Geographically Distributed Database Management at the Cloud's Edge

by

Cătălin-Alexandru Avram

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2017

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Patrick Martin Professor, School of Computing, Queen's University
Supervisor:	Ken Salem Professor, Cheriton School of Computer Science, University of Waterloo
Internal Members:	Bernard Wong Associate Professor, Cheriton School of Computer Science, University of Waterloo
	Tamer Özsu Professor, Cheriton School of Computer Science, University of Waterloo
Internal-External Member:	Wojciech Golab Assistant Professor, Electrical and Computer Engineering, University of Waterloo

Abstract

Request latency resulting from the geographic separation between clients and remote application servers is a challenge for cloud-hosted web and mobile applications. Numerous studies have shown the importance of low latency to the end user experience. Small response time increases on the order of a few hundred milliseconds directly translate to reduced user satisfaction and loss of revenue that persist even after a low latency environment is restored. One way to address this challenge in geo-distributed settings is to push all or part of the application, along with the data it requires, to the edge of the cloud - closer to application clients. This thesis explores the idea of taking advantage of clients' proximity to the edge of the network in order to reduce request latencies.

SpearDB is a prototype replicated distributed database system which operates in a star network topology, with a core site and a large number of edge sites that are close to clients. Clients access the nearest edge, which holds replicas of locally relevant portions of the database. SpearDB's edge sites coordinate through the core to provide a global transactional consistency guarantee (parallel snapshot isolation or PSI), while handling as much work locally as possible. SpearDB provides full general purpose transactional semantics with ACID guarantees. Experiments show that SpearDB is effective at reducing workload latencies for applications whose access patterns are geographically localizable. Many applications fit this criteria: bulletin boards (e.g., Craigslist, Kijiji), local commerce or services (e.g., Groupon, Uber), booking and ticketing (e.g., OpenTable, StubHub), location based services (mapping, directions, augmented reality), local news outlets and client-centric services (e-mail, rss feeds, gaming). SpearDB introduces protocols for executing application transactions in a geo-distributed setting under strong consistency guarantees. These protocols automatically hide the complexity as well as much of the latency introduced by geo-distribution from applications.

The effectiveness of SpearDB depends on the placement of primary and secondary replicas at core and edge sites. The secondary replica placement problem is shown to be NP-hard. Several algorithms for automatic data partitioning and replication are presented to provide approximate solutions. These algorithms work in a geo-distributed core-edge setting under partial replication. Their goal is to bring data closer to clients in order to lower request latencies. Experimental comparisons of the resulting placements' latency impact show good results. Surprisingly however, the placements produced by the simplest of the proposed algorithms are comparable in quality to those produced by more complex approaches.

Acknowledgments

I would like to express my gratitude towards my supervisor, Dr. Ken Salem who has been incredibly helpful throughout the long process of developing all the work that is described in this document and instrumental to its completion. I have learned a lot from him both personally and professionally.

A special thanks to Dr. Bernard Wong who also took part in the projects that have lead to this thesis and provided me with much needed guidance along the way.

I would like to thank the other members of my committee as well: Dr. Tamer Özsu, Dr. Patrick Martin and Dr. Wojciech Golab; for taking the time to read and review this thesis.

And finally I would like to thank my family for all the support and understanding they have provided throughout my studies.

Table of Contents

Preamble **i**

Examining Committee Membership iii

Author's Declaration v

Abstract vii

Acknowledgments ix

Table of Contents xi

List of Figures xv

List of Tables xix

1 Introduction **1**

1.1 Latency 1

1.2 Edge Applications 4

1.3 Thesis Organization and Research Contributions 4

2 Related Work **7**

2.1 Latency 7

2.2 Caching and Content Delivery Networks 8

2.2.1 Edge of the Cloud 9

2.2.2 Economics of the Cloud-Edge 10

2.3 Databases 10

2.3.1 General Purpose Distributed Databases 10
2.3.2 In-Memory Databases 12
2.3.3 Other Levels of Abstraction 12
2.4 Data Partitioning 13

3 SpearDB Overview 15
3.1 Characteristics of SpearDB 16
3.2 Database Interface 17
3.3 Consistency 18
3.4 Edge Localization 21
3.5 Site Databases 22

4 Transactions in SpearDB 25
4.1 Transaction Execution 27
4.2 Committing Transactions 27
4.3 Transaction Shipping 34
4.4 Discussion on the Effects of Failures 35
4.5 Data Movement 36
4.5.1 Data Movement Operations 37
4.5.2 Dynamic Data Movement Discussion 41

5 SpearDB Prototype Evaluation 43
5.1 Experimental Setup 43
5.2 Micro-Benchmark: Commit Strategies 44
5.3 Micro-Benchmark: Locality 46
5.4 Micro-Benchmark: Walter 49
5.5 Evaluation Using RUBiS 52
5.5.1 RUBiS 52
5.5.2 Experiment Design 53
5.5.3 Results 54

6 Data Partitioning - Primary Copy Placement 59

6.1 Primary Copy Placement 60

6.2 A Simple Affinity Approach to Primary Copy Placement 62

6.3 A Greedy Approach to Primary Copy Placement 64

6.4 Primary Copy Placement as Hypergraph Partitioning 66

6.4.1 Building the Hypergraph 67

6.4.2 Hypergraph Partitioning and Retrieving the Placement 70

6.5 A Linear Programming Solution to Primary Copy Placement 72

6.6 Experimental Evaluation of Primary Copy Placement 74

6.6.1 Locality 75

6.6.2 Hot/Cold Tuples 77

6.6.3 RUBiS, Once Again 80

6.6.4 Greedy Iterations 82

6.7 Primary Copy Placement Conclusions 83

7 Secondary Copy Placement 85

7.1 Secondary Copy Placement Example 88

7.2 Problem Complexity 89

8 Conclusions 93

References 95

Appendices 107

Appendix A: Site Database API 109

A.i Garbage Collection 111

A.ii New Copy Operations 111

Appendix B: SpearDB PSI Guarantee 113

B.i No Write-Write Conflicts 113

B.ii Causality 116
B.iii Transaction-Consistent Database Snapshots 118
Appendix C: Optimized Greedy Primary Placement 123

List of Figures

1.1 Map of Amazon Web Services Datacenter Locations 2

1.2 Points-of-Presence for CDNetworks Provider 3

3.1 SpearDB System Architecture 15

3.3 Some Common Database Consistency Guarantees and Their Features . . . 19

4.3 `BEGIN_TX`, `GET`, `PUT`, `DELETE` at site `S` 28

4.4 `COMMIT_TX` and `ABORT_TX` at Originating Edge Site `E` 29

4.5 SpearDB Commit Decision 29

4.6 Commit Procedure for Single Site Commits at site `S` 30

4.7 Installation of Committed Transaction at site `S` 31

4.8 `coreCommit` Operation at `CORE` 31

4.9 Two-Phase Commit Coordination 32

4.10 `CALL` at site `S` and callCore at `CORE` 34

4.11 Secondary Replica Creation and Removal at Edge `E` 38

4.12 Core to Edge `E` Ownership Transfer 40

4.13 Edge `E` to Core Ownership Transfer 40

5.1 Commit Strategies Experiment Setup 45

5.3 Effects of Commit Strategy and Network Latency (L) on Transaction Response Times 46

5.4 Locality Experiment Setup 47

5.5 Goodput as a Function of Locality 48

5.6 Transaction Abort Rate as a Function of Locality 48

5.7 SpearDB and Walter Response Times 50

5.8 SpearDB and Walter Abort Rates 51

5.11 RUBiS Database Response Time by Page and Page Type @ 80% Locality . 55

5.12 RUBiS Database Average Response Time by Locality 57

6.3 Simple Affinity Primary Placement Algorithm 63

6.4 Greedy Primary Placement Algorithm 65

6.5 Example of Greedy Primary Copy Placement Stuck in Local Minimum . . 66

6.6 A Two-Tuple Transaction Expressed as a Hypergraph for Primary Copy Placement 68

6.8 Hypergraph for the Example Workload and Network 71

6.9 Remaining Hyperedges After Min-Cut for Hypergraph 72

6.10 Simplified Cost Primary Copy Placement Problem 73

6.11 Primary Copy Solution Quality Comparison for the Locality Workload . . 76

6.12 Primary Copy Algorithm Runtime Comparison for the Locality Workload . 76

6.13 Locality Workload Size as a Function of P 77

6.14 Primary Copy Solution Quality Comparison for the Hot/Cold Tuples Workload 78

6.15 Primary Copy Algorithm Runtime Quality Comparison for the Hot/Cold Tuples Workload 79

6.16 Hot/Cold Tuples Workload Size as a Function of S 80

6.17 Greedy-Core Iterations as a Function of S for the Hot/Cold Tuples Workload 80

6.18 Primary Copy Solution Quality Comparison for Subsets of the Rubis Workload 81

6.19 Primary Copy Algorithm Runtime Scaling for the Rubis Workload 82

6.20 Example Greedy Solution Quality and Solving Times as a Function of Algorithm Iterations 83

7.2 Secondary Copy Placement Problem at Edge e 87

A.1 SiteDB Entry List Example for some Key 109
B.1 Equivalent pseudocode portions of all transaction commit branches at a site `S` or the `CORE` for a single `key` transaction 114
B.2 Pseudocode portions of all branches that can lead to a site `S` or the `CORE`'s `commitVTS` to be updated 119
B.3 `GET` at site `S` 121
C.1 Optimized Greedy Primary Placement Algorithm 124

List of Tables

3.2 SpearDB Application Programming Interface 18
3.4 SpearDB Site Database Interface 23

4.1 SpearDB Per-Site Metadata 26
4.2 SpearDB Transaction Descriptors 26

5.2 Commit Strategies Experiment: Data Placement 45
5.9 SpearDB Schema for RUBiS 52
5.10 RUBiS Experiment Network Round Trip Latencies 53

6.1 Primary Copy Placement Input Components 60
6.2 Example Primary Copy Placement Input and Corresponding Output . . . 61
6.7 Outcome of All Possible Hyperedge Combinations for a Single Transaction 69

7.1 Secondary Copy Placement Input Components 86
7.3 Example Secondary Copy Placement Setting 88
7.4 Added Traffic and Latency Savings for Example Input 88
7.5 Latency Savings per Added Bandwidth Unit for Example Input 89

Chapter 1

Introduction

An overwhelming number of new applications and services are being deployed directly to the cloud. The elasticity, automatic provisioning and scaling, and low entry cost of the cloud, as well as the instant access to a global audience, make this platform a favorite for start-up companies and the applications they develop. But this global scale comes with its own interesting set of problems.

1.1 Latency

Request latency is one such critical issue for modern cloud-hosted web, mobile and software-as-a-service applications. Numerous studies [7, 61, 44, 85, 24] have shown the importance of low latency to the end user experience: For example, Bing found that a 2 second slowdown in response time reduced queries per user by 1.8% and revenue per user by 4.3% [85]. Google Search found that a 400 millisecond delay resulted in a 0.59% reduction in searches per user. Even after the delay was removed, these users still had 0.21% fewer searches [85]. To address this problem, applications are heavily engineered to adhere to latency budgets. However, the distance between end users and application servers, what Leighton refers to as the "middle mile" [50], represents a fundamental challenge to low latency.

