., split/merge/migrate partitions). A single one-round transaction is used to execute an operation which executes in one RTT since in MVS each operation touches exactly one partition. The transaction execution protocol ensures proper isolation
from other concurrent operations so a partition can act as a simple single-server key/value store with a trivial implementation. For put the transaction includes the item’s key, attributes and data, while for remove it just includes the key. For get the item’s version is optionally provided with the key and its value is returned along with the server’s vote as part of the commit protocol.

Each partition has a logical structure lock associated with it. Regardless of the isolation mode (i.e., lock mode or clock mode), a transaction temporarily blocks the partition it is referencing from re-structuring by acquiring the structure lock on the partition. This prevents the partition from splitting/merging/migrating while the transaction is executing. If the partition cannot be locked, it means a re-structure is in progress and the operation is touching the wrong partition; if the partition is splitting/merging, the transaction is referencing the wrong partition, if it’s migrating, the transaction is talking to the wrong server. Regardless, the transaction aborts and retries using the correct partition.

4.3.3 Structure-Modifying Operations

The structure of partitions is also altered using one-round transactions to be coherent with operations. Any structure-modifying operation has to acquire the structure lock on the partitions it touches before proceeding. This ensures partitions don’t simply disappear while an operation is halfway through execution. The process of acquiring the structure lock waits for transactions in flight to finish, while delaying new transactions from starting until the lock is released. Once the locks on the partitions are acquired, re-structuring begins. Since each partition has its own structure lock, only small parts of a table become unavailable when being re-structured. All other operations touching other partitions proceed as they normally would.

Partitions are split based on load in order to distribute overall load between more servers. Two factors contribute to a partition’s load. First is data congestion where the number of items stored in a partition exceeds a threshold. Second, is when the rate of transactions a partition serves exceeds its capacity. Since split is triggered under heavy load, it is designed to complete in one RTT. A special one-round transaction is used to create new child partitions with specific bounds,
populate them with the data they should hold and push them out to other servers. Strong consistency requires serialization of operations. If certain items tend to become over-popular, the load balancer will keep splitting the partitions hosting such items down to partitions holding a single hot item. This provides the finest granularity of serialization possible while still maintaining consistency.

Migrating a partition to another server is useful in various scenarios. For example, when a new server is spawned to scale the system, or when a server is to be terminated to reduce operational costs, partitions need to be migrated between servers. Also, in the case where the load of each partition is below the threshold but the aggregate load of all partitions exceeds the capacity of the underlying physical host, migrating some partitions can help balance load. Migrating a partition uses a one-round transaction to move data off one server onto another, and simultaneously fix parent/child references. Thus migrate executes in two RTTs. Merge is to reduce fragmentation by combining under-utilized partitions into one. It takes a couple of transactions to trigger and execute merge.

4.3.4 Partition Hierarchy

The hierarchy formed by the partitions of each table allow every server to be able to locate all the partitions belonging to that table by only knowing where the root is. This hierarchy is used to efficiently compute the servers that need to be contacted per operation. However, in a large system with billions of items stored in tens of thousands of partitions spread across hundreds of servers, it is not efficient nor necessary for every server to know where each partition is exactly located. It need only be able to locate one when required.

To prevent constant network accesses each time a server tries to execute an operation, the partition hierarchy of each table is lazily cached by a server as soon as it initiates its first operation on the table. This provides the servers with item locations on a need-to-know basis and speeds up operation execution. Thus at any given time, no server has to have a global view of the partition hierarchy.

Over time, parts of a server’s cache will go out of sync with the actual hierarchy due to structure modifications. This will cause operations using stale cache entries to reference the wrong partitions or servers. To prevent this, the Participants of
Figure 4.3: Search Clones are used to partition data based on secondary attributes. Items are grouped into attribute groups to better support range search. Each $K_i$ is a full copy of an item.

each one-round transaction validate that the transaction was initiated using the right partition hierarchy. If not, the operation is rejected and the Coordinator is notified by a special message piggy-backed on the vote. The Coordinator will then back track the steps it took when executing the operation to find the stale parts of its cache and re-synchronize them. Lazy caching allows structure modifications to be spaced through time when being propagated to the servers, avoiding an immediate broadcast to hundreds of machines upon each change. This significantly reduces the network costs of structure modifications.

4.4 Search

4.4.1 Search Clones

Unlike normal key/value operations that use keys to uniquely identify data items, search is required to allow the caller to specify some constraints on secondary attributes and should return all data items in a table that satisfy the given constraints. A constraint could be a specific value on the attribute, a given range specified by minimum and maximum values, or * which means to search the full possible range
of the attribute.

Items are stored in MVS based on their key which is oblivious to secondary attributes. To provide search on a table, Innesto creates parallel Search Clones [78]. Each Clone is a separate copy of the entire table partitioned differently based on a subset of secondary attributes. In our example of storing items with secondary attributes equal to their physical \(X\), \(Y\) and \(Z\) position, a Clone may be partitioned based on any subset of the set \(\{X, Y, Z\}\). In each Clone, items with the same value on the subset of attributes are grouped together into an attribute group. This will ensure items in the same attribute group are always positioned in the same partition. Figure 4.3 illustrates how items are put into attribute groups. A Clone using the subset \(\{X\}\) will place all items with the same \(X\) value in the same attribute group, while a Clone using \(\{X, Z\}\) will group items with the same \(X\) and \(Z\) together.

Data stored in the Clones is kept consistent with MVS by using the same one-round transaction which updates MVS to also update the Clones. This will cause write operations (put & remove) which used to touch one partition to now touch multiple partitions (one partition in MVS and one in each Clone), forcing them to complete in two RTTs. Partitioning based on secondary attributes helps improve search performance since the Coordinator can quickly traverse the partition hierarchy of a table and identify partitions that hold potential results. Their data is read using a single read-only one-round transaction that completes in one RTT and returns a consistent snapshot of data upon completion. Adding Search Clones helps improve search performance by reducing write performance. Recent studies have found high read/write ratio of large-scale key/value stores [50]. Thus such a trade-off seems reasonable, while tables needing high write performance can disable Clones altogether.

Most applications know the kind of search they will more likely execute, and the application is provided the freedom to specify which Clones should be created at table creation time, if any. It is also possible to add and remove Clones after the table has been created during runtime, though the table will be unavailable while adding a Clone. To add a Clone, write operations have to be paused until the new Clone is successfully added. The servers hosting MVS are told to duplicate all the data they have of the table into the new Clone. Removing a Clone is much faster and write operations can execute normally while a Clone is being deleted.
It might seem that with $D$ secondary attributes, a single Clone that partitions based on all $D$ attributes is enough to answer any type of search. While this is true in theory, in practice it is not efficient, a problem known as the curse of dimensionality [54]. The number of partitions grows exponentially as more attributes are used for partitioning. This will lead to numerous partitions spread across almost all the servers. Hence, search requests with * on one or more attributes will be forced to contact all the servers. In contrast, partitioning using fewer attributes reduces the number of partitions and increases the chance that each search will contact a single server which reduces network load. However, since partitions can only be split down to attribute group granularity, caution needs to be taken when requesting Clones. Imagine a secondary attribute being a single bit. Partitioning millions of items based on a single bit is very inefficient.

4.4.2 Sparse Clones

With large datasets, duplicating multiple copies of the data is not a reasonable approach. Innesto allows Sparse Clones that only keep a copy of an item’s key, secondary attributes\(^2\) and version number. Since data in the Clones is updated synchronously with MVS, the version numbers of the items are consistent with each other. This reduces the space required for storing arbitrary size items to a small predictable overhead.

Using Sparse Clones only affects search performance. A search on a Sparse Clone returns the key and version number of each item. The value of the results are then retrieved from MVS using extra get operations on the specific versions. This adds an extra RTT to search. Table 4.2 shows the overall summary of operation execution complexity and storage space required for a table that uses no Clones, full Clones or Sparse Clones.

Usually MVS will have the requested version of an item. If for any reason a specific older version of an item is no longer available, the results are invalidated and search is retried. Note that each partition can separately tune its parameters

\(^2\)Only the attributes used for search. We expect the size of secondary attributes to be small and comparable to the key size, and much smaller than the actual data. For applications where the data is represented as a combination of its attributes, an order preserving hash function can be used to decrease their size.
Table 4.2: Innesto’s latency and storage requirements for storing $N$ items with different configurations.

<table>
<thead>
<tr>
<th>Config</th>
<th>Write*</th>
<th>Read</th>
<th>Search</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVS</td>
<td>1 RTT</td>
<td>1 RTT</td>
<td>n/a</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>MVS + K Clones</td>
<td>2 RTT</td>
<td>1 RTT</td>
<td>1 RTT</td>
<td>$O(KN)$</td>
</tr>
<tr>
<td>MVS + K Sparse Clones</td>
<td>2 RTT</td>
<td>1 RTT</td>
<td>2 RTT</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>

* Does not include stable log cost. For replication add an extra RTT, for disk logging add a synchronous write duration.

of how many versions to maintain and for how long. Simple statistical analysis techniques can be used to tune these parameters if search operations start to fail. Also, a partition can switch to and from being Sparse at any time. This is useful if a write-heavy partition is required to store long histories of data.

### 4.5 Extended API

#### 4.5.1 Operation Transactions

Composability is an important feature of transactions which allows the functionality of multiple transactions to be combined into one. Since all Innesto operations use one-round transactions, any combination of operations can be atomically batched together in the form of larger transactions which will execute all at once or not at all, e.g., put, remove and get a group of items altogether. Also, the operations need not be limited to a single table. While they will still be one-round transactions and therefore not as powerful as full-fledged database transactions, they provide enough functionality for many OLTP tasks.

#### 4.5.2 Conditional Operations

Since voting is part of the one-round transaction commit protocol, conditional operations are another extra feature added to basic key/value operations. A conditional operation is a normal key/value operation that only completes if a certain number of conditions hold true. For example, a conditional put is one that only inserts the item if the item’s value is equal to the specified condition. Consider an
Listing 4.1: Extended API example that acquires a lease using operation transactions. The caller’s id is set using a `put` and a counter is incremented only if the lease is acquired using a `conditional_put` that checks the lease to be free beforehand.

```java
tx = new Tx();
innesto_op(tx, table_1, COND_PUT, leaseId, 0, 1);
inestroy_op(tx, table_2, PUT, ownerId, appId);
inestroy_op(tx, table_3, INCREMENT, counterId);
if commit(tx) == true:
    // Lease successfully acquired.
```

application that needs to increment a shared counter. This is done by first reading the counter value, incrementing it and putting the new value with the condition that the current value of the counter be what was originally read. Conditional operations can also be used to acquire leases on certain data or change the data if a lease is held. The need for conditional operation has forced other systems to add them to their API (e.g., SimpleDB [6] and Spinnaker [108]).

Conditional operations are the only Innesto operations where the functionality is different depending on the isolation mode. In the one-round transaction commit protocol, the condition of conditional operations is checked at vote time, while the modifications are applied when the outcome of the transaction is decided, which potentially may be some time later. In lock mode, the conditions and target modifications are locked and conditional operations can impose a condition using one key and apply a modification on another key (e.g., if item 1 has a certain value then put item 2). They can also be combined with any other operations without restrictions. In clock mode, since data is not locked, there is a window of time between voting and the outcome where the condition could be violated. Hence, in this mode conditional operations can only impose conditions and modifications on the exact same key (eliminating the window of vulnerability), and cannot be combined with any other operation in transactions.

### 4.5.3 Extensions

The final feature of Innesto is to allow application-specific extensions to be added to basic operations. This will allow shipping computation with an operation. In the
Table 4.3: YCSB workload specification in terms of operation mixtures and access distribution. By default, each item has ten attributes with a total size of 1KB.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Characteristic</th>
<th>Operations</th>
<th>Dist.</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Update Heavy</td>
<td>50% Read/50% Update</td>
<td>Zipf</td>
<td>Session Store</td>
</tr>
<tr>
<td>B</td>
<td>Read Mostly</td>
<td>95% Read/5% Update</td>
<td>Zipf</td>
<td>Photo Tagging</td>
</tr>
<tr>
<td>C</td>
<td>Read Only</td>
<td>100% Read</td>
<td>Zipf</td>
<td>User Profile Cache</td>
</tr>
<tr>
<td>D</td>
<td>Read Latest</td>
<td>95% Read/5% Insert</td>
<td>Latest</td>
<td>User Status Updates</td>
</tr>
<tr>
<td>E</td>
<td>Range Scan</td>
<td>95% Scan/5% Insert</td>
<td>Zipf</td>
<td>Threaded Conversations</td>
</tr>
<tr>
<td>F</td>
<td>R/M/W</td>
<td>50% Read/50% R/M/W</td>
<td>Zipf</td>
<td>User Database</td>
</tr>
</tbody>
</table>

shared counter example, a special increment operation can be created that atomically increments the value of a counter without the extra step of reading it first. To add a new extension, a special handler needs to be added to Innesto servers which implements the desired functionality given the partition, operation type and optional data. Together with operation transactions, extensions bring more of the functionality of relational databases to the cloud. Listing 4.1 presents an example of combining the extended API features to acquire a lease.

4.6 Evaluation

We evaluate the performance of our implementation of Innesto (roughly 32,500 lines of C++ code) using the Yahoo! Cloud Serving Benchmark (YCSB) [70], an industrial benchmark used to measure the performance of cloud storage systems. YCSB consists of 6 different workloads operating on a single table using different data access patterns which mimic real-world cloud applications (Table 4.3). YCSB also provides a low-level extensible storage interface which allows direct comparison of performance with different storage systems. We compare Innesto’s performance to that of Cassandra [94], a leading industrial key/value datastore, and HyperDex [78], a new second-generation datastore.

We use a set of 9 machines each equipped with two Quad-Core Intel Xeon (E5506) processors with 32GB of RAM and 12 TB storage running unmodified 64-bit Linux version 3.2.0-48. On the machines we deployed Cassandra version 1.0.12, HyperDex version 1.0.rc4 and Innesto. Eight machines formed a cluster
4.6.1 Key/Value Performance

We first evaluate the average throughput of all three systems. In each run, we load the system with 10 million items (the ‘Load’ phase). YCSB items by default have 10 attributes with a total size of 1KB. Five of the workloads operate on keys, while workload E searches for items based on record number. All three systems were configured to tolerate 1 server failure and use their default consistency configurations, which means Cassandra provides eventual consistency while HyperDex and Innesto provide strong consistency. HyperDex was set to be searchable and Innesto was configured to use one Search Clone partitioned based on record number and use network replication. Unless stated otherwise, Innesto uses lock mode isolation. Figure 4.4 shows our results.

Cassandra represents the range of performance cloud-scale applications require from the underlying storage system. Any new system, regardless of the new features it has to offer, should at least provide competitive performance with Cassan-
HyperDex represents the only other key/value datastore that can provide the same search capabilities as Innesto. Thus, we wish to see how Innesto performs compared to each system, where one sets performance requirements and the other sets expected features.

In Figure 4.4 we can see for both write heavy workloads (the first three bars) and read heavy workloads (middle three bars) Cassandra provides an average throughput of 30K-45K op/s due to its weak consistency model. A write request is written to disk on the receiving server (not necessarily the target server) and the client is notified immediately, while the operation is slowly routed to its destination server in the background. Additionally, read requests don’t need to wait for the latest version of an item to arrive, yielding high overall throughput for Cassandra. However, Cassandra performs poorly for scan operations (right-most bar) which is 20x slower than other workloads. On the other end, HyperDex’s search throughput outperforms Cassandra’s scan by a factor of 7x, due to its data arrangement, but performs poorly for write-heavy workloads since write operations are chained serially in its value-dependant chain. Thus, HyperDex only performs well for read-heavy workloads. Innesto’s operations are designed to compete in the minimum possible number of network RTTs and update Clones in parallel. Thus Innesto is able to outperform HyperDex’s write and search performance, while having competitive performance in the other workloads compared to Cassandra. Innesto has an order of magnitude (10x) higher throughput for search compared to Cassandra.

4.6.2 Load Balancing

Load balancing is an essential requirement for large-scale cloud applications which is why most key/value stores (e.g., Cassandra) use hash based routing for statistical load balancing. Innesto performs its own load balancing in order to preserve its partition hierarchy which is used for search. We wish to see how Innesto performs as the data stored in a table grows. In the Load phase, YCSB inserts items into the same table. We measure average overall throughput every time half a million new items have been stored. Figure 4.5 shows our results. As expected, Cassandra is able to maintain a steady throughput achieved through statistical load balancing. HyperDex uses a central coordinator to partition a table when it is created. Thus
Figure 4.5: Temporal throughput in Load phase of the YCSB benchmark. Notice how HyperDex’s performance decreases with dataset size while Cassandra and Innesto maintain a steady throughput.

as a partition hosts more data, performance decreases. Innesto consistently splits congested partitions and distributes them evenly between all the servers which is why it can maintain its high throughput as the data is growing. Innesto’s load balancer amortizes the cost of maintaining a partition hierarchy.

4.6.3 Clone Cost

To illustrate the costs of adding search capabilities to Innesto, we configure Innesto to run YCSB without any search capabilities (MVS only), and compare its performance when it supports search with a single Clone. We also measure performance when configured to use a Sparse Clone.

Figure 4.6 shows that adding Search Clones decreases throughput almost in half for Load. Without Clones, all Innesto transactions are designed to complete in one network RTT, providing an extremely high key/value throughput. With the addition of Clones, transactions fall back to two RTTs required by two-phase commit. Using Sparse Clones has more or less the same performance as using full Clones for all workloads except Workload E which uses search. The extra RTT required for fetching search results from MVS contributes to a 30% decrease in
search throughput.

4.6.4 High Dimensionality

YCSB item attributes are not searchable for normal key/value datastores. To measure the cost of high data dimensionality, i.e., data that is searchable on multiple different attributes, we vary the number of attributes stored per item (out of each item’s 10 attributes). A separate Search Clone is created per stored attribute to serve alongside the Clone based on record number. We investigate how Innesto and HyperDex perform with highly dimensional data. Innesto was configured to use Sparse Clones and we measure the throughput in the Load phase when inserting 5 million items.

Figure 4.7 shows that throughput decreases as the number of clones increases for both systems. With 8 Clones Innesto performs better than HyperDex with 3 since HyperDex chains operations serially while Innesto updates the Clones and MVS in parallel. With 11 clones Innesto’s throughput drops to 24% of not having any Clones. This is due to the overhead of adding extra items to each transaction multiple times (approximately 150 bytes per extra Sparse Clone). However, such a system can efficiently answer 11 different types of search requests.
4.6.5 Lock vs. Clock

In this section we evaluate the performance cost of locking data items compared to the clock-vector isolation method which provides the same strong consistency guarantees. We configure Innesto to use both locking as done in all experiments so far, denoted as Innesto-Lock, and clock vectors, denoted as Innesto-Clock. We repeat the experiments of Section 4.6.1, running all YCSB workloads in each mode as seen in Figure 4.8.

Compared to locking, using clock vectors reduces throughput by approximately 31.8%. In clock mode, each server maintains a virtual time counter and refrains from processing a transaction (and all others ordered after it) until its timestamp has been finalized. Thus, even though a transaction may be ready to execute, it might be delayed by another potentially non-conflicting transaction which is still pending on a final timestamp. This creates false sharing between unrelated transactions on the same server. In contrast, locks are held on item granularity to avoid any kind of false sharing. Also, the clock mode commit protocol forces multi-server read-only transactions to complete in 2 RTTs, compared to their 1 RTT execution time in lock mode. This reduces search performance.

Lock based systems are known to start thrashing as lock contention increases.
Figure 4.8: Average throughput for YCSB benchmark for Innesto in lock mode and clock mode. Compared to lock mode, clock mode has on average 31.8% lower throughput.

To trigger lock conflicts, we modified Workload A’s access distribution to be hotspot. In the hotspot distribution, given a number of items, $P\%$ of all operations only reference a fixed subset of $H$ hot items. The client uses a thread pool to issue concurrent requests. The ratio of $H$ to the size of the thread pool represents the probability of different threads accessing the same item. With 10 million items, we set $H$ to be one fifth of the thread pool size and varied the probability $P$ of accessing a hot item. Thus with a high enough $P$, threads are guaranteed to conflict with each other as seen in Figure 4.9.

In lock mode, lock contention increases with $P$, degrading performance. Under contention, three major sources of delay contribute to low performance. First, transaction messages could be received by the Participants in different order. This causes partial lock conflicts which will result in the locks being released and the transactions being retried, adding extra RTTs. Second, under lock conflicts transactions wait an exponentially weighted random time before retrying to avoid deadlocks. Third, all locks should be acquired on all servers before a transaction can proceed. Thus, the transaction has to wait for the slowest server to keep up. Note that each server could process other transactions while waiting, but the processing
time of each conflicting transaction increases. On the other hand, in clock mode transactions are globally ordered regardless of what items they touch. No matter what the access distribution, overall throughput is more or less the same.

Finally, to investigate the effects of the size of hot items, we varied the ratio of hot items $H$ to the thread pool size with a hotspot access probability of 95%. In Figure 4.10 we see that with a low ratio, lock-based throughput drops to around 1.5K op/s while clock-based throughput remains almost unchanged. As soon as the ratio increases over 1 which represents a very low probability of conflict, lock-based throughput is restored close to its maximum value.

Our experiments in this section illustrate the two ends of the performance spectrum offered by Innesto. Lock mode generally has better performance and offers more features but overall throughput depends on the application’s data access patterns. Compared to lock mode, clock mode doesn’t support generic conditional operations, but offers performance independent of the type of workload. The important thing is Innesto servers can operate in both modes on the same data and dynamically switch modes at run time.

**Figure 4.9:** Average throughput of Innesto in lock mode and clock mode using the hotspot distribution. The $x$-axis shows the probability of accessing a hot item.
Figure 4.10: Average throughput of Innesto in lock mode and clock mode using the hotspot distribution with a fixed access probability of 95% with varying number of hot items.

4.6.6 Summary

Our results using micro benchmarks show that for basic key/value operations Innesto performs as well as Cassandra, while having superior consistency guarantees and an order of magnitude better performance for search while providing a much richer API. Innesto scales better than HyperDex with dataset size and performs better for write heavy workloads and for data with high dimensionality in every aspect.

4.7 Related Work

4.7.1 Cloud Storage & Key/Value Datastores

Traditional key/value datastores rely on two major classes of architectures. First are the Chord [112] ring-based architectures. Amazon’s Dynamo [76] which serves as the basis of many cloud storage systems like Cassandra [94] is an example. It provides a schema-less data model with the main focus on availability by providing eventual consistency via a limited interface to the data. The second class are based on partitioning. BigTable [65] (and its open source implementations HBase [8] and
HyperTable [22]) partition a table into tablets. A tablet is a group of rows in a table with predefined bounds indexed globally where consistency is only guaranteed within the same row. All systems primarily operate on item keys via the basic key/value interface. Consistency is relaxed or limited to single items. While Innesto also uses partitioning, cross-partition data consistency is guaranteed via a much richer interface. Spinnaker [108] uses Paxos over a tablet-based storage system for high performance consistency. However, the focus is on replication not supporting search.

Cassandra [94] supports secondary indexing for multi-attribute range search and uses brute force to filter out results of a single-attribute scan. Our evaluation shows that Cassandra’s scan is an order of magnitude slower than Innesto’s search. Newer cloud systems like PNuts [69] have moved toward using tablets to provide semi-relational interfaces. PNuts also supports single attribute range scans in the form of iterative traversal of items. However, the result of the scan does not provide the same level of consistency that Innesto search does. Megastore [51] moves beyond that to bring more relational features to NoSQL. Data is partitioned into entity groups where consistency is guaranteed within each partition but has relaxed consistency across them. ACID transactions are supported within an entity group and are heavily used in Google [20], evident that modern cloud applications need transactions no matter how limited their functionality. Innesto supports cross-table cross-partition transactions with high performance, unlike Megastore that has low write performance across partitions.

HyperDex [78] is a recent feature-rich system that marks a second generation of key/value datastores. It uses value dependant chaining to serially execute transactions. This forces write operations to complete in multiple network RTTs. Innesto’s operations are designed to complete in the lowest amount of time possible. HyperDex’s central coordinator partitions each table statically when a table is created. Modern cloud applications have varying load over time which leads to high load fluctuation in HyperDex. Innesto servers continuously monitor their load and shift partitions accordingly to adapt as access patterns change. Warp [44] is HyperDex’s latest addition that allows it to support ACID transactions. However, still being based on chaining, transactions are executed serially with declining performance as the number of operations within a transaction increases.
4.7.2 Transactional Datastores

Amazon’s SimpleDB [6] added the support for conditional operations to provide some sort of simple transactions, another indication that cloud applications need a form of transactions. The B-tree [48] based on Sinfonia [49] uses one-round transactions to implement a $B^+$-tree which supports single-attribute range search. Due to false sharing, the B-tree has very bad write performance and can only index small (couple of bytes) values. Minuet [111] enhances the B-tree’s write performance and allows creating versioned snapshots for what-if analysis. Still, Minuet can only index small values unlike Innesto that performs well when storing arbitrary sized values for write-heavy workloads. Augustus [105] partitions data and uses two-phase commit between partitions to provide Byzantine fault tolerance (BFT), but the partitions are not meant for search. The commit protocol is very similar to Innesto’s one-round transactions with the addition of BFT. Spanner [71] is a globally distributed database that uses Paxos for consistency, designed for georeplication not high dimensional data. Like Innesto, all systems can support ACID transactions but neither support multi-attribute search.

4.7.3 Peer-to-Peer (P2P)

A large body of work exists for supporting multi-attribute search in P2P using network overlays. Arpeggio [68] uses a Chord ring and artificially inserts extra items for secondary indexing which hold lists of keys having the secondary index. Thus it can efficiently support \texttt{get} on exact secondary attributes but not on ranges. CAN [109] uses spatial partitioning to split a multi-dimensional key space and assign each one to a single server. CAN maintains no hierarchy and messages are routed toward the destination in multiple hops. MAAN [62] uses a Chord ring with an order preserving hash function. Range searches are mapped to consecutive ranges on the ring where the result is the intersection of multiple different searches. Mercury [57] creates multiple order-preserving Chord rings and a multi-attribute search is the intersection of multiple single attribute searches on separate rings. SkipIndex [126] indexes an $N$-dimensional key space using skip graphs (i.e., $N$-dimensional skip lists). It uses a partition hierarchy to execute range search. MURK [82] uses KD trees to partition the key space based on load. It uses space-
filling curves to map a multi-dimensional space to a single dimensional one when constructing skip pointers between partitions.

While these systems support search, their main focus has been routing in P2P networks. Operations generally execute in multiple steps with no consistency guarantees. Innesto explicitly uses transactions to ensure a correct global serialization of operations. Also, they partition based on all secondary attributes which does not scale with high dimensional data. Innesto supports efficient range searches by allowing an application to select what types of partitioning should be performed on a table. Innesto also allows a table to be optimized for different types of range searches.

4.7.4 Distributed Transactions

Efficient lock-based one-round transactions were introduced in Sinfonia [49]. Sinfonia exposes an unstructured linear address space. Multiple different applications [48,111] built over Sinfonia illustrate the flexibility of one-round transactions. However, Sinfonia memnodes have very limited functionality which forces bad design decisions [48,72]. Innesto’s transaction layer was inspired by Sinfonia but replaces the linear address space with flexible data types, allowing Innesto to directly intervene in the commit protocol. For example, Innesto’s split takes advantage of this to efficiently execute in a single RTT which would’ve taken multiple RTTs if directly implemented over Sinfonia.

Our implementation of Innesto’s clock mode is based on Granola [72] which uses clock vectors to isolate one-round transactions in a lock-free manner. Our evaluation shows that Innesto’s locking protocol starts to fail under high contention, while Granola does not suffer from such problems. However, since Granola transactions are executed some time after voting has finalized without any locking, Innesto cannot support generic conditional operations as part of its extended API in clock mode. Also, in the voting phase of Granola, servers directly communicate with each other and exchange $O(P^2)$ messages for a transaction with $P$ participants. With a high number of Search Clones this could become very expensive for Innesto. Additionally, since Granola uses a single counter per server, it causes false sharing between operations touching different partitions. We consider mod-
ifying Granola’s voting mechanism to exchange $O(P)$ messages and maintaining item counters per partition to perform synchronization at a finer granularity and reduce false sharing between transactions as future avenues in pursuing this work.

4.8 Conclusion

We present Innesto, a second generation searchable key/value datastore that provides the ability to efficiently search stored items based on their secondary attributes. It provides strong consistency with a transactional interface and can tolerate group failures of underlying machines. Innesto provides a scalable cloud-based solution to applications that cannot tolerate traditional limitations of NoSQL. Innesto uses an efficient implementation of two-phase commit for high performance one-round transactions. Key/value operations are carefully designed to execute in the least amount of time possible. Innesto’s load balancer allows maintaining a partition hierarchy used to quickly locate data. Our evaluation of Innesto using an industrial benchmark show its performance competes with Cassandra while offering a new world of features.
Chapter 5

SPEX

5.1 Introduction

Millions of dollars are spent each year on infrastructures that host large-scale virtual environments (VEs), such as World of Warcraft [46] and Second Life [33]. The two common approaches taken to host multiple concurrent users in such VEs are *sharding* and *full partitioning*. Sharding entails running multiple parallel instances of the VE, each assigned to a different host machine. Users are then assigned to a specific shard and cannot migrate or interact across their boundaries [99]. Full partitioning, on the other hand, involves only a single instance of the VE partitioned into smaller regions, each handled independently. Users are then assigned to specific hosts based on their most recent locations and are transparently handed off to other hosts as they pass region boundaries [34].

In a sharded VE, user interactions are limited to those within the same shard. Though the VE may have a high concurrent user count (e.g., World of Warcraft has over hundreds of thousands of active users on record [124]), each user can only observe and interact with a small fraction of other users in the world. With full partitioning, the interaction range of a user is limited to a user’s current region, or at most the neighbour regions when nearing borders. Long range interactions (e.g., looking through a telescope to a distant location) are fundamentally unsupported. Most systems also do not utilize resources effectively due to their static partitioning. Since all regions should be available, all servers need to be online even though...
the user distribution across regions may vary widely by date or time. This results in over-provisioning or under-utilization of the resources [96]. While changing the partition mapping offline alleviates the issue somewhat, it is a manual task and disrupts service up-time. Static partitioning also poses a limit on user density, as the maximum number of users in a region is capped to what a single core can simulate (with tasks such as game logic calculations, visibility management, or physics). Thus, densely crowded locations, such as stadiums or exhibits, are inherently difficult to realize (i.e., there is a density limit by design). Finally, First-Person Shooters (FPS) require prompt message delivery (e.g., 150ms [67]) and long-range interactions (e.g., sniper rifles). Due to the high volume of state updates per second with complicated interrelationships, large-scale VEs today cannot support FPS games without either restricting user interactions or using a centralized component.