One response to this problem is to move the server side of the application closer to end users. Cloud infrastructure providers offer services in different geographic regions. The application can run in multiple regions, with the application's database partitioned or replicated and distributed across the regions. However, in addition to the complexity and overhead introduced by partitioning and replication, the level of granularity that these

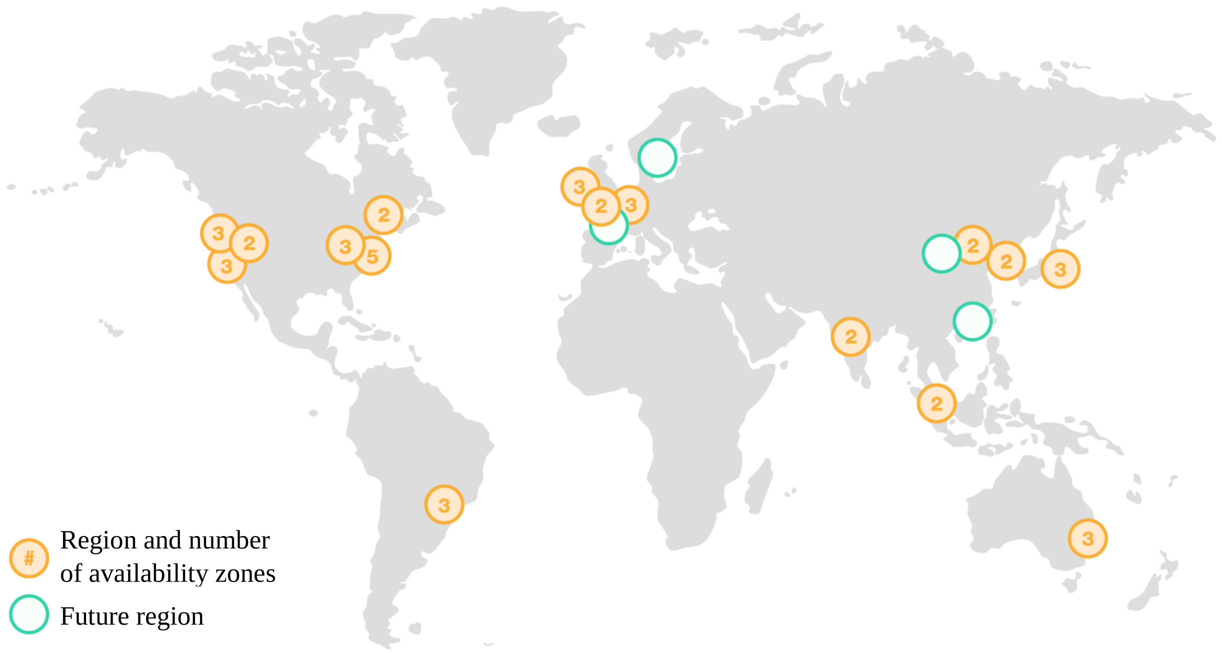


Figure 1.1: Map of Amazon Web Services Datacenter Locations as of June 2017 (16 current locations, 4 future locations)

regions typically provide doesn't fully address the middle mile problem as there may still be significant latency between end users and the nearest application service. Figure 1.1 shows the location of every datacenter Amazon Web Services makes available to clients that wish to host their web applications. Amazon Web Services is one of the largest cloud service providers, yet even in their densest zones (the US) there are many locations for which the closest datacenter is thousands of kilometers away. The cost, scale and power requirements of full datacenters makes it unlikely that these *core* regions cloud service providers make available will significantly increase in number in the future.

For static content, the latency problem has been addressed by content delivery networks (CDNs). CDN points-of-presence (POPs) can easily number in the hundreds as seen in Figure 1.2, and their density guarantees that most users are very close to at least one of them. These CDN points-of-presence are often referred to as *the edge of the cloud* because of the location they occupy in the Internet's network topology. The role of CDNs is to cache frequently used static application data closer to users. Such cached data is expected to rarely change and when it does it is updated by a central publisher and pushed out to a large audience of subscribers (i.e. applications) that are scattered over large geographic

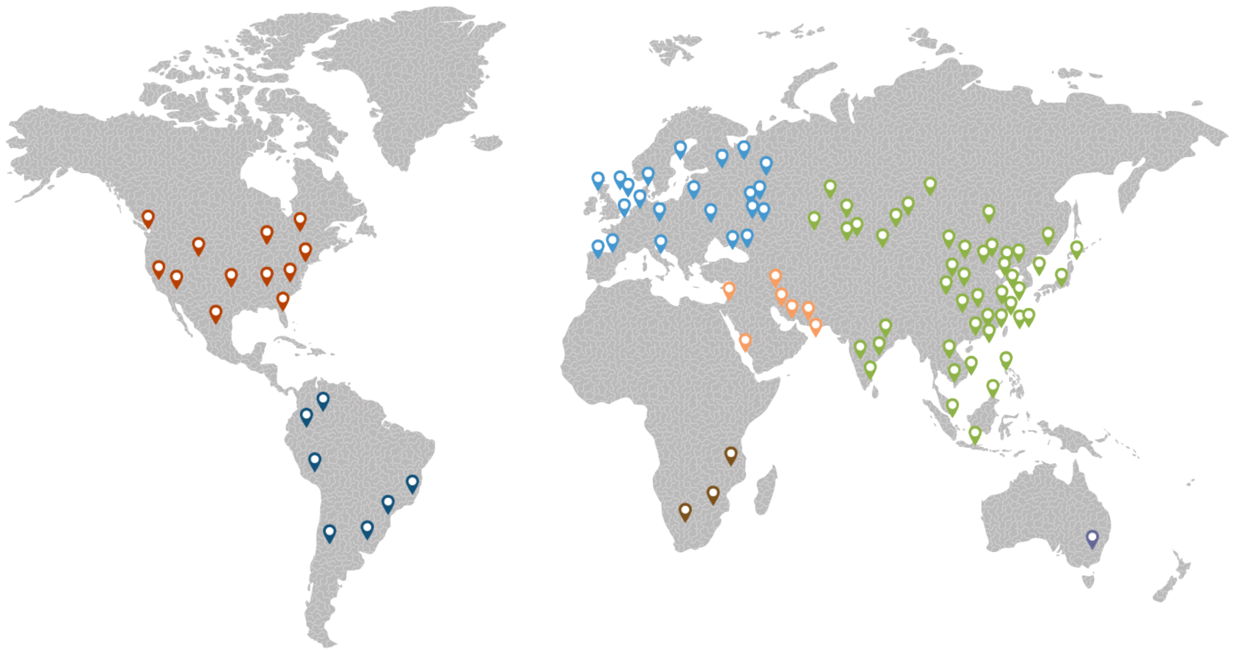


Figure 1.2: Points-of-Presence for CDNetworks Provider as of June 2017
(200 PoPs in 100 Cities)

areas. Movie streaming applications as well as static web content (e.g. images, style-sheets, fonts, JavaScript libraries) are typical use cases for CDNs.

Mobile edge computing [9] carries this trend even further. Mobile edge computing refers to pushing computing services from central cloud datacenters to the edge of the network - either to provide very low latency, or to provide the application with access to information that is available only at the edge, such as information about the state of the wireless connection to the client devices.

Modern general purpose applications, however, make heavy use of dynamic and user generated content through commenting and rating systems, sharing and social features as well as targeted, interest or location based advertising or news update delivery. All of these features that users have come to expect can't be fulfilled by the simple caching paradigm of CDNs.

A better solution may be to push the entire application all the way to the edge of the network. The server side of an application can be replicated at hundreds, or perhaps thousands, of locations, with each location responsible for serving nearby end users in a small geographic region, such as a metropolitan area. This has the advantage that users can be as close to the application as they are to CDN points-of-presence. Such a solution is well suited for latency critical applications.

1.2 Edge Applications

The key challenge in pushing applications to the edge of the network is the database. Pushing the application to the edge without pushing the database will not solve the latency problem, since the application will need to communicate with the core for database access.

The problem addressed in this thesis is how to provide database management for widely distributed edge applications. My thesis is that, by exploiting data replication and carefully designed synchronization protocols, it is possible to provide transactional database services to edge applications without re-introducing the latency that has been eliminated by moving the application to the edge.

The focus is on latency sensitive applications that provide *geographically localized services*. This means that, for a given geographic area, there are parts of the application's database that are used primarily by end users in that area. For example, consider a service such as Uber, which operates globally, but which has a substantial local component. That is, Uber passengers and drivers typically use the system in the city or region in which they live, and thus a substantial fraction of the application's activity involves interactions among users within a single region. Many other applications also fit this criteria: bulletin boards (e.g., Craigslist, Kijiji), local commerce or services (e.g., Groupon, Uber), booking and ticketing (e.g., OpenTable, StubHub), location based services (mapping, directions, augmented reality), local news outlets and client-centric services (e-mail, rss feeds, gaming).

1.3 Thesis Organization and Research Contributions

Chapter 2 describes related work. Chapter 3 presents an overview of SpearDB - a distributed, replicated database system that is designed to support edge applications. SpearDB is based on an asymmetric core/edge architecture. Chapter 4 describes SpearDB's protocols for read/write transaction execution. These protocols are designed to enforce strong consistency guarantees while minimizing application latencies. Specifically, they are based on the protocols used by Walter [88] and provide the same *Parallel Snapshot Isolation* (PSI) consistency guarantees over a geo-replicated database. These protocols have been revised to take advantage of the asymmetric core/edge architecture, and introduce additional features, such as stored procedures with transaction shipping. Further protocol extensions allow for live data migration, which may be used to further improve latency for workloads that change over time.

The SpearDB protocols have been implemented in a prototype which is architected as a middleware layer that sits above a local database system at each edge site as well as the core. It is responsible for all inter-site communication, and exposes an extensible record store interface to application programs. Chapter 5 presents an empirical evaluation of the SpearDB prototype. The various commit strategies used by SpearDB are shown to be well-suited to the geographically localized workloads for which SpearDB is targeted. SpearDB's PSI protocol is further compared to a baseline modeled after the original PSI protocol implemented in Walter [88].

The performance of SpearDB depends on the placement of primary and secondary copies of data at the core and edge sites. Suitable placements are application-specific and depend on the workload. In Chapter 6 and Chapter 7 the problems of primary copy placement and secondary copy placement are studied. Several algorithms are developed to provide approximate solutions. The effectiveness of the placement strategies produced by the algorithms is evaluated using different workloads.

Chapter 2

Related Work

This chapter describes other research efforts that have similar or complementary goals to those of this thesis. Works that approach the problem of latency are described in Section 2.1. Section 2.2 discusses research on content delivery network and caching. Related database research is discussed in Section 2.3, while Section 2.4 covers related work on the subject of data partitioning.

2.1 Latency

Recent studies [44, 61, 35, 10] have found that even a small increase in web page load times can substantially increase the likelihood that a customer would switch to a competing service or store causing loss of revenue. Several programming and testing principles have been proposed [43, 34] to help in building better web applications that try to use the least amount of network latency necessary. There is a significant amount of research published on the topic of network latency and its effects [11, 13, 82, 101, 54, 91, 15]. This body of work showcases the importance of trying to reduce network latency for the end product (a web application or service) so that it performs to its users' expectations. Since retrieving database results is often an integral step in answering client requests in these applications and services, it follows that this process needs to be as efficient as possible.

A variety of tools and techniques are available for latency and performance analysis. Server side performance analysers and stress testing tools like the commercial solutions offered by Neustar [65] and HP's Performance Center [37], as well as open-source projects like Benchlab [13], allow application-specific synthetic workloads to be tested under a variety of realistic load conditions. Several network link emulators are available [67] to test different network conditions and topologies. For measuring client-side page load times, Rajamony and Elnozahy [82] present a technique that embeds measurement JavaScript directly in a downloaded web page.

Butkiewicz et al. [11] present a detailed analysis of the complexity of modern web pages showcasing how web page features, such as the number of objects retrieved, the types of content retrieved, and the number of sources from which content is retrieved, are related to the time required to render the page. The WProf [100] in-browser profiler is used to identify the number of round trip times as the major limiting factor in loading a web page. While caching can lessen the problem, alternative network protocols to HTTP (like SPDY [33]) are found to have a higher impact in reducing the network congestion. Wei and Xu [101] describe a technique for monitoring network traffic into and out of a web service, while the WebProphet system [54] is able to predict the load time of a web page under hypothetical conditions through client-side (*"what-if"*) latency analysis. The "What-If Scenario Evaluator" (WISE) [91], on the other hand, takes a machine learning approach to model the effects of changes in a web service's deployment configuration on the server side.

Chen et al. [15] focus on the effect of changes in the inter-tier latencies in a multi-tier service architecture. They propose a metric called the *link gradient* to capture the effect of the latency between two tiers on the overall performance of the system.

2.2 Caching and Content Delivery Networks

As mentioned in Section 1.1, content delivery networks (CDNs) are a class of systems that use the edge of the cloud to reduce end-user latencies. CDNs, like geo-distributed caches [25, 62, 27], are used primarily for relatively static data, with changes being pushed from a central content provider towards end users at the edges.

There are many uses for CDNs. However, in this paradigm, both the application and the application's database remain at the core. Feng et al. [56] present CDNs as a group of caching servers behind a load balancer; and their usage is motivated as a performance and scalability improving technology that reduces the Slashdot effect - the slowdown or crash of a small website as a result of a massive spike in traffic caused by a popular website linking to the small website. An alternative load balancing algorithm for choosing a server in a CDN is discussed in [99]; CDNs are motivated as ways of improving availability, response time and overall system throughput while avoiding overloaded servers and congested networks.

An interesting alternative to CDNs is presented by Yuan et al. [105]. The authors propose to use the already existing proxy server infrastructure to perform either caching of dynamic content or computation offloading. However at the time the paper was written, proxies were more commonly available than CDN nodes - a situation that is no longer true today. The other contribution of Yuan et al. [105] is a comparison of the benefits of offloading data and computation versus the caching of dynamic content and building dynamic web pages locally. The results seem to be in favor of the simpler caching technique, but at the time of writing, dynamic user-generated content was uncommon and is not explored in the paper. A dramatic shift in the nature of workloads that web pages see has taken place a few years after this research has been published and is attributed to the explosion of social networks.

Passarella [73] provides a very good survey of CDN-related papers, which goes all the way back to the 1960's when these networks were first being discussed.

2.2.1 Edge of the Cloud

Dragonfly [45] is a media delivery system similar to CDNs but working on top of peer-to-peer (*P2P*) networks. The system uses 3-tiers: the first two tiers are a cloud core and a network of edges, while the last tier consists of a P2P dynamic overlay of locally clustered user groups. IBM's Websphere Edge Server [40] product offloads parts of an application to different caching servers either within a datacenter or on CDN nodes, but database access is treated as a complementary problem left to be solved by the application developers. Several database deployment scenarios are suggested, all of which focus on caching read-only query results. A more limited approach is proposed by the DynCoDe system [79] which would only allow edges to construct dynamic content by concatenating a permutation of static content fragments. Several other projects [105, 5] have considered an architecture in which all or parts of the application are pushed to the network edge to reduce request latencies. However, all of these systems only provide a non-transactional database cache at the edge.

More generic approaches in projects such as the SAVI [66] network or the ActiveCDN [83] system allow for dynamic runtime provisioning of general purpose computational resources at edges, upon which custom code may be executed. This is similar to the on-demand virtual instance provisioning that cloud service providers offer [4, 32, 12], but at a more granular level. Such cloud edge computational resources are essential for the type of system proposed in this thesis.

2.2.2 Economics of the Cloud-Edge

A relatively lesser known fact about content delivery networks is that, aside from latency savings to the end users, they provide economic benefits to the cloud edge hosting companies that put up the hardware necessary to run them. Avoiding network trips to the core by provisioning a local cache of data provides monetary savings in the form of lower bandwidth costs for the edge operator.

Vleeschauwer et al. [98] discuss optimum caching strategies for telecom CDNs. This work looks at the traffic patterns within CDNs based on different network topologies and caching algorithms, and motivates the use of CDNs to reduce stress on the networks due to increasing video content. The economic implications of cooperative caching (an array of distributed caches cooperating to serve each others' web requests) are discussed by Hosanagar et al. [36]. The study includes ways of incentivizing participation as well as the situation of cache peering spanning organizations. The Globule system [76] is designed to create ad-hoc CDNs on top of a donation-based peer to peer (*P2P*) network of resources. Content providers are able to group together and form their own collaborative CDN by virtually using roughly the same amount of resources they would to host their one single service. The paper explains and deals with the high-churn environment and unreliable participants.

These works are complementary to the goals of this thesis as there are similar economic implications to a distributed database capable of saving network bandwidth between the edge and the core.

2.3 Databases

The field of databases is very broad, with computerized databases dating as far back as the 1960s. However, this section will focus mostly on distributed databases as they are more closely related to the system proposed in this thesis.

2.3.1 General Purpose Distributed Databases

One important motivation for building distributed databases is to allow them to scale out in response to growth in data volumes or workloads. Systems built for this purpose are normally not intended to accommodate the high latencies that come with geographic distribution, and are deployed in a single location. Examples include Bigtable [14] and

HBase [93], Deuteronomy [51], research prototypes such as G-store [20], Centiman [23], Calvin [95], and Rococo [63], and commercial relational database systems [38, 12, 70, 71].

Other distributed database systems are designed to be deployed in geographically distributed environments. However, many such systems impose restrictions on the types of transactions that they support. Systems like Dynamo [22], Cassandra [49], Scatter [29], and PNUTS [17] offer atomic operations on single keys only. COPS [58] focuses primarily on single-key atomic operations, but does offer an extension that would allow for read-only atomic access to multiple keys. Eiger [59] additionally provides multi-key write-only transactions, and guarantees that transactions can run locally, but provides only causal consistency. Neither COPS nor Eiger permits arbitrary transactions that include both read and write operations. MDCC [46] allows multi-key transactions, but provides a relatively weak read-committed consistency guarantee.

On the other end of the spectrum are geo-distributed database systems that are designed to provide strong transactional consistency guarantees, such as snapshot isolation or serializability [18, 60, 8, 74]. The drawback of these systems is that such strong guarantees are expensive to provide in a geographically distributed setting. Both snapshot isolation and serializability require that all sites agree on the commit order of all transactions. Such agreement requires cross-site synchronization, and results in high latency for update transactions. In contrast, SpearDB provides PSI, a slightly weaker guarantee that allows sites to disagree on the ordering of non-conflicting transactions. This eliminates the need for cross-site synchronization in many cases, and thus lowering latency.

Walter [88] is closely related to the work in this thesis, in that it targets geo-distributed data and provides a parallel snapshot isolation consistency guarantee. Unlike Walter, SpearDB uses an asymmetric architecture, with a distinguished core site, because it is targeted at edge computing. Section 5.4 describes an adaptation of Walter's transaction protocols to the core/edge setting. This is then used as a performance baseline for SpearDB.

Gemini [52] is a geo-replicated system in which an application must distinguish operations for which ordering is important from operations for which it is not. Gemini then can avoid enforcing ordering for operations for which it is not necessary. As a result, Gemini, like SpearDB and Walter, allows some operations to have different execution orders at different sites. Gemini's approach, which relies on the application being able to identify operations that need not be consistently ordered, is complementary to SpearDB's, and could potentially be combined with PSI to further reduce the amount of cross-site synchronization which is required.

Lynx [108] takes a different approach to providing both low latency and serializability. Transactions consist of a chain of pieces, each running at a different site. Under some conditions, Lynx can safely acknowledge a transaction after execution of the first piece, resulting in low latency. However, this approach requires static analysis of transactions in advance to determine when early acknowledgment is safe.

There are a few systems that have focused on database management in an edge computing setting, like SpearDB. Lin et. al. [57] describe SEQ DB, which uses a core/edge architecture similar to SpearDB's. However, SEQ DB enforces one-copy snapshot isolation, which demands a total ordering of transactions. SEQ DB uses a centralized sequencer for this purpose, meaning that all transactions require cross-site synchronization. SwiftCloud [107] makes use of client-side caches in a manner similar to the way that SpearDB uses edge sites. However, because SwiftCloud is concerned with disconnected client operation, it allows database states at different clients to diverge as a result of write-write conflicts.

2.3.2 In-Memory Databases

Many databases, including distributed ones [6, 55, 72] are resorting to in-memory data storage to reduce latency. This move is motivated by the price of memory decreasing and the available sizes increasing over time. Using in-memory storage is a complementary method of latency reduction, however the focus in this thesis is on the latency caused by network traffic rather than the individual machines running the database.

2.3.3 Other Levels of Abstraction

The Sinfonia system [3] proposes a distributed shared memory system to be used as a basic block for building other distributed systems while bypassing any complicated message passing algorithms. The novel concept being proposed is the "minitransaction" primitive which allows efficient and consistent data access, abstracting away the processes that deal with concurrency and those that handle failures. Aguilera et al. [2] build a scalable distributed B-tree data structure on top of these minitransactions. Like SpearDB, these works simplify data storage in a distributed environment, but the focus is on lower levels of abstraction. However these could potentially be the basis for more complex structures, generic transactional support and even a full relational database management system.

Gao et al. [28] propose to address the challenge of data consistency and replication in a geo-distributed setting by taking advantage of a particular application's query needs in order to build custom DBMS implementations as part of the application logic. While this thesis also addresses this issue, the focus is on generic abstractions and protocols that are application-agnostic.