We are motivated to remove some of these restrictions with SPEX, an infrastructure that supports scalable spatial publish/subscribe (SPS) [86]. SPEX is designed for VEs with hundreds of concurrent users. Users are allowed to roam freely in the VE without any density limitations while being able to interact with any user or region at arbitrary distances. The main contributions of SPEX are the following. First, we design an architecture that de-couples game logic processing from interest management [101], so that message routing within the VE server cluster is done very efficiently with dedicated resources. Using SPS as the primitives for interest management allows interaction ranges to be highly flexible. The SPS service is also consistent and fault tolerant in its stored states, by using a distributed transaction provider [118]. This enables us to combine publish/subscribe (pub/sub) with spatial queries, two features traditionally treated separately (i.e., pub/sub is for transient messages, while queries are done on persistent states). Second, we combine dynamic spatial partitioning with scalable pub/sub (also known as application-layer multicast) [64] for adaptive load balancing. This provides a two-level progressive mechanism to assign loads to different hosts, and to deliver more messages should the density level require. This eliminates the density caps of users in the VE, while balancing system operation costs. Third, by adopting a fully distributed architecture, we practically scale our implementation of SPEX to tens of hosting servers, and support 750 users with end-to-end latencies within 100 milliseconds over continental distances on the Internet.
In this chapter, we first present a background on interest management and SPS in Section 5.2. An overview of techniques used in SPEX is described in Section 5.3, and Section 5.4 describes how SPS is performed. Section 5.5 presents implementation details where use cases of SPEX can be found in Section 5.6. We evaluate our implementation in Section 5.7, compare to related work in Section 5.8 and conclude in Section 5.9.

5.2 Virtual Environments

A VE simulates an environment in which a group of actors move and interact with it based on a set of clearly defined rules, such as physical laws. Each actor encodes its behaviour in a series of events, and runs a local version of the simulation based on its observation of the last known states of the VE. Changes to the states of the VE are sent to the actors periodically via update messages. When scaling the number of actors in the VE, communicating the latest states of the VE to the actors becomes a challenge since the number of messages exchanged grows quadratically with the number of actors. Interest management [101] is a well-known technique to address this problem based on the observation that while the VE may be arbitrarily large, an actor has a bounded interest range, i.e., its Area of Interest (AoI) [104]. An actor’s AoI may have any possible shape and includes all the locations in the VE of interest to the actor. Actors ensure that the VE has knowledge of their latest AoI, while the VE ensures that the actors are up-to-date on the state of the VE inside their AoI.

While the concept of AoI is widely known in VE research, there are cases where it may not fully capture the requirements of a VE. For example, while an actor’s visibility or field of view can be supported by having an AoI in the VE, when a task requires making an effect on an area (e.g., casting a fire spell burning everything within a radius), the concept of Area of Effect (AoE) [114] may be more relevant. Combining both the concept of AoI (i.e., a subscription of events) and that of AoE (i.e., a publication of events) yields the more general concept of SPS.

SPS [86] is defined as the fundamental set of functions required to fully handle interest management in a VE. SPS mandates that an actor have the freedom...
to specify a *subscription space* and a *publication space* of arbitrary shape. Any message sent to a publication space should be delivered to all overlapping subscription spaces. From a high level perspective, most existing VE functions can be observed as special cases of SPS functions. For example, common VE operations such as chat, trade, attack, etc., can all be built by knowing neighbours spatially, or affecting other entities within a spatial radius. The significance of SPS thus is that an architecture supporting SPS can be the basis for any kind of VE. Hence, if scalable SPS becomes possible, so will a VE with a large number of concurrent users. In this context, SPEX is a novel approach to large-scale VEs by supporting SPS functions in a scalable and consistent manner.

### 5.3 Overview

Figure 5.1 shows the basic architecture of SPEX. We employed a cloud architecture consisting of *SPEX hosts*, a group of machines cooperating with each other to provide a unified and fully distributed service. With a cloud architecture, we are not concerned with bandwidth usage between the hosts, and CPU resources can be
added on the fly. Additionally, network round-trip times (RTTs) between them are in the order of milliseconds, allowing state stored on other hosts to be retrieved in a short amount of time when required.

SPEX consists of three main components: *proxy*, *partitioning* and *multicast*. Each component is fully distributed, i.e., multiple SPEX hosts form each component. Also, a single SPEX host can be part of multiple components, e.g., be a proxy host, partitioning host and multicast host at the same time. No host ever acts as any kind of coordinator, nor is one required to maintain any form of global state.

The proxy is the frontend of the system which directly communicates with users. A proxy host serves a subset of the users and acts on their behalf in the VE. Each user sends its commands (in the form of events), such as movements and interactions in the VE, to its assigned proxy. The proxy handles all necessary functions communicating with other components and sends the required information to the user in the form of regular updates. A user has no strict affinity to a specific proxy host and can be freely handed-off from one to another. This is essential to evenly distribute users between hosts.

SPEX is designed to practically scale to tens of hosts to support hundreds or even thousands of users in a single VE. SPEX’s primary goal is to overcome the limitations of sharding and full partitioning. To this end, we make no assumptions about user behaviours and impose no restrictions on how densely they can cluster. Note that this does not mean SPEX will perform under any circumstances, but will have a graceful performance degradation under heavy load rather than a complete break down. SPEX supports scalable SPS by providing a more generic API compared to traditional VE architectures. The API does not limit how users can interact with other users or parts of the VE, no matter how far apart they are.

SPS functionality is provided by using partitioning to quantize publication and subscription areas with arbitrary shapes into meaningful sub-regions of the VE. Pub/sub messages are forwarded to the hosts responsible for managing the target sub-regions. To correctly deal with potential consistency issues that could occur due to quantization and message re-ordering on different hosts, SPS functions are each executed using a separate one-round transaction which ensure correct global ordering of events across hosts. Due to unrestricted user behaviours and interactions, potential hotspots (caused by flocking) and over-congested regions are in-
evitable. SPEX avoids hotspots as much as possible by increasing the quantization granularity. However, when hotspots are impossible to avoid, their effects are minimized by utilizing additional resources as much as possible to multicast relevant updates. In the remainder of this section we elaborate on the details of how each task is done.

5.3.1 Dynamic VE Distribution

Spatial partitioning is a widely employed technique where the VE is split into a grid of fixed-sized cells, triangular strips or hexagons [33, 85, 91]. Each partition represents a VE region handled by a specific host identified by a partition mapping. Partitioning reduces the workload by having actors only interact with partitions that overlap their AoI, but suffers from two major limitations. First, selecting an optimal size for the partitions is challenging: a large size will result in a mismatch when mapping AoIs to regions, causing extra overhead when forwarding irrelevant events to users; while a small size will result in an excessive number of regions, inducing more load per actor operation. Second, when using a static partition mapping (e.g., computed offline using a hash function), there is a high workload variation across highly popular partitions (hotspots) and under-utilized ones.

SPEX avoids these limitations by using progressive partitioning combined with a dynamic partition hierarchy. With progressive partitioning regions are partitioned based on the density of events within them. If the activity in a region exceeds a threshold, partitioning is refined by splitting the original partition into smaller ones. Conversely, inactive partitions are merged to reduce the total number of regions. This causes the number of regions to be proportional to the distribution and frequency of events, regardless of the VE size. Figure 5.2 presents an illustration of a partitioned VE using both full partitioning and progressive partitioning based on a quad-tree [80]. Each dark spot represents a source of events. With full partitioning, the number of partitions is inversely proportional to the region size regardless of the event distribution. For a large VE this could result in millions of partitions. On the other hand, with the same event distribution, progressive partitioning distributes the events more evenly between fewer partitions, and avoids
The key enabler for SPEX’s progressive partitioning is its dynamic partition hierarchy, which distributes partitions between the hosts. Each partition maintains a pointer to its parent and child partitions in the form host_id : partition_id, where host_id uniquely identifies the IP and port of the host and partition_id identifies the partition. Figure 5.3 depicts how a VE progressively partitioned into different regions form a hierarchy, and how the hierarchy is distributed between four SPEX hosts.

To publish an event in the VE, a host finds the most suitable partitions for the publication and forwards the event to them. The root of the hierarchy encloses the entire VE and is stored in a specific host known to all others (this does not constitute a single point of failure due to replication as described later). The rest of the partitions can then be located by traversing tree links starting from the root in logarithmic steps. In practice, upper layers of the hierarchy tend to change infrequently. Thus each host optimistically caches these parts of the hierarchy, providing replication. This improves performance by reducing the communications needed among hosts when locating a partition. However, the structure of the lower parts is constantly changing. These changes are lazily propagated to hosts to avoid excessive replication overheads. If a host tries to publish an event using a stale version of the hierarchy, the publication is rejected and the host is told which parts of the hierarchy have changed. The host will then have to update its cache before retrying.

The distributed hierarchy provides a scalable way to locate partitions with low hotspots as much as possible.

Figure 5.2: (left) Full partitioning vs. (right) Progressive partitioning.
5.3.2 Consistency & Fault Tolerance

An important issue that threatens the practicality of a distributed VE is consistency. While some inconsistencies may go unnoticed by users or only somewhat degrade its enjoyability, others endanger correctness and operability of the system. Two major sources of inconsistency exist in a distributed VE. First are inconsistencies in the VE states caused by conflicting interactions between actors. For example, if two or more actors try to perform the same operation, like picking up the same item or shooting at the same actor, there needs to be a strict and deterministic mechanism to reject all but one operation. Second are inconsistencies in the infrastructure caused by conflicting operations by the hosts due to incorrect assumptions. As an example, consider the scenario where one host is using a stale version of the partition mapping (i.e., a wrong assumption) to publish events to partitions that have previously been split into smaller ones or deleted/merged by other hosts. Such an operation should cleanly and completely be dealt with without any side effects.
Even when the assumptions are correct, a more subtle form of inconsistency arises when a single event triggers a host to send messages to multiple other hosts. Since there is no guarantee on what order each host will process the different messages it receives, inconsistencies could result due to this lack of consistent event ordering. Consider when host $S_1$ needs to send messages $M_{S_1A}$, $M_{S_1B}$ to hosts $A$ and $B$, respectively. At the same time, host $S_2$ also needs to send messages $M_{S_2A}$, $M_{S_2B}$ to $A$ and $B$. With simple message passing, there is no guarantee that $A$ and $B$ will both either see $S_1$’s message before $S_2$’s message, or vice versa. They might see them in different order, which means that they will process them in different order. Inconsistencies would show if the messages are in conflict, e.g., trying to pick up the same item as explained earlier, as each server would award the item to different players. This situation aggravates as the number of recipients increases.

We note that distributed VE systems targeting on scalability often do not explicitly deal with consistency. In the case of partitioning and sharding, interactions are limited to the current partition or shard, so consistency is implicitly ensured by design through the sole owner of the partition or shard. Other systems that try to provide a single VE instance, e.g., EveOnline [17] or Pikkotekk [27], ignore event ordering issues, but events also do not trigger message delivery to multiple servers (i.e., AoE is limited). Hence they avoid the second form of inconsistency. SPEX is designed to surpass the limitations of similar systems. Interactions are not confined to a single shard or partition, nor are AoEs restricted (via the support of generic area publications). Therefore, we carefully built in a consistency layer in the architecture of SPEX which explicitly deals with all forms of potential inconsistency.

SPEX addresses consistency by using a distributed transaction provider, a light-weight wrapper around the transportation layer that acts as a mediator between the hosts. It prevents inconsistencies from propagating into upper layers. A standard API for a transport layer is to provide a means to “send message $M$ from sender $S$ to receiver $R$”. At some point afterwards, the message is delivered to a message handler on receiver $R$. The transaction provider provides a similar yet more powerful interface. First, it allows the sender to also state any assumptions they might have about the message along with it. The message is only delivered if the assumptions hold at delivery time. Second, it allows a compound message to
have multiple recipients. That is, a message can consist of smaller messages, each of which is delivered to a different receiver. Finally, it ensures atomic and globally ordered delivery of the message. All recipients will receive the messages in the exact same order, or not at all. In the case of delivery, the message is delivered to the same message handlers of the receivers. The transport API offered by the transaction provider is hence transformed into “send message $M$ from sender $S$ to receivers $R_1,...,R_n$ only if conditions $C_1,...,C_k$ hold”.

Using the distributed transaction provider, the actor conflict example is resolved by having an actor to send the message “pickup item only if it’s still available”. The host conflict is also resolved by each host publishing events like “publish event to partition only if it hasn’t varied”. Rejected messages are returned to the sender which should take appropriate actions. An actor failing to pick up an item knows that it should not add the item to its inventory, and the host realizes that it should update that part of its cache. However, the benefits offered by the transaction provider are not free. They come at the cost of potential extra delivery time caused by the complexity of the underlying protocol and synchronization overhead. Nonetheless, the order of extra delay is that of a network RTT, at most a few milliseconds in SPEX’s architecture.

The distributed transaction provider may also provide protection against failures. Since all important communications are transmitted through it, each host may be seamlessly backed up with a number of mirror hosts. The transaction provider will ensure that backups are in perfect sync with the primary. In the event of the failure of a primary, a backup will take over and operations will resume. This obviates the need for redundant failure protection mechanisms in other components.

5.3.3 Adaptive Load Balancing

The load on a VE infrastructure drastically varies over time. The total number of users changes with time, and users tend to cluster in different locations based to current events [96,99]. Having the ability to shift load between hosts is essential for both scalability and cost efficiency. Since the processing power of any given host is limited, when a host’s load reaches a limit, it should be redistributed. Conversely, when the overall capacity of the hosts exceeds the total system load, loads should
be shifted away from under utilized hosts so they can be safely terminated to reduce costs.

We classify load into two different components. First is distributing a large number of users between hosts. With thousands of users, tens of thousands of update messages arrive at SPEX every second, and it might stream millions of messages back. We handle this by distributing the users between proxy hosts, and use progressive partitioning to reduce the quadratic communication complexity with the number of users by forwarding events to a bounded number of relevant partitions. When a partition’s load exceeds a threshold, the partition is split into smaller ones. The load balancing component ensures that the new partitions are distributed between different hosts to spread the load as evenly as possible. The dynamic partition hierarchy allows the partitions to have no affinity to a specific host, so that partitions may fluidly migrate between machines. This is especially useful when the total load of multiple partitions located on the same host exceeds its capacity, yet neither is individually high enough to be split and redistributed.

The second load component occurs when no amount of partitioning can reduce the work load to match a host’s capacity. A good example is the ice rink in a hockey arena which is a single over-important hotspot that most of the actors publish or subscribe to it. Another example is the flag carrier in a capture the flag match [55]. This form of load is why partitioned VEs limit the maximum actor density in the VE. SPEX handles this by reinforcing the partition hierarchy with scalable pub/sub.