Conflict-free replicated data types [86] may alternatively be used in geo-distributed environments to eliminate the problem of conflicting operations and remote synchronization. However, operations are limited to what each individual conflict-free data structure allows. Furthermore, states at different sites may reflect different arbitrary subsets of the global changes until, eventually, convergence is reached.

2.4 Data Partitioning

Automatic data partitioning is often used as a means to increase workload throughput. However, many proposed solutions [1, 104, 17, 103, 77, 78] don't account for multiple tuples/objects being accessed or modified by atomic transactions, since transactions are not supported by the underlying system the data is being partitioned for. This type of data partitioning can be used for load balancing server resources.

Data partitioning that accounts for multi-access atomic transactions is an NP-hard problem. There are several proposed solutions [19, 97, 64, 30, 90, 89, 96, 53, 75, 81] that address this but that don't account for data locality (i.e. proximity of data to its users). This allows data partitioning to inform placement decisions that reduce cross-server communication in addition to resource load balancing, but leaves the client-server communication path unoptimized.

The Spar system [80] could allow for a locality optimized partitioning, but the workload being optimized for is very specific to social networks and limited to transactions that touch a node in a graph and one or more of its direct neighbors.

Works like DB-risk [106] or that proposed by Sharov et al. [87] look at server role allocation in geo-distributed databases. These systems are not concerned with the placement of data as they work in a full-replication environment that supports general purpose transactions which are committed via quorum decision. The issue being solved by these works is the placement of voting replicas and the primary replica (or quorum coordinator) in relation to clients in order to minimize response time. SpearDB needs to solve the primary replica placement problem as well, but given the differences in commit protocols among these systems, the search space of possible solutions is entirely different.

Chapter 3

SpearDB Overview

To explore how to provide database support for edge applications, a prototype database system is proposed. This prototype is called SpearDB, which is an acronym for *Self-Partitioning, Edge-Aware, Replicated DataBase*. SpearDB has been designed to achieve several objectives. First, SpearDB should provide full transactional support for general purpose transactions. Second, it should provide a global view of the database and strong consistency in order to simplify application development. Finally, it should provide low latency in a geo-distributed environment for a specific class of applications: geographically dispersed applications with geographically localized database access patterns.

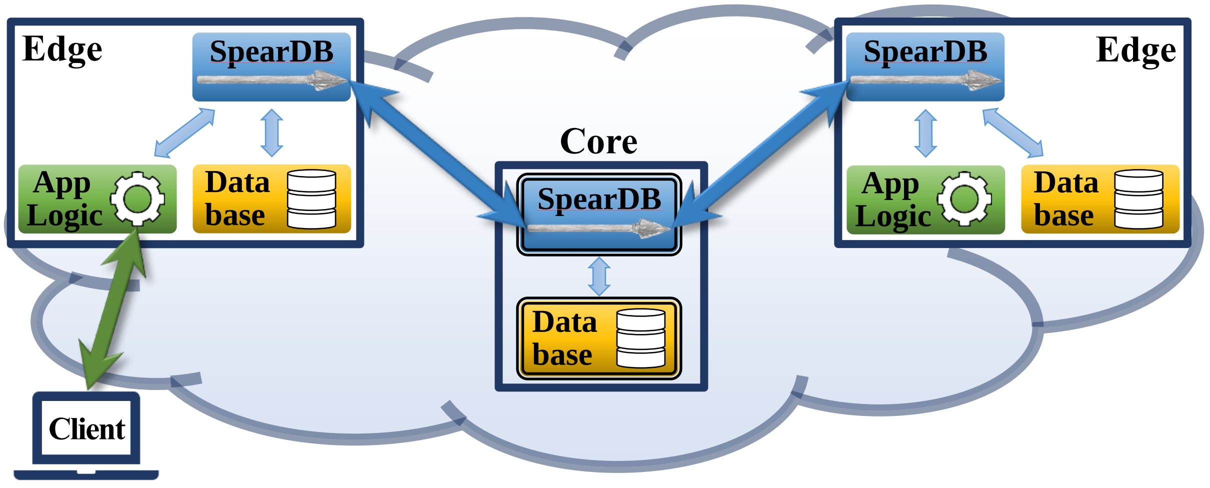


Figure 3.1: SpearDB System Architecture

Figure 3.1 illustrates SpearDB's architecture, which consists of a central core location with many edges directly connected to it. These edge and core sites form a star network topology over a large geographic region. As there are many edge locations, most end-users are in close proximity to at least one edge. The large geographic coverage of the edges also means that many of them are geographically distant from the core, and latency between edges and the core may be relatively high. For a continental U.S. deployment, round trip latencies between an edge and the core may be as high as a hundred milliseconds.

To provide persistent storage, SpearDB maintains a local database at each edge site and at the core. It replicates and distributes the application's data among these local databases. Ideally, SpearDB distributes the data to workload-appropriate locations such that the local SpearDB server can service application requests from the local database. This would eliminate the need to read data from other sites. However, because of updates and the need to provide a globally consistent view of the database, it may be necessary for a SpearDB edge server to communicate with other sites in order to satisfy an application request. Except for an increase in request latency, any such communication is transparent to the application.

SpearDB is designed to support client-server applications with geographically dispersed clients. The server side of the application, shown as *App Logic* in Figure 3.1, is replicated at each edge site. A client connects to the application server at its closest edge. The application server accesses the database by interacting with the local SpearDB edge service running on the same edge.

In SpearDB, each database record has exactly one primary copy at some site, and zero or more secondary copies at other sites. Primary copies are different from secondaries by virtue of the fact they are always modified first when the record changes. The SpearDB core site always maintains a copy of every record, although it may be either the primary copy or a secondary. In general, each SpearDB edge site will hold only a partial copy of the database consisting of a mix of primary and secondary copies of different records. SpearDB's core instance tracks the locations of the primary and secondary copies of all database records. In contrast, each edge instance is only aware of the records held at its location.

3.1 Characteristics of SpearDB

SpearDB has several salient characteristics. First, the primary goal of replication in SpearDB is to provide *low latency database access* by placing data at the edge sites, close to end users. Cross-site replication can also potentially be used to improve system availability and durability in the event of site failures, but that is not the main focus. Instead, SpearDB relies on the availability and durability provided by the local database system at each site. These local database systems may themselves replicate and distribute data (locally) to ensure that the local database is durable and highly available.

Second, because SpearDB is intended to support a large number of edge sites, it uses an *asymmetric architecture*. The core site in SpearDB plays a special role. It maintains a copy of the entire database, and it acts as an intermediary in any communication involving multiple edge sites. Having a core simplifies the management of sites and placement of data in SpearDB. Because of the core, SpearDB need not maintain a distributed data directory. Each edge site is aware of the data that it holds locally, and finds any other data through the core. The core also serves as a natural home for portions of the application database that have non-localizable access patterns. Finally, the core can also directly serve application end users for which there is no nearby edge site.

Third, operations in geographically localized services commonly include database updates, such as those induced by bookings, reservations, movement tracking, and "social" features such as user feedback and comments. Read-only database access at the edge is not sufficient to support such features. Thus, SpearDB supports *database updates at the edge*.

Finally, although SpearDB uses multi-site database replication to achieve low latency, replication is transparent to the application. Thus, from an application server's perspective, the entire database is readable and writable through the local SpearDB service at its edge. Applications "see" a single, global, logical database, and SpearDB provides a *global transactional consistency guarantee*, namely *parallel snapshot isolation* (PSI) [88], which is similar to the more familiar snapshot isolation (SI). PSI is further explained in Section 3.3.

3.2 Database Interface

SpearDB is a key/record store with a single index on the records' primary keys. It provides a GET/PUT interface on individual records and SpearDB applications perform GET and PUT requests in the context of atomic transactions. Each transaction starts with a BEGIN_TX operation, followed by an arbitrary sequence of record operations, and ends with a COMMIT_TX or ABORT_TX. SpearDB also allows transactions to be executed on the server side as stored procedures, via the CALL operation. Table 3.2 summarizes SpearDB's application interface.

Table 3.2: SpearDB Application Programming Interface

Operation	**Description**
BEGIN_TX	Starts a new transaction
GET	Retrieves a record given a primary key
PUT	Installs a new record or a new version of an existing record identified by a primary key
DELETE	Removes a record identified by a primary key
COMMIT_TX	Attempts to commit a transaction
ABORT_TX	Aborts a transaction
CALL	Executes a transactional stored procedure identified by name and given a list of parameters

In addition to allowing applications to retrieve and modify entire records, SpearDB's GET/PUT interface also supports retrieving and modifying the values of individual record columns or groups of columns. However, to simplify the presentation, only a more basic interface is described through which an application must GET/PUT entire records.

3.3 Consistency

SpearDB provides global Parallel Snapshot Isolation (*PSI*) [88] to each application transaction. Similar to Snapshot Isolation (*SI*), PSI ensures that each transaction's GET operations "see" a transactionally-consistent snapshot of the database, and that there are no concurrent update conflicts from PUT or DELETE operations. However, PSI is weaker than snapshot isolation in that different sites are permitted to order non-conflicting transactions differently. Therefore, PSI is better-suited than SI for SpearDB's intended deployment setting as it allows SpearDB to avoid expensive synchronous cross-site coordination for such transactions. Transaction processing in SpearDB is discussed in more detail in Chapter 4.

Figure 3.3 presents a list of commonly used database consistency guarantees. They are all related to PSI and arranged in a spectrum. On one end of this spectrum is serializability. This consistency level was first defined in the ANSI/ISO SQL standard [26] and is the strongest level of consistency available. Transactions executed at this consistency level appear as if they had been executed in a serial order. However because of the synchronization overhead required to attain this consistency guarantee, a weaker model is often used in practice even in monolithic (i.e. non-distributed) databases. In geo-distributed systems in particular, the large distances between node clusters make serializability infeasible due

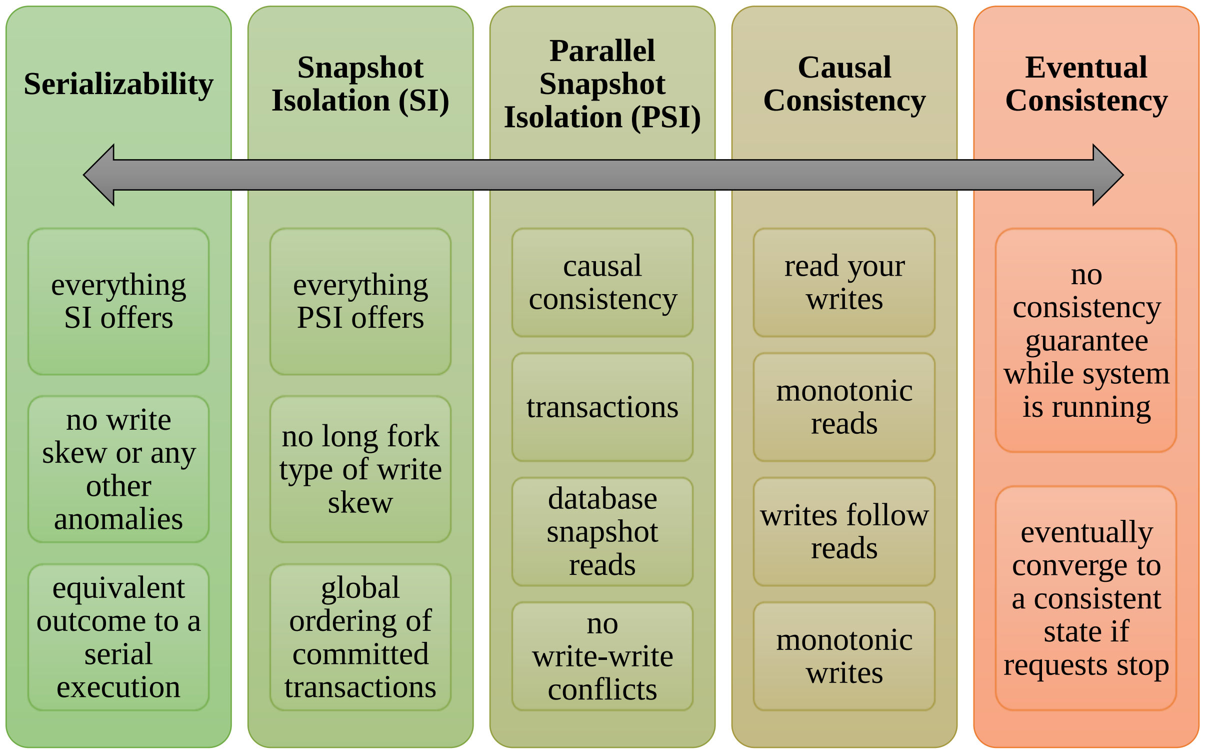

Serializability
Snapshot Isolation (SI)
Parallel Snapshot Isolation (PSI)
Causal Consistency
Eventual Consistency
everything SI offers
no write skew or any other anomalies
equivalent outcome to a serial execution
everything PSI offers
no long fork type of write skew
global ordering of committed transactions
causal consistency
transactions
database snapshot reads
no write-write conflicts
read your writes
monotonic reads
writes follow reads
monotonic writes
no consistency guarantee while system is running
eventually converge to a consistent state if requests stop

Figure 3.3: Some Common Database Consistency Guarantees and Their Features

to high latency and low throughput.

On the other end of the spectrum, eventual consistency is an umbrella term for different consistency levels where the database may be in an inconsistent state whenever a workload is being executed. If requests stop coming in to the database system for *a while*, the database will eventually converge to a consistent state, although no guarantees are usually provided on the time of the convergence. Eventual consistency is generally regarded as a low level of consistency.

Snapshot Isolation (*SI*) [21, 102], from which PSI is derived, is a strong consistency model where reads appear to happen from a snapshot of the state of the database at the beginning of the transaction. Write-Write conflicts are explicitly disallowed at commit time. In practice, SI is implemented using multi-version concurrency control (*MVCC*) where multiple versions of a tuple/row are maintained in order to create the illusion of a snapshot of the database being taken for every transaction. MVCC allows for much higher concurrency in executing workloads than the serializable level would allow. Because of the relatively strong consistency and high concurrency, SI is a very popular consistency level in database management systems.

When compared to serializability, snapshot isolation exhibits a concurrency anomaly called *"Write Skew"*. This anomaly occurs when two transactions concurrently read overlapping sets of tuples/rows and each perform updates to *different* subsets of the tuples/rows that both have previously read. Since the writesets are different, both transactions may successfully commit without having seen each other's updates. The potential inconsistencies arising from the write-skew anomaly can be avoided in arbitrary transaction groups by adding additional updates that would force a write-write conflict instead of a write-skew.

Parallel Snapshot Isolation [88] is derived from SI. It is a strong consistency level (although weaker than SI), and it is designed to work specifically in a replicated distributed environment with primary and secondary copies at different *sites*. Both PSI and SI provide transactions with a transaction-consistent database snapshot to read from, and both prohibit concurrent updates to database records. While in SI a transaction is guaranteed to see all committed transaction as of its start point, PSI only guarantees that a transaction will see all transactions that have committed at the same site. This mechanism allows database states at different sites to diverge for longer periods of time (*long fork*), while SI databases need to globally synchronize their state after every successful commit and thus only allow database states to diverge for concurrently running transactions (*short fork*). Both long and short fork are different flavors of the write-skew anomaly.

Furthermore, because of its global synchronization point, SI provides a globally agreed upon ordering of transactions. PSI, on the other hand, only guarantees a global partial ordering that is similar to the guarantees provided by the weaker causal consistency model. As shown in Figure 3.3, causal consistency provides four session guarantees, as defined by Terry et al. [92]:

1. **read your writes:** preceding writes in a session are seen by following reads;
2. **monotonic reads:** read operations see an increasingly up-to-date version of the database - i.e. reads see all writes that have been seen by previous reads within the session;
3. **writes follow reads:** writes made during a session are globally (i.e. at every site) ordered after any writes that have been seen by previous reads in the session;
4. **monotonic writes:** writes must follow previous writes within the session.

Causal consistency, however, is defined for individual operations within a session. PSI further adds the concept of atomic transactions and defines the causal ordering between them that must be observed globally at all sites.

To summarize, SpearDB provides a PSI consistency guarantee which requires that transaction execution satisfies the following three properties:

1. **causality:** Under PSI, different sites are free to order transactions differently, provided that causality is preserved across all sites. This means that if a transaction T_a commits at a site before T_b starts at that site and T_b eventually commits, then it must commit after T_a at every site.

2. **transaction-consistent database snapshots:** Each transaction T that starts at site S reads from a transactionally consistent snapshot of the database that includes the effects of all transactions that commit *at site* S before T starts, plus any updates that are made by T itself. T's snapshot must not include the effects of any transactions that were not committed at S. Since different sites can commit non-conflicting transactions in different orders under PSI, a transaction's snapshot depends on which site it starts at.

3. **no write-write conflicts:** Concurrent transactions may not update the same record. Two transactions T_a and T_b are concurrent if T_a has started before T_b has committed at T_a's start site, **and** vice versa: T_b has started before T_a has committed at T_b's start site.

3.4 Edge Localization

A SpearDB transaction is *localized* if the primary copies of all of the records it updates are co-located at the same site. Furthermore, a localized transaction is *edge-localized* if the primary copies of these records are at the edge site from which the transaction request originated. SpearDB is designed for applications that mostly issue localized or edge-localized transactions. Whether or not a transaction can be localized depends on the properties of the transaction and the placement of replicas at SpearDB sites.

Consider the case of a web service such as Groupon, which allows business owners to promote their business by offering discounts to large groups of customers. Local businesses advertise through Groupon's platform in order to reach customers in their community. Suppose that Groupon is using SpearDB and primary copies of user records are placed at the SpearDB site closest to the users' homes. Furthermore, suppose records describing deals offered by local businesses are also placed at the SpearDB sites near those businesses. For the common case of users participating in a local deal, the resulting transaction can be edge-localized. Non-localized transactions, such as a user looking for a deal in a remote location for an upcoming vacation, are also possible in SpearDB. However, these transactions would have higher response latencies than localized transactions for local customers. To simultaneously support localized, edge-localized and non-localized transactions, SpearDB uses a different commit protocol for each transaction type.

Placement of replicas is important for performance in SpearDB since primary replica placement impacts commit strategy and secondary replica placement determines whether GET operations can be performed locally. Replica placement optimization for SpearDB is discussed in Chapter 6 and Chapter 7 along with automated tools to recommend such placements. Similar tools already exist for other related settings [16, 19, 31, 48].

3.5 Site Databases

SpearDB does not itself store application data. Instead, persistent storage of application data is provided by *site databases*, with one such database at each site, including the core site (see Figure 3.1). SpearDB serves as a middleware between the application and the site databases, and is responsible for managing replication, coordinating transactions, and handling all inter-site communication. Each site database operates independently.

SpearDB requires limited functionality from a site database. A site database can be implemented using various different types of database systems, ranging from relational database systems to scale-out key-value stores like Cassandra [49] or BerkeleyDB [68] SpearDB's main requirement is that the site database can store multiple versions of a record for each key, and each version carries an associated version number made up of a $\langle$commit site, sequence number$\rangle$ pair. In addition, SpearDB expects to be able to install a set of record versions atomically in the underlying database. Table 3.4 illustrates the required interface that a site database must support. The READ, WRITE and FLUSH operations are used during regular transaction processing, while READALL and MERGE are used to install additional copies of records at edges.

For the regular transaction processing operations, READ returns the the record version visible at a given vector timestamp for a given key. The WRITE operation atomically appends new record versions to the lists of a set of keys. A set of key/value pairs (records) is provided as input. A commit site and sequence number are also provided as input. These

Table 3.4: SpearDB Site Database Interface

Operation	Description
READ	retrieve the latest record version of a key that has been visible at a given timestamp
WRITE	atomically append new record versions for a given set of keys
FLUSH	ensure that all previous WRITEs are durable
READALL	retrieve all record version of a key from the site database
MERGE	merge a foreign record version history of a key into another base version history by removing any versions occurring in both histories from the foreign history and prepending the foreign history to the base one

will serve as the version number of all newly appended record versions. Finally, FLUSH is used to ensure that previously-written record versions are durable in the site database.

In order to install a copy of a record at an edge, the READALL operation is used to read the history (all previous versions) of a full copy of the record. The MERGE operation is then used to write this history to a different partial copy of the record (at a different site). This procedure is further described in Section 4.5.1.

Some additional functionality, such as garbage collection of old versions, has been omitted from Table 3.4 to simplify the interface description. A detailed explanation of the site database API is provided in Appendix A.

The connection to the underlying database requires a simple driver that maps SpearDB's site database interface to the underlying database's own API. Several such drivers have been implemented: one for a custom in-memory database system, another for Berkeley DB Java Edition [69] database instances, and a third one for SQLite [94] databases. The SQLite driver can be used for any SQL-capable DBMS. Note that the choice of driver and underlying database may be made independently for each site in the system.

Chapter 4

Transactions in SpearDB

There are a number of technical challenges faced by SpearDB, such as how to partition, replicate and distribute the database to the various sites, and how to provide low latency access to the database while enforcing consistency guarantees. The focus of this chapter is the *transactional* aspect of SpearDB. Specifically, the assumption is that the application's database has already been replicated and distributed appropriately across the SpearDB sites. Chapter 6 and Chapter 7 describe the ways this replication and distribution can occur.

SpearDB applications can group GET, PUT, and DELETE operations together into arbitrary transactions, using the BEGIN_TX, COMMIT_TX, and ABORT_TX transactional operations shown in Table 3.2. All database updates within a transaction are isolated until the transaction commits, and become visible atomically at the commit point. As was noted in Section 3.3, SpearDB provides a parallel snapshot isolation (PSI) guarantee for all transactions. Appendix B presents a formal proof of this claim.