Scalable pub/sub (e.g., Scribe [64]) has been used for hosting distributed VEs in a myriad of different architectures [91][104], where a topic (or channel) represents a source of events. Scalability is achieved by allowing multiple hosts to jointly disseminate a topic’s messages by forming a reverse forwarding tree for each topic. This allows more hosts to help forward events to a subset of the recipients, as opposed to one host forwarding to them all. However, it comes at the cost of extra network hops per publication (another reason we employed a cloud architecture to minimize these costs). However, scalable pub/sub is not suitable for hosting a VE on its own. It provides a discrete range of topics where continuous AoIs and AoEs somehow need to be mapped to them, which is a non-trivial task. Additionally, a VE based on pub/sub alone has difficulties answering spatial
queries, and inconsistencies may result from conflicting publications with nothing there to prevent them.

SPEX already handles consistency issues and has a partitioning component that maps areas to partitions. Thus scalable pub/sub fits in nicely with these components. Each partition is associated with its own topic which is created and destroyed along with it. Any events that occur in the partition are published to and disseminated by the topic. As we will see in our evaluation, adaptive load balancing plays an important role in hosting regions with densely clustered actors.

5.4 SPS Functionality

Proxy hosts receive events from users and act on their behalf in the VE. They deal with player churn and maintain enough player information to be able to translate application events into SPS functions as described next.

5.4.1 Spatial Publications

Publishing an event requires finding a list of partitions that intersect the area of the publication. For point publications the target will be a single partition which is the lowest one in the hierarchy (a leaf in the tree). For area publications the host must publish to the set of all leaf partitions that intersect the area. Publishing to non-leaf partitions would create parallel paths for publications, i.e., publishing to either of the ancestor partitions of a leaf. The transaction provider cannot detect the dependency between publications on different paths and would lead to inconsistencies. Thus we do not allow publications to non-leaf partitions.

Partitions are found through their parents using the partition hierarchy. Results are cached for future publications to increase locality. Once found, a one-round transaction is used to write the publication to the partitions. The transaction provider handles synchronizing messages on different hosts. When the transaction is executed, the event is written to the topics associated with the partitions involved in the publication, which disseminate it to other hosts interested in the event. These hosts in turn forward it to appropriate users.
5.4.2 Spatial Subscriptions

Proxy hosts are responsible for interest management of their users. They handle user subscriptions by maintaining a per-user subscription list, a list of all topics subscribed to by the respective user. The host then manages its own subscription list as the union of all the subscription lists of its users. It subscribes to each topic inside its subscription list. Once a publication is received for a topic, all subscribed local users are then forwarded the event.

The unit of subscription is a partition. Subscribing to a partition’s topic is done using the partition hierarchy. A subscription list is populated by intersecting the subscription areas with partitions. This is done recursively. The subscription process starts at the root partition of the VE which covers the entire VE space. For each non-leaf partition, subscription is then refined by also subscribing to intersecting child partitions, while maintaining the subscription to current partitions. This allows hosts to have no global view of the VE but to quickly discover unknown parts as needed. As actors move in the VE their AoIs change. Modifying a subscription is done by computing the difference between the subscription areas of the old AoI and the new one then subscribing/unsubscribing to the topics accordingly.

Modifications to the partition hierarchy as a result of splitting or merging partitions are propagated to the subscribers in the form of special events. When a partition is split, the event will cause subscribers to refine their subscriptions by additionally subscribing to newly created partitions. For merge, new publications will arrive on the parent partition’s topic. Recall we do not allow publications on non-leaf partitions. This is a signal that child partitions have been destroyed and the partition is now a leaf. Upon receipt, subscribers to the destroyed child partitions will unsubscribe from them.

5.5 Implementation Details

5.5.1 Spatial Partitioning

Our spatial partitioning component uses Innesto [117]. Innesto is a distributed key/value store that uses spatial partitioning. We use an event’s spatial coordinates as its key and the actual event as the value. Innesto supports keys with any number
of dimensions so it can support a VE with any number of dimensions (e.g., 2D, 3D, etc.). We added a publish function to Innesto. Publish uses an event’s key to compute the partitions that should receive the event and delivers it to each partition’s publication handler.

A publication handler is hooked to the root of a topic in the scalable pub/sub component which handles pub/sub delivery. The publication handler monitors the rate of events published to its partition and decides when to split the partition. It also handles publishing special events upon structure modifications. A partition is versioned with a modification counter that is incremented each time a split or merge occurs. The version number is used to identify publications that have used stale cached entries of the distributed partition hierarchy. The publication is rejected if its provided version number mismatches the latest value. This protects the partition hierarchy from race conditions.

5.5.2 Scalable Pub/Sub

The scalable pub/sub component implements Scribe [64]. The hosts form a ring based on their unique host ids. Each partition serves as a different topic. The basic functionality is to subscribe/unsubscribe from different partitions. All events that occur in a partition are published to its topic. Subscription requests are routed across the ring with hops of exponentially decreasing size until they reach either the destination or an intermediate host that already has a subscription to the topic. When an event is published, the publication is forwarded along the reverse path the subscriptions came through. This will use more hosts to help forward the events, which scales as the topic becomes more popular.

5.6 Discussion

One important ability of SPEX is to decouple publications from subscriptions (i.e., AoEs from AoIs). In most existing systems [33, 56, 83], the AoI of a user is tightly coupled with its position in the VE, i.e., a user publishes its location and its AoI is assumed as the area surrounding its position in the VE. SPEX allows users to fully decouple these two. A user can publish to any location within the VE, while its AoI could also be on another region in the VE. This decoupling allows SPEX-based
applications to implement new features previously not possible. For example, a user could be observing a distant location through a telescope. It could also interact with the distant location by shooting at it with a long-range sniper rifle. We discuss SPEX’s specific use cases as follows:

(a) **Client/Server:** SPEX can replace the relay/messaging server in client/server architectures, e.g., FPS games. Instead of connecting to a single server, the players connect to SPEX hosts for interest management. Additionally, server-side arbitrators could be plugged-in to subscribe to regions and handle event processing based on game logic [86] (e.g., physics). Such an architecture could be run in a dedicated environment to provide epic-scale cloud-based gaming over a WAN [104]. SPEX can also be used for LAN games, e.g., a larger scale virtual city such as the game Grand Theft Auto, where not all actors are human controlled (Non-Player Characters or NPCs) and their total number exceeds what a single server can handle.

(b) **Peer-to-Peer (P2P):** P2P games distribute the VE between peer (i.e., client) machines [91], each acting as a server for parts of the VE. Each peer then has to find other peers hosting events of their interest (i.e., *neighbour discovery process*), which presents a major challenge in P2P gaming. SPEX could be used as an external service to augment existing P2P architectures. Peers can simply use SPEX for neighbour discovery. They will update their states in SPEX at a low frequency [55,104], providing a pointer to themselves (i.e., IP and port), and SPEX will find their neighbours for them. A peer will then directly contact found neighbours for higher frequency updates as it normally would in a purely P2P architecture.

One could also imagine using SPEX under a hybrid architecture. For example, some peers behind NATs may have problems allowing other peers to connect. Others might not have enough networking resources to serve peers. In a hybrid architecture, a peer is given the option to either serve game states itself, only storing a self pointer in SPEX for free, or to host all their states in SPEX and let SPEX serve the peers on their behalf, an option which may require a paid subscription.
5.7 Evaluation

In this section we evaluate SPEX. We emphasize that our evaluation is based on an actual implementation of SPEX running on real machines, not a simulation. We thus obtain results that are more realistic and could be of value to system developers. We use two evaluation platforms. Sections 5.7.1, 5.7.2, 5.7.3, 5.7.5 and 5.7.6 use the testbed setting: a set of 22 machines with Pentium IV 3200MHz processors and 1GB of RAM, connected with Gigabit Ethernet. Section 5.7.4 uses the EC2 setting: 60 High-CPU Extra Large Amazon EC2 instances [3], each having 8 virtual cores with 2.5 EC2 compute units of processing power and 7GB of memory to repeat some tests at a larger scale with more actors. Half of the machines in each experiment are used to host SPEX and the other half act as clients. Each client machine emulates multiple actors, allowing us to evaluate SPEX with a larger number of actors than the available machines.

The purpose of our evaluation is to measure four different aspects of SPEX:

- **Responsiveness** by measuring end-to-end latencies of event delivery.
- **Scalability** by investigating the effects of varying the total number of actors on resource usage.
- **Correctness** by measuring accuracy, a metric obtained by comparing SPEX’s AoI matching to that of an ideal system.
- **Stability** by measuring SPEX’s performance variance over long periods of time and non-uniform load.

We borrow evaluation parameters from one of the most popular large-scale VEs, Second Life [33]. Actors are randomly spread across the VE with AoIs of radius 64 units surrounding their position. An actor walks in the VE, updates its position and forwards its new position and updated AoI to its associated host. Actor positions are published as new events using point publications and each actor’s subscription list is updated as an area subscription. To provide a highly dynamic fast-paced VE, we use FPS-grade update rates for the actors to move and change their AoIs, i.e., 10 moves and AoI modifications per second.
Changing the dimensions of the VE allows us to control the density of the actors. In Second Life, a server is assigned a 256 × 256 square region which can support at most 100 actors \[34\]; 16 Second Life servers can host a VE of size 1024 × 1024 and 64 can support 2048 × 2048. Using our SPEX hosts, we create VEs with these three sizes, and vary the number of actors in the VE in separate experimental runs. We investigate two types of actor behaviours, random way-points and cluster movements. In random way-point, an actor selects a random destination in the VE and moves toward it, selecting a new one once it arrives at its current destination \[83,104\]. In cluster movements, the actors tend to visit certain hotspots more often than other places, causing them to cluster in the hotspots more densely \[86,87,99\]. We first start our evaluation with random way-points in Sections 5.7.1-5.7.4, then compare results to cluster movements in Section 5.7.5.

Each run lasts for 120 seconds \[86,87\] and the VE starts as a single partition. We clip out the first 20 seconds to capture 100 seconds of run time, thus we give SPEX’s load balancer 20 seconds to quickly partition the VE and balance it between the hosts. Each run is repeated 5 times to average out random actor placement effects. The duration of 120 seconds was selected to allow SPEX to stabilize, while keeping collected data traces within reasonable sizes. Nonetheless, we investigate SPEX’s performance variations over a full hour in Section 5.7.5. Finally, Section 5.7.6 presents results on synchronization overheads of using one-round transactions instead of direct message passing.

5.7.1 Processing Latency

The quality of experience (QoE) of actors in a VE is known to be highly sensitive to end-to-end delay. End-to-end delay is the time between when an actor causes an event until it is observed by others. Previous studies have found upper bounds of 150ms for FPS games on end-to-end delay as the border of enjoyability of the game. Anything above that starts to drastically degrade QoE \[67\]. In a breakdown of end-to-end delay, a message has to propagate from the user machine to a SPEX proxy host (network delay), spend some time inside SPEX being duplicated and forwarded to other proxy hosts (processing delay), and another network delay back to other user machines.
We measure end-to-end delay as the time it takes for an event to leave one user machine and arrive at another, and measure SPEX processing delay as the time it takes for one SPEX host to internally deliver a publication to others. Together, both measurements allow us to quantify how the actors observe events in the VE, and how much of that is due to the distributed architecture of SPEX. If SPEX processing delay is low enough, it leaves enough time for network delays while still adhering to the bound on end-to-end delay. To obtain these measurements, we synchronize the clocks on the evaluation machines using NTP. Each event message contains two timestamps. One corresponds to the time it leaves the user machine and one is when it arrives at SPEX. The timestamps are used to calculate a message’s delays. We present two characteristics of delay. Mean represents the mean delay computed for all messages in a run. We also compute an upper bound on the 99th percentile delay, measured as the 99th percentile of the maximum delays measured in 1 second intervals at each machine. The error bars represent minimum and maximum measurements across the runs of each setting.

Figure 5.4 presents our latency results for different VE sizes and different numbers of actors. Mean SPEX processing delays (not shown) are well under 4ms and the 99th percentile is an order of magnitude larger but still less than 50ms. As the
number of actors in the VE increases, the average number of actors within the AoI of any actor increases. This puts more load on the hosts which increases processing delay.

Our latency results are significant in three important ways. First, SPEX’s short mean delay allows us to use it for any type of application, even epic-scale FPS games, as it leaves ample time for network delays. Second, by using more hosts, SPEX can scale a single Second Life region to host more actors, enabling more congested areas. An interesting extension is for Second Life to keep its current partitioning method (i.e., by regular grids), but assign each partition to a separate SPEX instance rather than a single core, allowing a set of SPEX hosts to handle the interest management for a single Second Life region. Third, to host a VE of size 2048 × 2048 in Second Life, the operators would normally have to use 64 cores. In SPEX, the number of hosts can dynamically scale from 1 to 64 to match current demands. Thus, an operator can balance system cost and performance with SPEX.

The per-second update frequency of each actor directly impacts performance. More updates imply that actors publish and change AoIs more frequently, putting more stress on the system to deliver more events and change subscriptions more frequently. We thus investigate the effects of update rates on maximum SPEX processing delays with a map size of 1024 × 1024 in Figure 5.5. Increasing the update rate increases the 99th percentile worst delay at larger scales. With 20 updates per second, average worst case delays are close to 100ms with a mean below 3ms. The higher update rates push the host CPU to be fully saturated, increasing delay. We expect SPEX to perform better if more hosts were available.

5.7.2 Bandwidth

Physical resources available on each machine are limited. We need to ensure that a host’s bandwidth usage does not exceed what is available. We measure bandwidth as application throughput sent to and received by the machines. For an actor, it simply consists of the data stream uploaded to and downloaded from its associated SPEX host. A SPEX host’s bandwidth consists of data it communicates with all its actors and the data it communicates with other SPEX hosts.

Figure 5.6a and 5.6b illustrate the average bandwidth measurements for each
Figure 5.5: 99th percentile SPEX processing delay in the testbed with different update rates at map size $1024 \times 1024$.

host. Bandwidth grows linearly with the number of actors, since the VE’s states grow with the number of actors, and more events have to be delivered to more targets. This shows that SPEX’s interest management has reduced bandwidth growth from quadratic to linear. In a more congested VE, an actor is subscribed to more events than in an uncongested VE. As evident in Figure 5.6c which presents average actor download bandwidth, with a smaller VE the number of events delivered to an actor is significantly larger than that of larger VEs. However, the required bandwidth for the actors is within the range available to residential users. As for the hosts, part of their traffic traverses the internal network (LAN). The rest is streamed over the Internet, but does not exceed bandwidth available to commercial services.

5.7.3 Accuracy

In this section we quantify how close the correctness of SPEX is to a hypothetical ideal system to ensure that SPEX does not trade correctness for performance. In an ideal system, all events forwarded to an actor are correctly matched to its AoI. SPEX is a fully distributed system and various delays in the system could cause
Figure 5.6: SPEX host and actor average bandwidth in the testbed.
forwarded events to sometimes mismatch the latest AoI. We define accuracy as the ratio of the number of correct events the actors receive divided by total events they receive. A correct event is defined to be one that would be delivered by a hypothetical ideal communication system that operates with zero delay. Events that were delivered but should not have been, and events that should have been delivered but were not, count as incorrect events. Accuracy is the only way to measure the correctness of a distributed architecture to ensure no events are lost or incorrectly disseminated in the system.

There are two sources that contribute to inaccuracy in SPEX: event processing delay and subscription delay. With processing delay, an event arrives at one host and spends some processing time going through various components before being delivered to other hosts. The longer it takes for the event to reach its destination, the higher the likelihood that the target actors will possibly change their AoIs. Thus the event delivered in an ideal system might not be delivered in a system with processing delay. When an actor makes a movement and updates its AoI, its host immediately subscribes to topics that are in the new AoI and unsubscribes from topics that are no longer in the actor’s AoI. In practice, there is a delay for the pub/sub component to fix the reverse forwarding tree to include/exclude the host, something that would happen immediately in an ideal system. This is what we refer to as subscription delay. The longer it takes for subscriptions to be updated, the higher the likelihood that the actor might receive events from its old AoI but are not in its current AoI, or miss events that occurred in the new AoI before the new subscriptions take effect.

We compute server-side accuracy offline using two different timestamped logs stored by the hosts per actor: position log, the actor’s position in the VE, and an event log, any event that was ever delivered to the actor. Using the position log we fully reconstruct the global state of the VE at any given time. We use this state to find the ideal set of events each actor should have received (i.e., each actor should receive relevant events within 100ms). By comparing this ideal matching to the event log we identify any missing or incorrectly forwarded events (i.e., events that should not have been received or were not received) and compute accuracy. SPEX is streaming hundreds of megabits of data per second, and for accuracy we have to log every event for the entire experiment duration, which severely affects
performance. So we only compute accuracy up to 275 actors. Figure 5.7 illustrates the accuracy of SPEX. For all VE sizes and all numbers of actors accuracy is over 90%. When the VE is large enough SPEX is over 96% accurate. Only in the extremely small VE of $256 \times 256$ does accuracy drop to 91%. Recall from Figure 5.4 that processing delay starts to increase in this VE size. Accuracy highly depends on system delays, as confirmed by our results. Previous studies have shown FPS games can cope with loss rates of 5\%–10\% as long as events are delivered within 100-150ms [53]. Compared to studies on FPS games and to related work [55, 86], we consider SPEX’s accuracy to be high enough.

### 5.7.4 Cross-Continent Latency

We now report the experiments in EC2 using almost 3 times more SPEX hosts and client machines to test SPEX for VE scales never previously tested. SPEX hosts are located in Amazon’s US-EAST region (Virginia), while the client machines are located over 4000 kilometers away from the hosts in the US-WEST region (California). We found the base round-trip delay between the clients and the hosts using `ping` to be around 85ms. This distance allows us to capture the effects of communicating across a continent on end-to-end latency. Using 60 EC2 instances for multiple hours is costly, so only a VE of size $1024 \times 1024$ is used to re-run the
experiments in Section 5.7.1 with an average of 2 runs.

At the largest scale of 750 actors, mean end-to-end delays are under 91ms, enough to host an epic-scale FPS game. Note that there is a base 85ms delay from a client to a host and back to another client, meaning update messages spend an average of less than 6ms inside SPEX. Figure 5.8 illustrates various end-to-end delay percentiles of the worst case delay. Parts of these worst-case delays are due to over crowded regions in the VE, where the event is being forwarded between the hosts inside SPEX. The rest are due to unpredictable events in the Internet, e.g., TCP retransmits.¹ For example, with 750 actors, the 95th percentile of worst case delay is around 215ms. From an actor’s perspective, this implies that in 100 measured intervals, events over 215ms old are received only in 5 of these 100 intervals, and in the other 95 intervals all events are on time. From the host’s perspective, every second, out of 750 actors, only 37.5 actors (i.e., 5% of them) receive events older than 215ms. Based on our EC2 setup, the distance between the hosts and clients is large enough to cover North and Central America, and even parts of South America. Users in these regions should expect a quality of service that is at least as good as what we present. More distant clients would observe an

---

¹ All streams between SPEX hosts themselves and their clients use TCP to be safe to public networks, including EC2 and the Internet, respectively.
added delay (due to distance) compared to our results.

5.7.5 Stability

All our experiments thus far started with a single un-partitioned VE and pushed SPEX’s load balancer to quickly repartition the VE and distribute it between hosts. In this section we measure the robustness of the overall architecture, and validate our experiment settings to not bias in favour of SPEX. We would like to ensure that our experiments have fully and correctly captured SPEX in its steady state operation.

We start with 275 actors in the testbed setting, the largest setting possible to fully log the system states from Section 5.7.3, and re-run all experiments for a full hour to measure variance in SPEX for all world sizes. If we observe significant variance in the measurements over time, our experiments would have failed to capture the steady state characteristics of SPEX. Since the hosts are streaming hundreds of megabits of data per second, it is not possible to log everything for an hour. Hence, we periodically measure the system’s behaviour. For latency, every 2 minutes we compute SPEX’s mean and max processing delay as the average and maximum processing delay of all updates within the last 2 minutes, respectively. For accuracy, every 400 seconds we log all required information for 30 consecutive seconds, thus sampling SPEX’s accuracy approximately every 7 minutes. As Figure 5.9 shows, we found performance samples to be consistent over time with no significant changes. Within one hour, mean delay samples varied less than 0.5ms and accuracy less than 1%. Our results here confirm that evaluations in previous subsections have correctly quantified SPEX’s performance.

As a last step to validate our experimental setup, we investigate the effects of actors clustering in hotspots. In the 256 × 256 world size (already a congested area), we setup 1, 4 and 8 hotspots to represent locations in the VE where actors tend to investigate more often. This will cause regions around the hotspots to have a higher density of actors. With 1 hotspot, an actor is either at the hotspot and goes to another random location, or is not at the hotspot and elects to go to the hotspot 50% of the time. With 4 and 8 hotspots spread in the VE, an actor elects to go to a randomly selected hotspot with high probability (80%), otherwise
it goes to a random location [86]. We ran the experiments for a full hour. As shown by Figure 5.10, with 1, 4 and 8 hotspots we found the mean delay to rise up to 7-8ms. Maximum delay still varied mostly between 40ms-60ms and accuracy was still maintained at around 91-92%. Hence SPEX’s load balancer successfully managed actors clustering in the VE. We believe this clustering is one of the worst loads actors can impose on a VE. We expect SPEX to behave under other types of clustering (e.g., more/less hotspots, larger world size, etc.) with a response somewhat between that of uniform distribution and a single hotspot. Our results in this section validate the duration and load of our experiments, where we observe no significant change in delay or accuracy over time or over non-uniform load.

5.7.6 Synchronization Costs

In our evaluation we have measured SPEX’s latency under a workload where the actors produce point publications and have area subscriptions. We did not yet investigate area publications. This is because defining area publications, how they relate to the VE and the effects they might have on other actors is extremely application specific and cannot be easily mimicked. An area publication, depending on its size, may result in a transaction that spans multiple partitions. Since the partitions could reside on different hosts, the transaction could touch several hosts. Hence, in this section we quantify the effects of using one-round transactions for synchronizing generic events that span multiple servers. We wish to measure the basic overhead of using the transaction’s commit protocol to ensure global serialization. Comparison is done to the alternative of simply sending messages to servers in the form of one-to-many communications without any predictable ordering. We leave evaluating synchronization overheads with meaningful area publications as future work.

In the testbed setting, 11 machines act as servers while 5 client machines issue an aggregate of 1000 transactions per second. We vary the number of servers each transaction contacts from 1 (a normal message) to 11 (an event that effects partitions on all servers). Note that 11 represents extreme all-to-all communications. Each transaction delivers a simple 8-byte timestamp generated at commit time to its target hosts. It is designed to be small to not IO saturate the machines. A host
Figure 5.9: SPEX performance over period of 1 hour of run time with random waypoints.
Figure 5.10: SPEX performance over 1 hour of run time with VE size of 256x256 and different numbers of hotspots.
Figure 5.11: Cost of using one-round transactions for global synchronization instead of simple message passing.

computes the one-way delay it took for each timestamp it receives to get from the client to it, be correctly serialized according to the commit protocol, and processed by the server. We compare this delay to when the client directly sends the message to all servers. Isolation mode was set to clock mode which performs well under heavy contention. Results are an average of 3 runs each.

The results in Figure 5.11 show with 1 server, i.e., point publications or area publications that fit in a single partition, there is no distinguishable difference between using transactions or direct message passing. The commit protocol is optimized to only take a single RTT to complete for single server transactions. The number of messages that have to be delivered increases linearly with the number of servers. As a result so does latency. For higher number of contacted servers the commit protocol uses 2 RTTs. The gap between message passing and transactions increases with the number of servers but never exceeds a factor of 2x. Hence, the lower the RTT between the hosts, the lower SPEX processing delay will be. We speculate SPEX to have a higher processing delay in the presence of area publications than what we measured in Section 5.7.1. But as we observed in this section, the difference will be in the order of a few extra milliseconds. SPEX will live up to its promise of unrestricted area publications.
5.8 Related Work

5.8.1 Commercial Deployments

Successful commercial systems currently exist that support large-scale VEs. Second Life [33] uses partitioning to assign each square region of size $256 \times 256$ to a specific core [34]. Each region can handle at most 100 actors and actors can only influence the current region they reside in. World of Warcraft [46] uses sharding to separately run multiple parallel instances of the VE [99]. Users are assigned to a specific shard and cannot migrate across them, nor can they interact with other shards. EveOnline [17] supports numerous players in a single VE. It uses a solar system architecture [1], but at its heart is a single centralized SQL database. Pikkotekk [27] has the record of supporting 999 players in single FPS game. A group of cell servers handle different cells (i.e., partitions) of the VE. Message passing is used to communicate between the cells. However, all users connect to a central Pikko Server which act as a mediator between users and cell servers. While it supports area effects, distributed consistency is completely ignored. SPEX uses a fully distributed architecture. Users do not have interaction limitations, nor is there any central component. SPEX supports area publications and explicitly deals with consistency.

5.8.2 Query/Broadcast-based

Our work focuses on the support for low-latency interactions such as FPS games. Colyseus [56] and Donnybrook [55] also aim to support scalable FPS. Their approaches, as well as the later cloud-based architecture by Tayarani et. al. [104], use two classes of update rates for interest management, one low frequency position update (or spatial query for Colyseus) is used for neighbour discovery, and a high frequency update via direct connections is used for actual interactions. Donnybrook has demonstrated up to 900 players via simulations, while Tayarani et al. have shown up to 320 players interacting in a cloud environment. However, the low frequency update is broadcast globally, which imposes an inherent upper bound on scalability. The query time for neighbour discovery in Colyseus increases logarithmically with the number of actors, so the system will cease to satisfy real-time
requirements beyond a certain scale. SPEX uses progressive partitioning, thus a host only needs to know about a bounded number of partitions, regardless of the total number of actors. SPEX is also practically deployed.

5.8.3 Partitioning
Spatial partitioning based systems such as N-Tree [83] and OPeN [77] both utilize quad-tree to dynamically partition the VE. N-Tree targets to support spatial multicast (i.e., publication), and OPeN focuses on spatial query (i.e., subscription). While both also provide dynamic partitioning, they do not support the combination of both publication and subscription as in SPS, and both have been mainly evaluated by simulations or analysis [83]. SPEX supports both publication and subscription, with the more general primitive of SPS, and its feasibility is demonstrated using actual experiments. Matrix [93] uses dynamic partitioning to distribute a VE between multiple hosts. However, it uses a centralized server to hold the partition mapping which is updated upon each modification, signalling a bottleneck and single point of failure. In contrast, SPEX uses a scalable and fully distributed partition hierarchy.

Previous attempts have been made to use spatial partitioning in conjunction with scalable pub/sub, e.g., SimMud [91] assigns an application-layer multicast channel (based on Scribe) to each rectangular region. However, SimMud uses full static partitioning and has no load balancing. SPEX uses progressive partitioning while supporting adaptive load balancing.

5.8.4 SPS
S-VON [87] and VSO [86] are two recent proposals supporting SPS based on Voronoi partitioning. S-VON enhances a Voronoi-based Overlay Network (VON), which provides spatial subscription, with the additional function of spatial publication. However, it has no built-in mechanism for dynamic load balancing. VSO provides SPS based on a super-peer design, while dynamically adjusting the Voronoi regions to balance load. SPEX differs from VSO mainly in two aspects: 1) the partitioning is based on a hierarchy as opposed to flat Voronoi; 2) a distributed transaction provider is used to maintain consistency and reliability guarantees. While
VSO has demonstrated up to 1000 entities under balanced loads, both S-VON and VSO have only been verified by simulations.

The Data Distribution Management (DDM) [102] mechanism from the HLA standard is an earlier form of SPS. Initial works use only a server to perform the task of interest management, while subsequent works use grid-based partitioning and dynamic multicast address assignment to scale up the DDM operations [59]. DDM for discrete-time simulations (i.e., non real-time simulations where logical clocks of entities may advance at different time-scales) has also been introduced [115]. However, having a limited number of IP-based multicast addresses, as well as the fixed-size nature of grid-based partitioning, are classical bottlenecks to scalability. SPEX can be seen as providing both dynamic partitioning where the number of partitions is proportional to the user size, while the multicast channels are much more numerous, as they are logical and provided at the application-layer.

5.9 Conclusions and Future Work

We present SPEX, a system capable of supporting a large number of concurrent users in a VE, without common limitations, e.g., density cap or limited interactions. The system can balance scalability and operation costs by dynamically adjusting the pool of SPEX hosts. SPEX decouples publications from subscriptions, allowing users to publish to locations outside their current AoI, i.e., long-range interactions. It uses a fully distributed architecture with no components required to maintain global state, but allowing information to efficiently be located using a distributed hierarchy. Our contributions include: 1) enable long-range interactions with spatial pub/sub (SPS) based on a distributed transaction provider; 2) adaptive load balancing using a combination of progressive partitioning and scalable pub/sub to address the user density issue; and 3) evaluating the practicality of SPEX at large-scale both in a testbed and over the Internet with a real system. Our evaluation shows that SPEX scales to 750 actors and provides low end-to-end latencies and high accuracy. SPEX thus is an architecture suitable for hosting the next generation of on-line games: epic-scale FPS games.

SPEX focuses on VE distribution, interest management and message dissemination. One-round transactions provide the means to correctly synchronize events
spanning multiple servers, but we did not extensively deal with conflicts occurring between users, which is important to support more sophisticated game states and logic. Our future work is thus to create a complimentary component to simplify the dealing of user conflicts. SPEX takes user events and translates them to SPS functions. It creates per-user interest sets and AoEs. A future system could use this information and distributed transactions to deal with user interactions and conflicts. By using multiple transactions per user to over estimate users’ range of effects, we can then deterministically resolve conflicts in a fair manner.
Chapter 6

iEngine

6.1 Introduction

Relational Database Management Systems (RDBMSs) have for decades served as an integral part of computing environments and are the fundamental component for data storage in almost all industrial and commercial environments. Enterprises rely on RDBMSs for data storage, management and analysis due to certain key features. An RDBMS provides a high degree of expressiveness, sufficiently simple data model, sound mathematical foundation, and widely accepted standards.

Data is made available through the Structured Query Language (SQL) as the universal language. SQL is simple, yet general, and simplifies interactions with the data as it allows a very broad set of possible questions to be asked [45]. SQL is well understood and has enabled an ecosystem of management and operator tools to help design, monitor, inspect, explore, and build applications. Additionally, SQL is system agnostic. The application doesn’t need to know the details of how the RDBMS is implemented. A SQL programmer can reuse their API and UI knowledge across multiple back-end systems. This reduces application development time with increased code quality. It appears RDBMSs are more likely to play their current role for the foreseeable future [45].

Traditional relational databases, including those from Oracle and IBM, are built with a share-everything centralized topology. They scale up their throughput by adding more expensive hardware with immense licensing costs. With the emer-
gence of cloud computing and hosted services, Database-as-a-Service (DBaaS) is sought as an attractive new model of systems deployment, and an alternative to expensive centralized solutions.