Like Walter [88], SpearDB uses version numbers and vector timestamps to enforce the PSI guarantees. A version number is a pair $\langle S, seqno \rangle$, where *seqno* is a sequence number generated at site S. In SpearDB, each transaction is assigned a version number by the site that is responsible for committing it. A vector timestamp is a vector of version numbers - one per site. SpearDB uses vector timestamps to describe which transactions' effects are included in the database at a particular site and point in time. For example, in a system with four sites, the vector timestamp $\langle 6,\ 2,\ 7,\ 2 \rangle$ describes a database snapshot that includes six transactions committed by the first site, two committed by the second site, seven by the third site, and two by the fourth.

Table 4.1 summarizes the metadata maintained by each SpearDB site for tracking

Table 4.1: SpearDB Per-Site Metadata

Variable	Description
`seqNo`	current site sequence number
`commitVTS`	vector timestamp describing current state of the site's database
`database{}`	virtual representation of the site's database consisting of a mapping of record keys to their version histories
`lCopies{}`	map of record keys that have a local copy at the current site; the values in the map can take one of three values: PARTIAL, SECONDARY or PRIMARY - indicating the type of the local copy
`pCopy{}`	mapping of record keys to the site that holds their primary copy; only used at the core
`sCopies{}`	mapping of record keys to a set of sites that hold their secondary copies; only used at the core
`storeProc{}`	mapping of globally unique names to methods implementing stored procedure logic

transactions, record ownership, copies' locations and enforcing PSI guarantees. In addition, when a site creates a transaction, it creates a *transaction descriptor* which holds metadata associated with that transaction. During commit processing, parts of the transaction descriptor are passed among the site(s) involved in commit processing. Table 4.2 shows the fields in each transaction descriptor. The notation `tx.f` will be used to refer to field `f` in the descriptor for the transaction `tx`.

Table 4.2: SpearDB Transaction Descriptors

Field	Description
`startSite`	site at which the transaction was submitted
`startVTS`	vector timestamp describing transaction's read snapshot at the start site
`updates`	list of updates made by this transaction
`status`	commit/abort status of the transaction
`commitSite`	site responsible for committing the transaction
`seqNo`	transaction's commit sequence number at the commitSite

4.1 Transaction Execution

SpearDB applications running at an edge site interact with SpearDB using the API that was shown in Table 3.2. Figure 4.3 shows how SpearDB handles creation of new transactions, as well as GET, PUT, and DELETE operations. Each transaction is initiated at an edge site, which is referred to as the start site of the transaction. When a transaction starts, its startVTS is initialized to its start site's current vector timestamp, commitVTS. This represents the database snapshot from which the new transaction will read. It includes the effects of all committed transactions that the start site is aware of at the time the new transaction is started. Update operations (PUT, DELETE) are also handled locally at the start site, since the application of these changes to the database is deferred until the transaction commits. Finally, GET operations are also handled entirely at the start site provided that that site holds a copy (either primary or secondary) of the key that is being read. The READ operation (Appendix A) uses the transaction's startVTS to select the correct version of the record from the start site's local database. If the record is not present there, the GET operation is sent to the core site, which holds a complete copy of the database and is thus always able to respond to the request.

4.2 Committing Transactions

As explained in Chapter 3, every record (identified by its key) has exactly one primary replica. Based on the locations of the primary copies of the keys in a transaction's writeset (i.e. all the records the transaction is updating), SpearDB uses one of four different techniques to commit the transaction as follows:

- **local commit:** In the (hopefully common) case in which the start site holds the primary copies of all keys updated by the transaction, SpearDB is able to commit the transaction locally at the start site, with no need for synchronous inter-site communication. This means that the start site need not wait for communication with other sites before acknowledging the transaction commitment to the application.
- **core commit:** If the primaries of all updated records are located at the core, SpearDB performs single-site commit coordination there. This commit method is well-suited for transactions involving parts of the database that are not easily localized at a single edge. A core commit requires a single synchronous message round trip between the start site and the core.
- **remote commit:** Remote commit is a single-site commit that is used if the primaries of all updated records are located at a single edge site other than the start site. This is useful for data that are typically updated from a particular edge, but are occasionally updated

```
function BEGIN_TX()
  tx ← new transaction descriptor
  tx.startSite ← S
  tx.startVTS ← S.commitVTS
  tx.updates ← ∅
  return tx

function GET(tx, key)
  if S.lCopies[key] = PRIMARY or S.lCopies[key] = SECONDARY
    # S has a copy of key that isn't PARTIAL
    val ← READ(key, tx.startVTS)
  else
    val ← remote READ(key, tx.startVTS) @ CORE
  apply updates from tx.updates to val
  return val

function PUT(tx, key, value)
  append ⟨key, value⟩ to tx.updates

function DELETE(tx, key)
  append ⟨key, DELETE⟩ to tx.updates
```

Figure 4.3: BEGIN_TX, GET, PUT, DELETE at site S

from other locations. Remote commit requires a single synchronous round trip between the originating edge and the commit edge, via the core site.

- **distributed commit:** In the general case in which a transaction updates records with primary copies at more than one site, SpearDB falls back to a distributed two-phase commit protocol to commit the transaction. Distributed commits in SpearDB are always coordinated by the core site.

Figure 4.4 gives an overview of COMMIT_TX and ABORT_TX operations in SpearDB. If local commit is not possible, the start site passes the transaction to the core site. The core, which has complete information about the locations of the primary copies of all records, is then responsible for determining which of the three remaining commit methods will be used to commit the transaction.

Figure 4.5 illustrates the system wide view of transaction processing. Blocks representing key components of the algorithm and decision points are placed inside the rectangle representing

```
function COMMIT_TX(tx)
  if ∀ key ∈ tx.updates: key ∈ E.lCopies # all keys have local copies
    tx ← singleSiteCommit(tx) # commit locally, install updates
    if tx.status = COMMITTED
      remote async installTrans(tx) @ CORE # notify other sites, via core
  else
    tx ← remote coreCommit(tx) @ CORE # get a commit decision from the core
    if tx.status = COMMITTED
      installTrans(tx) # install transaction locally
  return tx.status

function ABORT_TX(tx)
  tx.status ← ABORTED
  tx.updates ← ∅ # forget deferred updates
```

Figure 4.4: COMMIT_TX and ABORT_TX at Originating Edge Site E

the site where these are executed. All arrow lines indicate the passage of time (but not to scale) and arrow lines that cross between the rectangle site boundaries indicate cross-site network

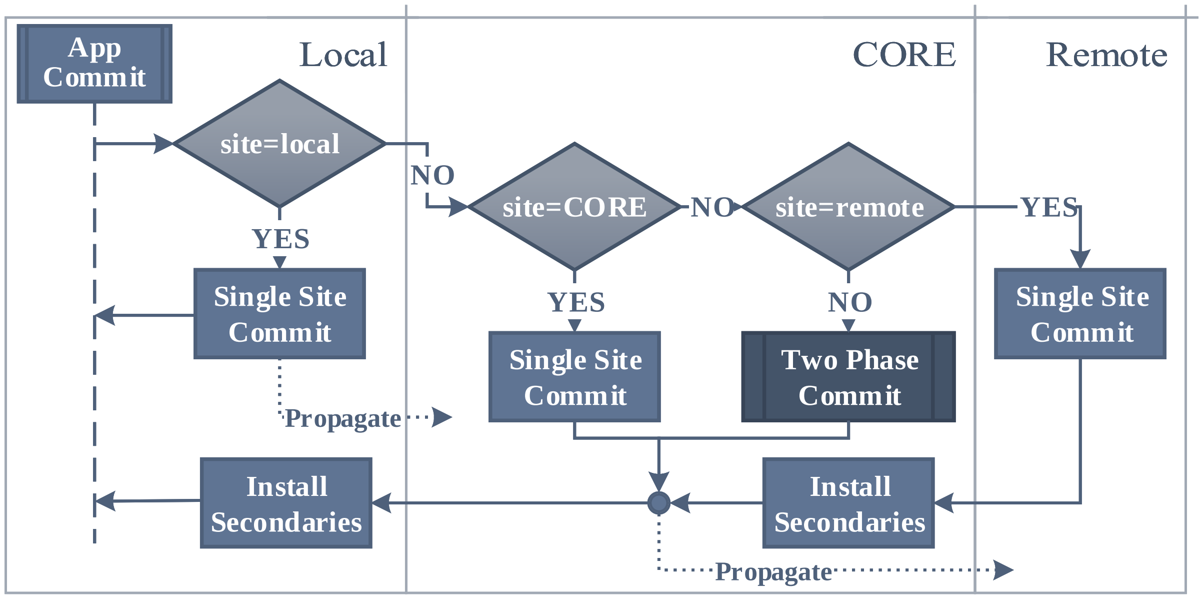


Figure 4.5: SpearDB Commit Decision

messages. The 2PC protocol (i.e. distributed commit) is responsible for further cross-site network messages, but its details are not shown in the figure. The dotted *Propagate* arrow lines represent asynchronous propagation. Finally, the transaction commit process begins and ends at the vertical dotted line representing the *App Commit*'s timeline.

There are four possible paths through the diagram Figure 4.5 - one for each of the 4 transaction commit techniques. Only one of these paths is followed for each individual transaction commit decision based on the answers to the rhombus shaped decision blocks in the diagram.

For a local commit, the edge site uses the `singleSiteCommit` procedure (Figure 4.6) to commit the transaction. This procedure enforces the PSI's prohibition against conflicting writes by checking for conflicts with concurrent transactions (lines 2-3) and aborting the committing transaction if a conflict is detected (line 15). Furthermore, ownership of the primary copy of all keys in the writeset needs to be checked again inside the critical section (line 4), in case a change of ownership of any primary copies has occurred concurrently. These type of ownership changes are discussed in Section 4.5. The procedure also enforces PSI's causality requirement by forcing a transaction that is attempting to commit too soon to wait until the transactions it causally depends on have committed (line 7). For local commits, the causality test is not needed and will never result in a wait. However, the `singleSiteCommit` procedure is also used for core and remote commits, for which the causality test is important.

```
1  function singleSiteCommit(tx)
2  | if ∀ key ∈ tx.updates: key not locked at S # check for locks (used by 2PC)
3  | and key unmodified since tx.startVTS  # enforce no concurrent updates
4  | and S.lCopies[key] = PRIMARY # recheck ownership of primary
5  |   tx.seqNo ← ++S.seqNo # assign commit version
6  |   # enforce causality:
7  |   wait until S.commitVTS ≥ tx.startVTS and S.commitVTS[S] = tx.seqNo - 1
8  |   WRITE(tx.updates, S, tx.seqNo) # store new versions of keys to database
9  |   # the WRITE operation is equivalent to appending ⟨val, S, tx.seqNo⟩
10 |   # to S.database[key], ∀ ⟨key, val⟩ ∈ tx.updates
11     FLUSH
12     S.commitVTS[S] ← tx.seqNo
13     tx.commitSite ← S
14     tx.status ← COMMITTED
15   else tx.status ← ABORTED
16   return tx
```

Figure 4.6: Commit Procedure for Single Site Commits at site `S`.
The vertical bar indicates a critical section.

```
function installTrans(tx)
  # enforce causality and respect commit site's commit order:
  wait until S.commitVTS ≥ tx.startVTS
         and S.commitVTS[tx.commitSite] = tx.seqNo - 1
  # extract the updates for keys that have a local copy at site S:
  localUpdates ← {⟨key, val⟩ ∈ tx.updates: key ∈ S.lCopies}
  WRITE(localUpdates, tx.commitSite, tx.seqNo) # store to site database
  # the WRITE is equivalent to appending ⟨val, tx.commitSite, tx.seqNo⟩
  # to S.database[key], ∀ ⟨key, val⟩ ∈ localUpdates
  S.commitVTS[tx.commitSite] ← tx.seqNo
  if S is the CORE  # core propagates updates to edges:
    remote async installTrans(tx) @ all edges except tx.commitSite
  else release any locks held by tx at S
```

Figure 4.7: Installation of Committed Transaction at site `S`

The critical section in Figure 4.6 needs to appear as if it is executed atomically. All vertical bars that appear in any of the pseudocode figures (Figure 4.6, Figure 4.9, Figure 4.12 and Figure 4.13) refer to the same critical region. Thus each site can have at most one thread executing inside this *one* critical region. Notice that the underlying database `WRITE` (line 8 in Figure 4.6) also needs to be inside the critical section so that all updates are installed in commit order. However the `FLUSH` to disk on line 11 is left outside the critical region in order to increase throughput, in case the underlying database is disk-based.

```
function coreCommit(tx)
  if ∀ key ∈ tx.updates: CORE.pCopy[key] = CORE
    tx ← singleSiteCommit(tx) # core commit
  else if ∃ E: ∀ key ∈ tx.updates: CORE.pCopy[key] = E
    tx ← remote singleSiteCommit(tx) @ E # remote commit
  else
    tx ← twoPhaseCommit(tx) # distributed commit
  if tx.status = COMMITTED
    remote async installTrans(tx) @ all sites
           except tx.startSite and tx.commitSite
  return tx
```

Figure 4.8: `coreCommit` Operation at `CORE`

```
# twoPhaseCommit runs at the core site
function twoPhaseCommit(tx)
  # find all sites that have primary copies of keys in tx.updates:
  pSites ← {S ∈ CORE.pCopies: (∃ key ∈ tx.updates: CORE.pCopies[key] = S)}
  remote async prepare(tx) @ all pSites
  wait for vote from all pSites
  if all sites vote YES
    tx.seqNo ← ++CORE.seqNo
    wait until CORE.commitVTS ≥ tx.startVTS
           and CORE.commitVTS[CORE] = tx.seqNo - 1
    WRITE(tx.updates, CORE, tx.seqNo) # store to CORE's database
    # the WRITE operation is equivalent to appending ⟨val, CORE, tx.seqNo⟩
    # to CORE.database[key], ∀ ⟨key, val⟩ ∈ tx.updates
    FLUSH
    CORE.commitVTS[CORE] ← tx.seqNo
    tx.commitSite ← CORE
    tx.status ← COMMITTED
  else
    tx.status ← ABORTED
    remote async release locks @ all pSites
  release locks held by tx at CORE
  return tx

# implementation of prepare at site S
function prepare(tx)
  # find all keys from tx.updates for which S has the primary copy:
  localkeys ← {key ∈ tx.updates: S.lCopies[key] = PRIMARY}
  if ∄ k ∈ localkeys that is locked at S
  and ∄ k ∈ localkeys modified since tx.startVTS
  and S.lCopies[k] = PRIMARY, ∀ k ∈ localkeys # recheck ownership of primary
    lock all keys in localkeys for tx
    return YES
  else return NO
```

Figure 4.9: Two-Phase Commit Coordination.
The vertical bar indicates a critical section.

Once a transaction has committed locally, its effects are propagated to the other sites in the system. Propagation is *asynchronous*, i.e., the edge site does not wait for propagation to complete before acknowledging a successful commit to the application. Edges propagate committed transactions by sending them to the core, which in turn propagates them to other remote edge sites. Propagation is handled by the `installTrans` function, which is shown in Figure 4.7.

When the transaction's start site is not able to commit the transaction locally, it forwards commit processing to the core site. The core determines where and how to commit the transaction, using the `coreCommit` function (Figure 4.8). This function determines whether core commit, remote commit, or distributed commit must be used to commit the transaction. Once the transaction has committed, it forwards the commit decision and commit version number to the originating edge, and initiates asynchronous propagation of the transaction to the remaining edges.

Asynchronous propagation in SpearDB is implemented using an additional optimization not shown in Figures 4.7 and 4.8. Since the core has an index of all secondary copies in the system (`CORE.sCopies`), only the part of a transaction's writeset that has a secondary copy at an edge needs to be sent to that edge. Other entries in `tx.updates` may be removed prior to sending. This allows for smaller network messages to be sent in order to limit bandwidth usage. The rest of the transaction's metadata, which includes the sequence number `tx.seqNo` and commit site `tx.commitSite`, still need to be sent to all sites in order to ensure the proper update of all sites' `commitVTS` vectors.

If distributed committal of the transaction is required, the core site will coordinate the commit, as shown in Figure 4.9. Each participating site confirms that it can commit its part of the transaction without a write-write conflict, and then locks the affected records until the core site can confirm that all sites are able to commit. Such locks cause concurrent conflicting transactions at the locking site to abort.

SpearDB's distributed commit implementation includes additional optimizations that are not shown in Figure 4.9: Since a single `NO` vote causes a transaction to abort, if the start site owns primary copies of any records updated by a transaction, it can invoke its prepare phase before requesting `coreCommit` at the core. If the start site's vote would be `NO`, the transaction can be aborted early and no network messages need to be sent. Furthermore, if the core also owns primary copies of updated records, it will execute it's prepare phase vote ahead of the other sites for the same reason. This particular optimization also halves the number of inter-site round trips for the 2PC case where the writeset is split between the originating edge and the core since the core needn't go back to the edge asking for a lock.

4.3 Transaction Shipping

In SpearDB, a transaction can GET any record, whether there is a local copy at the transaction's site or not. However, to handle GET requests for non-local records requires a synchronous message round-trip to the core site. If a transaction accesses multiple non-local records, multiple round-trips are required.

To address this problem, SpearDB provides a transactional stored procedure mechanism. Clients invoke SpearDB stored procedures via the CALL operation (Table 3.2). A SpearDB stored procedure is a general, parameterized method which may include SpearDB record operations. Stored procedure methods need to be created and replicated at all sites before they can be used. SpearDB executes each stored procedure invocation as a single transaction. The transaction's database snapshot includes all transactions committed at the site at which CALL is invoked as of the time of the invocation.

SpearDB is responsible for automatically starting a new transaction for each CALL invocation and it does so by calling BEGIN_TX at the local edge where CALL has been invoked. This ensures that regardless of the actual site where the stored procedure method is executed, the transaction will appear to have executed at the local edge and would have seen the database snapshot available to the local edge at the time of the CALL invocation.

By default, SpearDB executes stored procedures by shipping the CALL request to the core site, running the stored procedure at the core site, and shipping the results back to the originating edge. Because the core has a full copy of the database, the only two reasonable sites for executing the stored procedure code are the core and the originating edge. The latter is better if every tuple in

```
function CALL(name, params, readset)
  tx ← BEGIN_TX()
  if readset is not provided
  or (∃ r ∈ readset: S.lCopies[r] ≠ PRIMARY and S.lCopies[r] ≠ SECONDARY)
    return remote coreCall(tx, name, params) @ CORE
  else
    proc ← S.storeProc[name]
    return proc(tx, params)

function coreCall(tx, name, params)
  wait until CORE.commitVTS ≥ tx.startVTS # ensure updates have propagated
  proc ← CORE.storeProc[name]
  return proc(tx, params)
```

Figure 4.10: CALL at site S and callCore at CORE

the readset has a local copy (primary or secondary) and the transaction is able to commit locally as well. The core is better if at least one record read by the stored procedure is not present at the originating edge. Thus the more non-local reads in the transaction, the more appealing core execution becomes.

Notice that except for the case of a fully local writeset, the commit part of the transaction will always generate exactly one additional network hop in the case of local execution versus core execution; but the latter requires one network hop to begin core execution in the first place. Furthermore, executing the stored procedure at any other site in the system will require overall at least as many network hops as core or local execution.

Although core execution is SpearDB's default strategy for CALL requests, applications can influence the strategy by providing optional *readset hints* as part of the CALL invocation. If provided, a readset hint identifies the keys of records that may be read by a stored procedure invocation. If a readset hint is provided and SpearDB determines that all records identified in the hint are stored locally at the originating edge, then SpearDB will execute the stored procedure at the edge. The handling of CALL operations is summarized in Figure 4.10.

Notice that an empty readset is a valid hint. It indicates an update-only transaction. This is different from *not* providing a readset hint at all - which would indicate the programmer doesn't know or can't determine the readset at compile time. An empty readset hint triggers a local stored procedure execution, while lack of a hint results in the default core execution.

4.4 Discussion on the Effects of Failures

This thesis does not directly address the problem of failures. Furthermore, as a middleware, a system like SpearDB is dependent on the failure handling and recovery mechanisms of the underlying database, which may not implement any such contingencies. Instead, this section discusses the effects of entire site or network link failures.

The first scenario considered is the failure of an edge. While an edge is down, transactions sent to other edges in the system may continue to execute uninterrupted. Furthermore, they may also commit as long as their writesets don't contain tuples mastered at the failed edge. If a transaction tries to modify a tuple with the primary at the failed edge, the transaction would be made to abort at commit time. The core would notice the edge isn't reachable either when trying to acquire a lock for two phase commit or when trying to pass the transaction on to be single site committed at the failed edge. In the case of two phase commit, should an edge fail *after* informing the core of successfully acquiring all requested locks, the transaction may continue to commit at the core without problems.

Once a failed edge recovers, there are a few things to consider. If the failure was short-lived, a robust messaging implementation would deliver any outstanding messages. This would bring the edge back up to date and the system would resume normal operation. For longer-lived failures, and for the case of edge databases that can't be recovered (e.g. in-memory only storage option or a disk failure), the core may be forced to take *remedial action*. In order to keep the system online, any primary copies the failed edge held would be re-mastered at the core. Since the core holds a full copy of the database, this simply implies upgrading the appropriate secondary copies to primaries. The unfortunate consequence of this upgrade, however, is the possibility that transactions that have local committed at the failed edge and that have not yet propagated to the core would be forfeit. Finally, once the edge is back in operation, it would simply restart from an empty database and no locks held. Then it would rebuild its collection of primary and secondary copies over time, using data movement operations which will be described in Section 4.5.1. The amount of time for a failure to be considered long-lived is a system parameter.