DBaaS provides the luxury of a DBMS in the form of a service hosted in a cloud environment \[4,19\]. Service administration is mostly taken care of by the cloud provider, which has to deal with configurations, failures and outages. Resources are dynamically provisioned based on demand, inherently providing scale-in and scale-out as required. Most importantly, the application is exposed to a very appealing pay-per-use cost model. It only pays for what it uses without being forced to predict (and pay for) what it might use.

The majority of DBaaS offerings have not yet been able to provide one of the most crucial features of hosted services: scaling beyond one computing node. Amazon RDS \[4\] provides the feature of scaling out into a larger machine when demand increases. However, it can only go as far as the most powerful computing node Amazon has to offer, and not beyond \[120\].

Over the past years, large-scale applications have faced a steep growth in user demand for applications across the spectrum. In response, the scalability of RDBMSs has become an active area of research. The objective is to provide the ability to scale both in terms of storage capacity and query processing power.

Partitioning \[29,90\] is a widely employed technique used to tackle scalability, where the DBMS splits the data into smaller portions and indexing and isolation are handled independently and locally to each partition. Work in this area still continue to find ways to efficiently partition data to reduce costly distributed transactions \[73,74,100\], i.e., operations that cross partition boundaries. This requires a-priori knowledge of all queries that may ever be executed in the system. With current complex data querying requirements of applications, it is often hard to produce a good partitioning scheme that both helps scalability and reduces the number of distributed transactions.

On the other hand, scalability has been achieved by shifting towards more relaxed data consistency models than strict ACID compliance, and through data access primitives far less expressible than SQL \[65,76\]. The new abstraction has been key to the success of some systems, but the weak consistency data model gives the developers a hard time trying to develop stable, error-free systems \[51\]. Most of the
heavy lifting done in the DBMS is pushed up into the application adding unwanted complexity. Also, the narrow data access primitives suffice to only serve specific types of applications and come nowhere close to the generality of SQL. This has resulted in a general lack of interest from enterprises in the NoSQL paradigm [113].

While our ultimate goal is to implement a highly scalable truly elastic RDBMS, the most important question we seek to answer is whether we can build a fully SQL-compliant RDBMS without having to make compromises? How can an RDBMS simultaneously provide scalability, consistency and generality?

In current RDBMS architectures, from the query planning module to the storage module, all the components are tightly coupled with each other in a single node setup. Commercial RDBMS engines are highly sophisticated and optimized with a solid mathematical foundation. Re-designing database concepts in a distributed fashion is not a trivial task by any means. Instead, we take a different approach and consider how multiple RDBMS engines can operate on the same data set simultaneously, without having to know about each other.

We present iEngine, a memory-resident distributed database storage engine. Data is stored in memory for fast access and logged to disk for durability. iEngine is aimed to scale in both dimensions of storage throughput and query processing power without having to sacrifice any parts of SQL. An unmodified DBMS engine is built over a unified distributed storage layer which is capable of storing and indexing data and isolating concurrent operations from each other. The distributed storage can scale its throughput and storage capacity by adding more nodes to each of the components used for storage, indexing or isolation. Query processing power scales by adding more DBMS engines to the system.

The storage layer uses partitioning which is done internally and automatically. DBMS transactions are then implemented using multiple one-round transactions, i.e., high-performance short-lived transactions which use two-phase commit to execute. They provide the means necessary for a strong consistency data model. Concurrent transactions are isolated from each other using a distributed isolation component capable of locking arbitrary ranges of the data.

Our evaluation of iEngine show it to be a viable solution for DBaaS. It scales both for simple web based operations (lookups and inserts) and in the presence of complex range operations, and outperforms a single-node DBMS engine (InnoDB).
iEngine is a general purpose storage engine compatible with modern DBMS engines. It architecturally supports being simultaneously used by different engines. This provides the application the freedom to choose the most suitable engine for each task at hand.

### 6.2 Current RDBMS Architectures

In RDBMSs, data is stored in different tables. Each row of the table is a separate data item and can have multiple different attributes, each stored as a separate column in its associated row. Tables are created, destroyed, and populated with items all through SQL queries specified by the user. Transactions can be used to aggregate different queries into an atomic set.

Figure 6.1 shows the common components in a conventional DBMS architecture. The components can broadly be categorized into ones used for SQL processing and those used for storage.
6.2.1 Query Processing

When a query arrives, it is first parsed and checked for correct SQL semantics. Then an optimum execution strategy is devised by the SQL processing layer. This is the layer that provides the rich querying features in the DBMS.

The optimizer uses meta values obtained from the underlying storage as measurements. Examples of meta values are size of the index, average scan time, index lookup time and delete times for a given table. It then uses these measurements to come up with an efficient plan in order to return the data as quickly as possible to the user. Numerous factors contribute on how the execution is planned. Factors such as the number of rows that have to be fetched from storage, whether to use the index or to resort to a table scan, and which key would require fetching a fewer number of rows are taken into consideration.

6.2.2 Storing & Indexing

Once an execution plan is decided, data is fetched from the storage layer. One way to obtain data from a table is to check each row of the table against the query to determine if it should be part of the result. A task known as a table scan. Table scans are expensive and don’t scale as the data set grows. An alternate way is to use the index.

An index is a data structure that improves the speed of data retrieval operations on a database table at the cost of additional writes. Hash tables and B-trees are common structures used for indexing. The index uses more storage space to maintain an extra copy of data as items are added to, modified, and deleted from the data set. Indexes can be created using one or more columns of a database table, providing the basis for both rapid random lookups and efficient access of ordered records [14].

The query specifies how the data should be retrieved. The information in the query is presented in the form of constraints, such as items with specific values, items with values within a specific range, or items with any values. These will result in an index lookup, index range query, and table scan respectively. Once the data has been located (either via the index or from the table directly), it is read from data storage and the results are passed back to the query processing layer.
6.2.3 Isolation

A DBMS allows multiple transactions to execute concurrently for increased throughput. Consequently, different transactions should be isolated from each other when required. Locking is the primary technique used for isolation in RDBMSs. Locking is used for hiding the transient state of one transaction from others. As a transaction executes, the rows of the tables it touches are locked to ensure no other transaction can modify them until the transaction finishes. Based on the details of what to lock and when to release them, four different isolation modes are possible:

- **Read Uncommitted** There is no isolation among transactions. Any transaction can see intermediate results of other transactions. In terms of locking, read/write locks on items and ranges are only held during the execution of the current statement and not the entire course of the transaction.

- **Read Committed** Transactions are only allowed to see values once they are committed by others. A transaction can see values committed after it started executing. Write locks on items are held for the duration of the transaction while read locks and range locks are held for the current executing statement.

- **Repeatable Reads** Transactions can see only values committed before they start. Read/write locks on items are held for the duration of the transaction and range locks are held for the current executing statement.

- **Serializable** As if all transaction were executed in a global serial order. All read/write locks and range locks are held for the duration of the transaction.

Serializable isolation provides the strongest form of consistency, and is the hardest to provide. It is the only form of isolation which requires complex range locks. A range lock not only locks items that currently exist in a database falling within the given range, but also should prevent future items from being inserted into that range until the range lock is released. Hence, simple item locks will not suffice. This is to ensure that a phenomenon known as *Phantom reads* will never occur.
6.2.4 Distribution Challenges

Non-distributed RDBMSs follow a monolithic and shared-everything architecture with a tight coupling between the components. The query processing components are highly sophisticated and based on a solid mathematical foundation. Finding distributed alternatives to database concepts is by far a non-trivial task.

In the storage layer, indexing and locking are two important components. The indexing system is expected to be able to answer complex queries over the items it stores, such as finding items with specific values on selected attributes. Since almost every operation needs to either read from or write to the index, the index is an important potential point of contention.

The locking mechanism is required to lock both individual items or all ranges of items that have attribute values in specific ranges. The need for efficiently handling complex range locking renders most available scalable locking systems inefficient for direct use in a distributed RDBMS (see Section 6.8.3).

Finally, the RDBMS heavily relies on the strong consistency of both of these components, which presents a major challenge for scaling the storage layer of a single-node RDBMS.

6.3 iEngine

Figure 6.2 presents the general architecture of iEngine. We take a commodity-off-the-shelf RDBMS engine and scale it out horizontally by substituting its single-node storage with a distributed storage layer, leaving the engine intact. Since the engines are unmodified, iEngine provides the same level of generality as a traditional RDBMS. Similar to a single-node setup, an engine is given the illusion that it’s the only one running in the system and granted access to all of the data. In reality, numerous engines are allowed to coexist simultaneously by sharing the same storage layer.

The interface between an RDBMS engine and the storage layer masks out the underlying distributed nature and correctly isolates the engines from each other. Data storage, indexing and locking are the core functionalities of the distributed storage layer, where each component can be scaled independently from the others. With such a loose coupling between query processing and storage, iEngine
can scale in two dimensions. In the first dimension, the storage can transparently scale to provide more storage capacity and operation throughput. In the second dimension, the number of RDBMS engines interacting with the storage may scale to accommodate more query processing power and handle higher transaction rates.

The distributed storage layer can be broken down into three separate components used for storing data items, indexing the data and isolating transactions through locking. Each component scales independently to provide the ability to accommodate different workloads. For example, a rarely modified data set only requires the storage component to scale, while a read-only query-heavy workload would require more indexing power, and a write-heavy workload with long-lived transactions would need scalable locking performance.

Figure 6.3 illustrates the different components of the distributed storage layer. Theoretically, the separate blocks can reside on different physical machines. The storage handler is the binding between the query processing unit of an RDBMS engine with distributed storage layer. It communicates with the front-end of the storage layer which is the transaction handler.

The transaction handler is the only component in the system that knows about
SQL transactions and queries, and translates them into the appropriate set of smaller operations. The smaller operations use a key/value style, where each operation is assigned a globally unique key. This ensures dependencies between transactions touching the same data are correctly detected and isolated. The transaction handler uses one-round transactions to convey the key/value operations to the back-end servers that store, index and lock the data.

The back-end servers serve requests following the one-round transaction commit protocol. They use disks to synchronously log the data for durability. Other components ensure enough data is stored inside the servers to provide crash recoverability from failures of any part of the system.

### 6.3.1 Query Processing

We use MySQL [28] as the query processing RDBMS for our prototype implementation of iEngine. MySQL has widespread usage across different domains, and is used as a commercially hosted database service [4,19]. As illustrated in Fig-

**Figure 6.3:** Distributed storage layer architecture.
Figure 6.4: MySQL architecture.

6.3.2 Storage Handler

The SQL operations understood by the storage handler are those dealing with table creation/deletion, ones that define transaction boundaries (Start, Commit, Rollback), and the standard CRUD operations (Create, Retrieve, Update, Delete). The storage handler is responsible for mapping SQL based queries to commands understood by the storage layer. Table manipulation and transaction definitions have a simple translation to storage commands. The CRUD operations need some pro-
cessing to compute their respective key/value based operations.

Each operation’s key is computed based on the table’s name and information provided by the engine. Using the schema of a table and a given query, the storage handler checks to see if exact values on all attributes have been provided. If so, the query operates on an individual item. If not, the query is operating on a range. The operation’s key, type and additional information are sent to the transaction handler.

For individual items, the key is computed as a concatenation of all the attributes in the table’s namespace. For ranges, based on the data types specified in the schema, the item’s key is computed as a range key. A range key consists of two different keys, one specifying the minimum possible attribute values and one specifying the maximum.¹

Storage handlers don’t have a strict affinity to a specific transaction handler and are assigned to them in a round-robin fashion. For performance reasons, we try to run a transaction handler on the local machine to each MySQL engine to take advantage of faster communications. Each storage handler instance makes a separate connection to the transaction handler which uses blocking IO to prevent the engine from issuing requests while the transaction handler is trying to keep up.

To reduce extra communications with storage, the storage handler caches results returned from the previous requests to be used by subsequent requests on the exact same data. Caching only happens for the duration of a transaction and when the isolation level can ensure cached data may not have been modified by other transactions. Cache items are modified according to operations performed by a transaction to ensure they reflect the latest results on subsequent requests.

The storage handler interface is generic enough to support other RDBMS engines. This will allow a heterogeneous mixture of engines to operate on the same data, each specialized in a different specific task.

### 6.3.3 Transaction Handler

The transaction handler is the front-end binding point to the distributed storage layer. The main duties of the transaction handler are to store and index data, while also acquiring the required ranges or item locks. It is the only part of the storage

¹ A key can be thought of as a range key where the minimum and maximum are the same.
layer that knows about transaction boundaries. All lower layers beyond this point only understand individual key/value operations.

Commands arriving from the storage handlers are transformed into one or more separate operations. Some are executed as they arrive, e.g., searching for specific data items, while others are postponed until the transaction is committed, e.g., modifying stored data.

Data is stored and indexed inside Innesto, a distributed strongly consistent key/value datastore. Innesto uses spatial partitioning to support search on multiple different attributes and provides consistency by using one-round transactions. A modified version of Innesto is also used for range locking.

When a table is created, two separate instances of Innesto are created. One is used for indexing and storing data items, and one is used for logical locking of items and ranges of the table (lock manager). In contrast to mandatory locking which is usually hardware enforced, logical locking is a higher level mechanism where the existence of a data item within a data structure is interpreted as the given item being locked.

A table could have any number of rows, where only a subset of them are indexed. For example, a table may have ten different columns but only decide to index three of them. This means queries that provide constraints on values of the three indexed columns will complete by using the index, while ones that use the non-indexed columns have to perform a table scan. As a result, in this example two three-dimensional Innesto instances are created.

Data items are stored by inserting them inside the index. To lock an item, it’s key is inserted inside the lock manager, and removed for unlocking. Ranges are locked in a similar fashion, where the range key is inserted and removed to lock and unlock the desired range.

6.3.4 Correctness

To verify correctness of the overall system, we classify correctness into two categories: data integrity and data correctness. High level RDBMS logic, such as executing transactions, is conducted by commercial RDBMS engines on top of the distributed storage. Since we are not modifying the engine nor changing any
database fundamentals, data integrity is a given to the architecture.

Data correctness implies that all operations should comply to semantics required by an RDBMS. For example, each SQL transaction should be provided with the isolation level it requested, and rolling back a transaction should not create a transient inconsistent state, no matter how brief. Data correctness is affected by the distributed nature of the system. Since multiple operations may be simultaneously operating on the shared data, it should not be corrupted due to race conditions between (possibly) conflicting operations. iEngine uses one-round transactions to provide data correctness and Innesto for range locking and isolation. The interface to the storage layer complies to what the RDBMS requires and ensures data correctness is never violated.

6.4 Range Locking

Innesto provides a data partition hierarchy abstraction. Parent partitions cover their children. Each Innesto operation is tested against the hierarchy to find target partitions during execution. Single item operations usually touch a single partition, where the search operation is an exception that touches multiple. We extend this idea to provide the ability to perform range locking.

Two operations are supported for range locking which operate on range keys. Inserting a range key locks the range specified by it, and removing the key unlocks the range. A single item can be locked by inserting a range key which has its min and max set to the items value.

Figure 6.5 illustrates an example of acquiring a range lock. While a single dimension is shown for simplicity, the same logic applies to higher dimension. Executing a lock operation starts from the root partition to find all target child partitions that intersect the range key. For each target, the range key is fragmented down into a key confined by the bounds of the partition and is checked against the set existing lock fragments within in. A lock fragment has a reference count of how many owners currently hold it.

As with conventional locking, read and write locks are supported. When there is a conflict with a write lock fragment, or the new lock is a write lock with a conflict with a read lock fragment, locking fails. Conflicts between read lock fragments
Figure 6.5: Range lock example. The numbers represent the reference count of each fragment. Note that the partitions and locks could have any number of dimensions.

are handled by finding their intersection. If they align perfectly, the reference count of the existing lock is incremented. Else, three new lock fragments are created. The first one has a reference count of one representing parts of the new lock not in the existing range. The second is parts of the existing fragment not intersecting the new lock with its reference count unchanged. Third is the intersection fragment with a reference count of the existing fragment plus one.

6.4.1 Locks & Deadlocks

The commit protocol of one-round transactions used by Innesto uses short-lived locks. If locking fails by any Participant, all other Participants release their locks and the transaction is retried after waiting an exponentially growing random timeout. This is to prevent deadlocks. However, performance degrades drastically under contention.

In the face of a locking conflict, bandwidth is wasted by sending the transaction back and forth on each try. The transaction sits idle for a timeout hoping to make progress on the next attempt. Unnecessary contention is created by Participants which have to lock and release items when locking fails on another Participant.
In the case of executing RDBMS transactions, the transaction can’t make progress until its locks are acquired. Consequently, the system is brought to a complete halt beyond a certain point.

To avoid this, we changed the locking mechanism of one-round transactions to use blocking locks. As part of the commit protocol, if a transaction detects a locking conflict, it waits for them to be released. The Participant can still make progress on other transactions while one waits for the locks. Deadlocks are prevented by using a timer per transaction on each Participant. When the timer fires, it is a strong indication that there is a possibility of a deadlock. All locks are released upon a timeout as before with the addition that the SQL transaction is also rolled back.

6.4.2 Fairness

Starvation is a classical problem with locking systems. It is caused under contention where less restrictive locks constantly win over other stricter locks and starve them out. For example, read locks on a single item are less restrictive than write locks since they are compatible with each other and conflict with write locks, while write locks are always in conflict with both read and write locks. With the presence of enough read and write locks on the same item, if not dealt with correctly, read locks will constantly win over write locks indefinitely, preventing writes from making any progress.

With range locking, a new form of starvation occurs. As the range of a range lock broadens, it becomes more restrictive. Locks with smaller ranges (and single item locks) have the potential to starve broader range locks. We circumvent this by using an acyclic directed dependency graph for pending locks. The graph provides FIFO locking. Figure 6.6 depicts an example of range locking requests and the corresponding lock dependency graph.

Each node in the graph is a lock request pending on a conflict. A new lock request checks both the graph and existing locks for all possible conflicts and adds a dependency to them. A node with no dependencies is granted its lock. When a lock is released, the node is removed from the graph and locking resumes. A node keeps track of all possible conflicts since intermediate nodes could be removed at
any time due to timeouts or events occurring on other Participants.

FIFO locking add some extra waiting time to a range lock. In Figure 6.6 lock 5 has no conflict with an existing lock, but only on the pending lock 3. Without the graph, lock 5 would be granted its lock as soon as it arrived. However, this would starve lock 3 which now has to also wait for lock to finish.

### 6.4.3 Timeout Tuning

Tuning the timeout value of deadlock detection is an important task. A value too short would result in a premature abort for most transactions. This wastes the time they spent inside the dependency graph, and reduces the overall throughput of the system. Setting the timeout to a larger value would cause deadlocks to be detected too late. Blocked transactions would spend a long time before finally being aborted, keeping others from making any progress.

The workload of different partitions is different and varies over time. We compute the timeout value per Innesto partition by profiling each one’s workload. A histogram of the time it takes to acquire the locks on a partition are computed during the profiling period. The timeout is set at a configurable specific percentile of
the histogram (e.g., 95%).

Profiling data is migrated with partitions as they are split and migrated in Innesto. Lower percentiles will result in higher throughput in the presence of low contention. As contention increases, higher wait times become normal and most of the slow transaction will start aborting. Thus we allow the option to periodically profile the workload to decide new timeout values.

6.5 Anatomy of a Transaction

A transaction has three significant boundaries: starts, commit/rollback, and completion. Any CRUD operations performed between start to commit are performed atomically and isolated from other transactions. In the case of rollback, all effects of the transaction are erased. The time between commit/abort and completion is when the storage layer is busy applying or removing the effects of the transaction.

SQL CRUD commands are sent by the storage handlers to their respective transaction handler. The transaction handler transforms the CRUD requests on a table to individual requests on the index and locking Innesto instances associated with the table. Each operation’s key clearly defines parts of the data it would be touching so that the locking mechanism would work correctly.

Regardless of the type of operation, locks are acquired immediately based on the operation’s key. According to the modified locking protocol, locking most likely fails due to deadlocks, which will cause the transaction to rollback. Read operations (Lookup) are executed once the locks have been acquired. Executing write operations (Create, Modify, and Delete) is postponed until commit time. This will reduce the cost of rolling-back transaction halfway as only locks will have been acquired by the transaction. Once commit is received for the transaction, write operations are executed, after which locks are released.

There are a few exceptions to the mechanism described above based on what SQL expects when executing a transaction. For example, Create expects the item to not exist. Once the lock on the item is acquired the transaction can be sure no other transaction could create the item until it is done. However, if the item already exists, it will be too late to discover its existence at commit time. For this scenario, Create also performs a Lookup immediately after acquiring locks to ensure it is a
valid command.

### 6.6 Fault Tolerance

Failures are inevitable in distributed systems, in the form of hardware failures, network partitions or full scale datacenter outages. A failure could leave data in an inconsistent state due to partial execution of transactions. Thus we consider different failure scenarios to ensure no kind of failures would allow inconsistencies.

#### 6.6.1 Server Failure

Server failures are detected by other servers and machines trying to commit one-round transactions. One-round transactions use logging to protect the data from failures. All write items are logged synchronously as part of the commit protocol. Upon the failure of a server or group of servers, their log is used to reconstruct their data.

Innesto uses one-round transactions. As a result, any Innesto operation will execute to completion, even in the presence of failures. Only execution time of operations in flight during the failure will be affected, as some will have to conclude after the failed servers have been restored.

#### 6.6.2 Engine Failure

Engine failures are detected by their associated transaction handler instance. When an engine fails, transactions that may be committed are continued to completion. These will be transactions that the storage handler told its transaction handler to commit and was waiting for a completion notification. All other uncompleted transactions where commit hasn’t been received are rolled back by the transaction handler.

#### 6.6.3 Transaction Handler Failure

Transaction handlers are the only component in the storage layer that know about SQL transaction boundaries. While it is possible for storage handlers to cleanup partially executed transactions if they detect their transaction handler has failed,
such an approach will not withstand simultaneous failures of storage handlers and
transaction handlers.

There are two types of uncommitted transactions. Ones that have already
started but commit hasn’t been issued yet. Such transactions have acquired locks
and buffered their write operations. Another type of uncompleted transactions are
those that commit has been issued by the storage handler but the transaction han-
dler failed whilst commit was in progress. Such transactions may have already
applied some write operations but not all, and locks may not have cleanly been
released. To address recovering each type of transaction upon failures we use two
separate logs: undo log and redo log. Both logs are stored inside servers using
one-round transactions, hence, protected from failures.

An undo log includes a list of locks acquired by each transaction. Each lock
operation is in essence an Innesto insert, and each unlock a remove. Innesto allows
combining multiple operations using the same one-round transaction. We take
advantage of this to write to the undo log each time a lock is acquired and released.
The causes the undo log to be in perfect sync with the locking system even in the
face of failures. To recover from a failure, the undo log has to be traversed to find
lingering locks and remove them.

When the transaction handler receives a commit requests it has to apply the
write operations and release the locks. But before doing so, it makes a full copy of
the write operations and writes them to the redo log using a one-round transaction.
If a failure happens before the logging to the redo log completes, we treat this
transaction as if commit hasn’t been received. Otherwise, since the log will contain
a full copy of what needs to be done, the transaction can be completed by another
transaction handler using the redo log before locks are released.

6.7 Evaluation

We evaluate our prototype iEngine using a cluster of commodity machines. Each
machine has two quad-core Xeon E5506 processors and 32GB of memory. Eval-
uation is done using the TPC-C benchmark. TPC-C is a popular benchmark that
extensively simulates a real world transactionally consistent workload. Designed
by the vendors that build share-everything architectures, TPC-C represents an ex-
tremely challenging workload on a distributed RDBMS.

We compare the performance of iEngine against InnoDB, the default engine in MySQL. InnoDB is the only storage engine that ships with MySQL and supports transactions with serializable isolation. InnoDB is widely used, even in cloud hosted database offerings [4, 19]. All experiments use MySQL 5.5.8 source distribution and InnoDB 1.1.4. For a full list of InnoDB configuration parameters see [125]. In summary, the buffer pool was configured to use 28G, log only flushed once a second using fdatasync, double-write was disabled, and the log file size was set to 1900MB. When using iEngine, MySQL internal caches were disabled to avoid using stale data. TPC-C’s consistency checks were used to verify the results of TPC-C and the functionality of iEngine.

To emulate a practical workload on the system we used a benchmark client [40] which generates workloads based on the TPC-C [42] specification. We first test iEngine’s scalability in Section 6.7.2, then compare how it’s isolation component performs compared to InnoDB in Section 6.7.3. A modified workload is used in Section 6.7.4 to test how iEngine performs under web compatible workloads. In all our experiments in the first three sections crash consistency was disabled. Thus we evaluate the performance costs of crash consistency in Section 6.7.5.

6.7.1 TPC-C

TPC-C simulates a complete environment where a population of terminal operators execute transactions against a database. The benchmark is centered around the principal activities (transactions) of an order entry environment. These transactions include entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses [42]. Data is stored in numerous tables, the largest of which is the Stock table which contains 100,000 items per warehouse. For example, a dataset with 140 warehouses has 14 million items stored in the Stock table.

Each user requests a chain of transactions from the RDBMS. A total of five types of transactions are used in TPC-C, with a mixture of read-only transactions and write heavy ones. The most frequent transaction consists of entering a new order (i.e., the New Order transaction) which on average is comprised of ten different
Figure 6.7: Strong scaling of iEngine.

items. Close to 10% of all orders must be supplied by another warehouse. This is to enforce distributed transactions when data is partitioned based on warehouses. TPC-C requires a certain mixture of its different transactions to complete within specified time bounds for the benchmark to succeed. A run violating the expectations fails. The performance metric reported measures the number of New Orders that can be fully processed per minute, i.e., tpm-C.

The main objective of our evaluation is to throttle the system as much as possible and to see how it scales. All transactions were executed in serializable isolation level except the stock-level transaction which ran with read-committed isolation (and permitted by TPC-C specification). User wait times were set to zero to avoid artificial delays. Before each experiment, the database is loaded with data based on the TPC-C specification.

6.7.2 Strong Scaling

In a typical single-node RDBMS, the throughput of the system increases as the number of users making requests scales until the saturation point. Beyond the saturation point, throughput cannot increase as the system has reached its maximum
capacity. iEngine is designed to provide the ability to shift its saturation point to more than what a single machine can handle.

Strong scaling is defined as how throughput of a system varies with a fixed problem size. We evaluate iEngine’s strong scaling using a dataset with 90 warehouses using two sets of machines. One set, varying from one to three machines, only run back-end servers. The other set use the same number of machines as the back-end servers to run the MySQL engines, transaction handlers and the client programs emulating TPC-C user requests [40]. Experiments are run until TPC-C starts failing.

Figure 6.7 shows as the number of users increases throughput rises until it saturates. Beyond a certain threshold, TPC-C starts failing due to long wait times for transactions. By increasing the system size, the saturation threshold is moved from around 3K tpm-C supporting only 750 users to around 7.5K tpm-C with 2100 users. This is evident that iEngine can scale beyond what a single machine can handle with the presence of more processing power.

6.7.3 Contention

InnoDB represents state-of-the-art RDBMS storage engine. It uses Multi-Version Concurrency Control (MVCC) to isolate transactions in which data items aren’t locked, but separate versions of the data are maintained for different transactions. iEngine uses logical distributed locking provided by Innesto. We wish to evaluate how iEngine’s isolation performs compared to the highly optimized and centralized isolation component of InnoDB. We do this by increasing user contention.

User contention is created as the number of users operating on a fixed-size dataset increases. This causes transaction conflicts and rollbacks. This is quite common in retailer store websites where customer interest in special offers increases. Two different dataset sizes are used, one with 144 warehouses and one with 240. For each dataset and system configuration, we increase the number of users until TPC-C starts failing. For InnoDB we use a single machine setup which runs MySQL server and the TPC-C client. For iEngine, we use a six machine setup, where similar to the InnoDB setting, each machine runs all required components.

Figure 6.8 shows that with a small number of users, InnoDB outperforms
Figure 6.8: User contention.
iEngine. As contention increases, InnoDB’s performance drops drastically and TPC-C starts failing close to 850 users for 144 warehouses. Increasing the dataset size only shifts the breaking point as it reduces contention. iEngine has a consistent performance and can support over 2.5x users compared to InnoDB.

TPC-C extensively stresses out different components of the system. We found locking to be the bottleneck as most transactions need to operate on the same table. iEngine’s performance depends on Innesto which depends on how fast one-round transactions can complete. Using commodity hardware, network RTTs are in the order of a millisecond. Decreasing this down a few orders of magnitude could help improve performance.

### 6.7.4 Web Compatible Workload

TPC-C extensively uses range queries as part of the benchmark. Web workloads usually consist of operations on individual items and not ranges. In order to both emulate a web workload and to see how iEngine behaves in a write intensive workload, we run TPC-C but only execute the New Order transaction which it write dominated. We use the same experimental setup as before where six machines are
used for iEngine and one for InnoDB. The dataset size is 240 warehouses.

As shown in Figure 6.9, as before InnoDB starts off well with low user contention but throughput falls steeply afterwards. On the other hand iEngine maintains its performance level with contention and throughput gradually drops as the number of users increases. This pattern could be equally attributed to the concurrency control mechanism and the B-tree based indexing in InnoDB. This is an indication that indexing based on Innesto is a suitable alternative to conventional B-tree based indexing.

### 6.7.5 Crash Consistency Cost

Crash consistency is provided to ensure data is recoverable in a consistent manner in the event of failures. To measure the cost of crash consistency, using a dataset size of 60 warehouses, we use six machines to run iEngine. Two act as one-round transaction servers and four run the rest of the components. Logging to disk was chosen as the failure protection mechanism.

Figure 6.10 shows adding crash consistency to iEngine incurs an overhead between 40%-50%. There are two performance costs associated with adding crash
consistency to iEngine. First, each write item of a one-round transaction has to synchronously be logged. Second, in a higher level, each RDBMS transaction in iEngine has to use the redo and undo logs during execution and commit. These two factors contribute to an overall lower performance when crash consistency is enabled.

Logging to disk adds time to the completion time of one-round transactions. This causes locks to be held for longer periods of time, magnifying the effect of congestion. As a result, more deadlocks occur and TPC-C starts failing earlier. The performance of iEngine is sensitive to the RTT between the machines. With our commodity setup this was in the order of a millisecond. Decreasing this base RTT will significantly improve the performance of iEngine.

### 6.8 Related Work

#### 6.8.1 Data Partitioning

Partitioning has been the standard mechanism used in scaling out RDBMSs beyond single machines. Most scalable RDBMSs currently support partitioning one way or the other. Schism [73] uses workload-aware partitioning to minimize costly multi-partition distributed transactions. The workload of the system is periodically monitored to identify the set of tuples accessed together. Schism uses graph partitioning to find balanced partitions which will reduce the number of cut edges, i.e., distributed transactions. Relational Cloud [74] uses a similar partitioning scheme to decide on the correct data placement across autonomous RDBMS engines used as the back-end to DBaaS. The efficiency of data placement thoroughly depends on how easy it would be to partition the workload in order to reduce distributed transactions. The partitioning gives the notion of a shared-nothing architecture, giving each RDBMS engine full autonomy. H-Store [90] (commercially known as VoltDB [43]) is a memory-resident database that partitions data across many single-threaded engines. The unit of transaction in VoltDB is stored procedures, each executed to completion in a single thread.

While optimizing the partitioning scheme helps reduce distributed transactions, it cannot eliminate them for generic workloads. In the case of VoltDB, a-priori in-
formation about the workload is required which is not always available. Optimizing for minimizing distributed transactions has resulted in redesigning the RDBMS engine, a process in which full SQL support has been lost. iEngine uses Innesto which internally performs automatic partitioning. However, iEngine acts as a distributed storage layer and allows unmodified RDBMS engines to interact with the data keeping SQL features intact.

MySQL Cluster [29] performs automatic partitioning and scales horizontally. It allows multiple instances of MySQL demons to run and access the data via multiple different APIs. Data is replicated across data nodes to geo replication. MySQL Cluster’s storage engine, NDB, only support the read committed isolation level. iEngine on the other hand supports generic locking and additionally supports serializable isolation.

6.8.2 Distributed Indexing

In scaling out RDBMS architectures indexing plays a key role. Some options for indexing are hash based, B-tree based, and spatial based indexing. While hash based indexing provides the fastest mechanism to access the data it can not efficiently support range queries and is useful for only a few use cases.

B-tree based approaches are the most commonly used method and support both item and single dimension range queries. Most notably is the distributed B-tree [48] which uses one-round transactions provided by Sinfonia [49]. It scales well beyond a state-of-the-art single node B-tree (i.e., BerkeleyDB). However, it works well for read heavy workloads and suffers under write contention. Minuet [111] is an improved version of the B-tree which supports OLAP and OLTP based queries, but only supports simple transactions. iEngine uses Innesto for indexing which scales for both read and write heavy workloads while also supporting multi-attribute range queries.

Spatial based approaches natively support multi-attribute range queries, including Innesto. BATON [89] provides an overlay over a cloud storage. It supports efficient range queries [122] but has a scalability bottleneck since all transactions are serialized through a central component. Innesto is fully distributed and provides the right balance between scalability, strong consistency and multi-attribute range
queries. For more related work in this area see Chapter 4.

6.8.3 Distributed Locking

Specialized locking systems, such as Chubby [61] and ZooKeeper [88], provide scalable locking along with strong consistency guarantees. Due to the way locking is implemented in these systems, efficiently implementing complex multi-attribute range locking (which is a common requirement of an RDBMS) is fundamentally not feasible. Google's Percolator [107] also implements a scalable locking mechanism on top of BigTable, but only works on snapshot isolation and cannot support serialization isolation, as required by strict consistency. Thus these specialized systems lack the generality and expressibility needed by an RDBMS. iEngine's locking component uses generic distributed range locking on multiple attributes with serializable isolation. It supports atomically logging to the undo log for fault tolerance by using one-round transactions.

6.8.4 NoSQL

With ever increasing user demands and data sizes, service providers turned to NoSQL to provide desired scalability requirements through narrow APIs and weak consistency guarantees with low latency. NoSQL systems are designed in a way to scale horizontally without any bottlenecks compared to conventional RDBMSs. Systems such as Cassandra [94], BigTable [65], and Dynamo [76] usually use hash based data placement to load balance and spread the data across many nodes.

High scalability comes at the price of sacrificing consistency and decent query capabilities. There are different flavors in the supported query models of NoSQL systems, such as column-oriented, document based, key/value, graph based, etc. While some systems support a variant of range queries, they are far from what SQL provides. Additionally, isolating operations and grouping them into transactions is very hard to achieve. This has led to two different worlds of SQL and NoSQL systems, each with its own application domain in which it outperforms the other [81].
6.8.5 Scaling Storage

With scalability and performance of NoSQL on one end and feature rich SQL queries and transactions on the other, attempts to find middle grounds between the two have formed in the shape of building SQL over NoSQL. The RDBMS is re-architected by de-coupling the storage layer from the query processing engine.

Megastore [51] partitions data into entity groups where consistency is guaranteed within each partition but has relaxed consistency across them. ACID transactions are supported within an entity group and are heavily used in Google [20]. ElasTraS [75] uses a two level hierarchy of Transaction Managers to route transactions to the right partitions and Distributed Storage to execute them. ACID transactions are only provided within a single partition and only simplified mini-transactions are supported across them. Barntner et. al. [60] designed protocols required for building database applications over Amazon’s S3 [5], a scalable object datastore, but provide no equivalent ACID transactions. While iEngine has a similar architecture it does not limit consistency to a defined set of boundaries. ACID is provided irrespective of the data being touched in a transaction.

Microsoft SQL Azure [63] supports ACID transactions over multiple records but the database size is constrained to fit on a single node. For larger data sets, the application needs to partition the data among different database instances.

Dueteronomy [97,98] has an architecture similar to iEngine where data handling and transaction handling are separated from each other. The lock manager provides three different lock granularities: items, partitions and tables. Hence, range locks are not as efficient as range locking in iEngine.

Another dimension of scaling the storage layer has emerged in the form of using MapReduce to implement SQL. Tenzing [66] is used internally by Google to serve queries over a Peta-byte of data but implements most of SQL, not all of it. InfiniDB [23] uses a MapReduce-like approach to operate on data through a MySQL interface. Queries are parallelized to avoid thread-to-thread or node-to-node communications. Such systems work well to run data analytics on large data but suffer if the database is primarily used to operate on individual items.
6.8.6 NewSQL

NewSQL is a new class of RDBMSs that seek to bridge the gap between NoSQL systems and traditional databases. New engines such as Spanner \cite{71}, VoltDB \cite{43} and Clustrix \cite{13} are completely redesigned database platforms. They are designed to operate in a distributed cluster of shared-nothing nodes, in which each node owns a subset of the data \cite{30}. NewSQL mostly target application that have transactions that are short-lived with no user stalls, touch a small subset of data with no full table scans or large distributed joins, and are repetitive by executing the same queries with different inputs \cite{30}. They lack workload generality.

Other NewSQL systems aim to provide optimized storage engines for SQL. TokuDB \cite{41} replaces B-tree indexing with fractal tree indexing for MySQL. While fractal trees have asymptotically faster insertions and deletions compared to B-trees, TokuDB still focuses on single-server functionality. MonetDB \cite{26} also uses a MySQL interface and replaces row based data storage with column oriented storage and indexing. Isolation in MonetDB is done using optimistic concurrency control which is known to quickly start failing under contention.

6.8.7 Scalable Transactional Models

Our focus when building iEngine was to provide the foremost distributed equivalent to the main duties performed by the storage layer of a traditional RDBMS, i.e., indexing and locking, without having to modify the engine. In an orthogonal direction, an interesting body of work has recently been done on finding ways to implementing transactions in a more scalable manner. DORA \cite{106} explores how to implement a thread-to-data model rather than a thread-to-transaction model to reduce the contention. Consistency Rationing \cite{92} explores how to adaptively change the required consistency levels when it matters to get better performance and cost effectiveness in the cloud. Larson et. al. \cite{95} provide concurrency control mechanisms that allow transactions to never block during normal processing. They might have to wait before commit to ensure correct serialization.

The latter seems an interesting approach for iEngine. Isolation is currently being done using the locking component. It could be modified to implement different concurrency control mechanisms which would better suit the target workload. We
leave this avenue for future work.

6.9 Conclusion & Future Work

We present iEngine, a scalable RDBMS which supports full SQL. Instead of re-architecting the RDBMS to scale, iEngine allows numerous unmodified SQL engines to be plugged into a scalable distributed storage layer. iEngines scales in two independent dimensions of adding more query processing units for OLAP, and scaling the storage layer for higher throughput and capacity. SQL engines bind to the distributed storage layer through a standard storage interface. The front end of the storage layer provides a translation of SQL commands to storage functions and ensures conflicting transactions are globally isolated from each other.

Indexing and locking are done using Innesto which is built over one-round transactions. A modified Innesto is used for distributed range locking. One-round transactions provide consistency and fault-tolerance to protect iEngine from all possible forms of failures. Our evaluation of iEngine show it scales beyond the capabilities of InnoDB, one of the few full SQL compliant systems, to provide serializable isolation using the TPC-C benchmark.

Our goal in building iEngine was to provide a fully distributed and scalable RDBMS without having to reinvent the SQL engine and well understood query processing algorithms. Our main contributions were to find distributed equivalences for indexing and locking. Future work on iEngine would be to provide multiple different isolation mechanisms. In the general case where the workload is unknown pessimistic locking is a suitable approach. However, there is much to gain when optimism is an option. Providing the ability to select between different isolation methods at a database level is a task left for future work.
Chapter 7

Future Work: Argal

Using one-round transactions to build distributed systems simplifies reasoning about concurrency and correctness. The application developers only need to ensure any assumptions that they made when performing a computation are verified at commit time. In the case of concurrent modifications, all but one conflicting transactions are rejected. The rest have to recompute their operations with refreshed data.

While this provides strong consistency, as with any other system where a limited resource is shared with high demand, fairness may become an issue. In a system with multiple threads of execution, fairness requires all threads to make some progress over time. In a perfectly fair system, multiple identical threads continuously executing transactions should on average complete the same number of transactions in any given time period.

Fairness usually doesn’t play a significant role in most systems. Retrying transactions after aborts usually takes a very short time not observable in a macro scale. Even if the delays are visible, users are accustomed to restarting their current task (e.g., refreshing the webpage) if they feel their session is too slow. However, for our target application of fast paced FPS games, fairness will start affecting enjoyability of the game.

In Chapter 5 we presented an architecture used for epic-scale FPS games. Players connect to SPEX over the Internet and are provided with regular updates. When a player makes an action, SPEX internally uses one-round transactions to execute
them. Since players observe the VE over WAN latencies, what they observe (and act upon) is lagging behind the latest state. So even though a player acts based on what it sees, its actions may be rejected due to network latencies. What aggravates the situation is that if two players observe the same state and make the same action (e.g., shoot at the same target or try pick up the same item), the player with the lower latency to SPEX will constantly win.

We provide a solution for providing fairness using one-round transactions. Two key features of FPS games make our solution possible. First, although the exact actions of each player per frame are not predictable, their maximum range of effect can be estimated. For example, it’s unknown which other player a player with a specific weapon in some point of the VE will shoot at, but it can only be someone within its weapon’s range. With proper interest management, the effects of a player are limited to their interest set. Second, the state of events may be classified into two separate components. State that is used to decide what actions to perform, and state that will be affected by the execution of an action. In FPS games the position of other players is an example of state that directly influences decisions, and health is what is influenced by the execution of action like shooting.

These two features provide the basis for reservations used for fairness. At any given time, the actions to be taken by a player are unknown until the state reaches the player’s machine and the player’s respective action comes back at a later time. A reservation reserves a spot in the state of the game for all possible future modifications made by each player. As time progresses and the actual action arrives, the reservation is modified into a one-round transaction and applied. Figure 7.1 provides the general architecture of using reservations in Argal.

Parts of the architecture overlap with that of SPEX and may be reused. Exclusive state of a player consists of state that is only modified by that player. It represents the state that drives action decisions made by other players such as position and orientation. This information should be propagated as fast as possible. As soon as it arrives, the interest manager and distribution overlay disseminate it as soon as possible. These are the tasks of SPEX described earlier. Shared state is the state of the players that is affected as a consequence of executing actions. Health and score are examples of shared state which may be modified by multiple sources. Modifications to this state are propagated with delay as the reservations are trans-
formed into one-round transactions and disseminated with the same distribution overlay.

Reservations are implemented by using clock mode 2 (Figure 3.1c) of the commit protocol of one-round transaction where the final timestamp is decided by the Coordinator rather than by the Participants themselves. A reservation is made by only executing the virtual time voting phase with dummy items. Once the write items are known the dummies are replaced with actual items. The Participants know to postpone executing transactions until their items have been received.

Global ordering of events has been investigated before. Corfu [52] uses a central sequencer to issue tokens to concurrent threads. Data is appended to the end of a shared log in ascending order of tokens. Tokens are similar to reservations. A token can be considered as a reservation that reserves all items in the system. However, compared to reservations which use a distributed voting protocol, tokens are issued using a centralized component. A token based system suffers from false sharing between concurrent threads since potentially irrelevant operations have to
wait for prior tokens to seal before they can proceed. Slower threads will hinder all other threads in the system even in cases where they don’t share any state.

The granularity of reservations is individual items and only conflicting threads are synchronized with each other. The application using reservations takes two steps to complete each operation. In the first step the one-round transaction is populated with all items the transaction could possibly touch. At some point in the future where enough information is obtained (e.g. the user action has arrived or a timeout occurs), the actual items are filled with the required operation and the rest are set to no-ops.
Chapter 8

Conclusion

In this thesis we opted for consistency built over a distributed one-round transaction framework. One-round transactions provide competitive performance compared to normal message passing transports, while simplifying programming distributed applications and reasoning about concurrency. They also provide protection of data against many forms of failures, such as software crashes, hardware failures and full datacenter outages. This allows factoring out redundant fault-recovery mechanisms from upper layers in the application stack. All these contribute to simplifying the end application.

We use one-round transactions to tackle building consistent versions of some known difficult distributed applications. Innesto, our key/value datastore, provides multi-attribute range search with comparable performance to that of a state-of-the-art eventually consistent datastore. SPEX presents a new architecture for epic-scale FPS gaming which circumvents traditional limitations in large scale commercial deployments without sacrificing consistency. iEngine takes a direct approach to distributing an RDBMS by pushing isolation semantics down to a distributed storage layer obviating the need to modify the query processing engine. Thus it fully supports SQL while providing scalability.

One-round transactions can at any time be transformed into normal message passing. In contrast to the traditional approach of trying to provide consistency where absolutely required at some point in the software stack, our experience with one-round transactions has taught us that it’s much easier to start with consistency
and move towards relaxing it. The steps needed to port to this framework are to first ensure all shared data is explicitly identified and any computations affecting or affected by them convey this information to their one-round transactions. Additionally, an extra network handler has to be provided to the framework which ensures proper isolation.

Our applications themselves serve as building blocks for other distributed systems and services. iEngine is general purpose RDBMS. A significant portion of online services still use SQL databases for data storage and could benefit from a scalable backend. While we presented SPEX in an FPS gaming scenario, it could be used for any system with a large number of event sources and consumers.
Bibliography


[34] Second life server architecture. wiki.secondlife.com/wiki/Server_architecture. Last visited 2015-01-05. → pages [56, 72, 85]


[71] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD,


[84] GILBERT, S., AND LYNCH, N. Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News 33, 2 (June 2002), 51–59. → pages 5, 28


134