Another scenario to consider is the failure of a network link between an edge and the core. In this case, the disconnected edge may continue working and would be able to commit local transactions and answer local reads. Non local reads or commits would cause transactions to abort. To the rest of the system it would appear as if the edge has failed and everything would transpire in the same manner as described in the failed edge scenario. A short-lived network partitioning event would again be resolved by a robust messaging implementation. A heartbeat mechanism would monitor the network link to the core and detect longer-lived link interruptions and force the edge to take itself offline until core communication is restored. Once the core link is back up, the edge would find out from the core if remedial action has taken away ownership of its primary copies. If it has, the edge would need to restart from an empty database as described before in order to preserve PSI guarantees in the rest of the system (at the cost of possibly loosing some locally committed transactions). If it hasn't, both the edge and the core may be brought back up to date by re-sending any unacknowledged messages.

Finally, a core failure is equivalent to a network link failure between the core and every edge in the system. As described, a heartbeat mechanism would force all edges offline until the core is back up and running. Since the core wasn't functional, no remedial action would have forced any primary copies to be re-mastered. Thus edges would be able to re-send unacknowledged messages and the system would continue to function. SpearDB isn't designed to survive a permanent core failure or unrecoverable core database since there is no guarantee that the all tuples have more than one copy.

4.5 Data Movement

The effectiveness of SpearDB's transaction processing is reliant on the location of the primary and secondary copies of all keys. As such, it is important that these locations are able to change when required. This section describes protocols that can modify the placement of individual primary and secondary copies within the network of sites. The protocols described in this section are designed to work in parallel with SpearDB's regular transaction processing without interrupting it. The main goal of their design is to minimize disruption of the transaction processing. In order to achieve this goal, data migration operations are limited to the following set:

- **Secondary replica creation:** A new secondary replica of a key is created at an edge that doesn't have any replicas of the key.
- **Secondary replica removal:** A secondary replica of a key at an edge is removed from the system. This operation can not be invoked at the core since the core must always maintain a full copy of the database.
- **Core to edge ownership transfer:** The primary replica of a key is transferred from the core (which must own the primary replica at the start of this operation) to an edge that already has a secondary replica of the same key. At the end of this operation, the core's replica will be downgraded to a secondary copy while the edge's will become the new primary.
- **Edge to core ownership transfer:** The reverse of the core to edge ownership transfer, this operation downgrades a primary copy of a key that exists at an edge to a secondary, while at the same time upgrading the core's secondary copy to the new primary.

All of these operations target a single key at a time. Each is simple enough to be implemented in a way that causes minimal disruption to the parallel task of transaction processing. At the same time, this set of four operations is comprehensive enough that any placement change of a key can be achieved by stringing together one or more of the operations in the set into a *movement chain*. For example if a primary key needs to move from edge α to edge β that doesn't have a secondary copy installed the following operations will occur: First, secondary replica creation at edge β and edge to core ownership transfer from edge α can occur in any order. Finally, after both previous operations complete, a core to edge ownership transfer to edge β completes the movement chain. Optionally, the secondary replica that is left at edge α as a result of the transfer may be removed if desired. For more complex movement operations such as transitioning from one database placement to another, where multiple keys need to change locations, each key's movement chain may be executed independently and in parallel.

In Section 4.5.1, each of these operations are discussed in more detail. Section 4.5.2 discusses the use of these operations in a dynamic setting where the transactional workload of SpearDB changes over time, changing the optimal location of keys for transaction processing with it.

4.5.1 Data Movement Operations

The four data movement operations in SpearDB are designed to be invoked only under certain conditions as explained before (e.g. a secondary copy of a key can only be created if there isn't

```
 1 function installSecondary(key)
 2   assert key ∉ E.lCopies # there can't already be a copy at E
 3   remote CORE.sCopies[key].add(E) @ CORE # have the CORE update its index
 4   E.lCopies[key] ← PARTIAL # reads don't see it, new versions are installed
 5   history ← remote READALL(key) @ CORE
 6   MERGE(key, history)
 7   E.lCopies[key] ← SECONDARY # now reads see it as well
 8
 9 function removeSecondary(key)
10   assert E.lCopies[key] = SECONDARY # E must have a secondary copy
11   E.lCopies[key] ← ∅ # remove key from local copies list
12   remote CORE.sCopies[key].remove(E) @ CORE # have the CORE update its index
13   wait until all currently running transactions at edge E finish
14   E.database[key] ← ∅ # remove key from edge's database
```

Figure 4.11: Secondary Replica Creation and Removal at Edge `E`

already a copy of it at the same edge). These restrictions are presented as `assert` statements in the pseudocode shown within this section. It is thus expected that an actor invoking these operations must first ensure these conditions are met. The core's key location indexes (`CORE.pCopy` and `CORE.sCopies`) are intended to be used for providing these assurances.

Furthermore, the operations are implemented to safely execute in parallel with the transaction processing operations presented in Sections 4.1, 4.2 and 4.3. However they are not intended to safely execute in parallel with each other if they operate on the same key. This restriction allows the data movement operations to be implemented with fewer locks, thus limiting the amount of disruption to transaction processing operations. The drawback of this restriction is the simple requirement that operations on each key must be serial. Operations on different keys, however, don't have this requirement.

Figure 4.11 presents the pseudocode for the secondary replica creation and removal operations. When adding a new secondary replica, the issue is to do so atomically while other transactions at different sites in the system may still be performing updates to the same key concurrently. To overcome this issue, the core's secondary copy index is first updated (line 3) and then the secondary replica is installed at edge `E` in a `PARTIAL` state without any versions in its history (line 4). While in a `PARTIAL` state, read operations (line 9 in Figure 4.3) don't see this replica. However, transactions installed at edge `E` as part of the propagation protocol are aware of the replica and update it as seen on line 6 of Figure 4.7. Furthermore, since the core's secondary copy index now shows edge `E` as having a secondary copy, the propagation protocol will include all updates of the `key`. This means that any concurrent updates of the `key` at other sites will be installed in the version history of edge `E` after this point. All versions of this `key` prior to this point are available at the core in the key's version history from where they are read by the READALL operation on line 5 of Figure 4.11. Then the MERGE operation on line 6 of Figure 4.11 prepends the history read from the core to edge `E` to any updates installed by the propagation protocol as described in Appendix A. Finally, the `key` can be made a full secondary replica on line 7, making it visible to read operations as well.

Removing a secondary copy is much simpler and only involves updating the edge and core indexes for tuple locations. Any read operation in progress that has passed the check of these indexes will still have access to the record history for the `key` which is not immediately deleted. Once all currently running transactions conclude, the record history is expunged from the edge database.

The pseudocode for the core to edge ownership transfer operation is presented in Figure 4.12. This operation needs to be started at the core with the `giveOwnershipToEdge` function and requires the edge that is to receive the primary copy to cooperate in the transfer of ownership. The first step of this operation is to relinquish the primary copy (line 6). This needs to be done while in the critical section and locked (line 5) so as to not interfere with any committing transactions. After the critical section in Figure 4.12 completes, any committing transactions would fail their ownership checks (line 4 in Figure 4.6 and line 30 in Figure 4.9) causing them to abort. At the end of the `giveOwnershipToEdge` function, the core's indexes already point towards the edge for the location of the primary copy of the `key`, although the edge still has a secondary copy. Any transactions trying to commit at the edge will also fail the ownership checks and abort. The core then calls the `takeOwnershipFromCore` function at the edge that is to upgrade its copy of the `key` to a primary in order to finish the process; providing it with the latest sequence number of the latest transaction to have been committed at the core. Since the primary copy of the `key` was at the core, this sequence number is guaranteed to include the latest update to the `key` being transferred regardless of which transaction commit protocol has been used to commit the latest transaction modifying this `key`. The edge can then wait until its own vector timestamp catches up to this sequence number in order to ensure the secondary copy of the `key` is updated to the latest version, before finally, the secondary copy of the `key` at the edge can take on the mantle of the primary.

The reverse process: handing ownership of a primary key at the edge to the core is very similar and the pseudocode is presented in Figure 4.13. Again the process is designed to be started at the core which will involve the relevant edge in the process at the point it is required. Similarly to the core-to-edge transfer, the primary copy downgrade needs to happen while locked and inside the critical section. While the upgrade of the secondary copy to a primary again needs to wait for the destination site's (this time the core) vector timestamp to catch up to the source edge's sequence number. Again this sequence number is guaranteed to include the latest update to the `key` being transferred regardless of which transaction commit protocol has been used to commit the latest transaction modifying this `key`. A single site commit would have updated the sequence number at the edge, while the core would already be aware of any completed two-phase commit.

```
function giveOwnershipToEdge(key, edge) # runs at the CORE
  assert CORE.lCopies[key] = PRIMARY # CORE must have the primary
  assert CORE.pCopy[key] = CORE # CORE must have the primary
  assert edge ∈ CORE.sCopies[key] # edge must have a secondary
| blocking lock key # if already held, wait for release before acquiring
| CORE.lCopies[key] ← SECONDARY # downgrade CORE primary copy to secondary
  CORE.pCopy[key] ← edge
  CORE.sCopies[key].remove(edge)
  release lock on key
  # at this point no site owns the primary copy (the CORE thinks edge does)
  remote call takeOwnershipFromCore(key, CORE.commitVTS[CORE]) @ edge

function takeOwnershipFromCore(key, seqno) # runs at edge E
  assert E.lCopies[key] = SECONDARY # E must have a secondary copy
  wait until E.commitVTS[CORE] ≥ seqno
  E.lCopies[key] ← PRIMARY
```

Figure 4.12: Core to Edge `E` Ownership Transfer. The vertical bar indicates a critical section.

```
function takeOwnershipFromEdge(key) # runs at the CORE
  assert CORE.lCopies[key] = SECONDARY # CORE must not have the primary
  assert CORE.pCopy[key] ≠ CORE # CORE must not have the primary
  edge ← CORE.pCopy[key]
  CORE.pCopy[key] ← NONE # no one owns the primary copy
  seqno ← remote call giveOwnershipToCore(key) @ edge
  wait until CORE.commitVTS[edge] ≥ seqno
  CORE.lCopies[key] ← PRIMARY # CORE has the primary now
  CORE.pCopy[key] ← CORE # CORE has the primary now
  CORE.sCopies[key].add(edge) # and the copy at edge has been downgraded

function giveOwnershipToCore(key) # runs at edge E
  assert E.lCopies[key] = PRIMARY # E must have the primary
| blocking lock key # if already held, wait for release before acquiring
| E.lCopies[key] ← SECONDARY # downgrade to a secondary copy
  release lock on key
  return E.commitVTS[E]
```

Figure 4.13: Edge `E` to Core Ownership Transfer. The vertical bar indicates a critical section.

4.5.2 Dynamic Data Movement Discussion

Chapter 6 and Chapter 7 discuss ways of finding optimal placements for all keys in a database given, among other things, a workload as input. However there are many applications where the workload isn't known apriori or where workloads change periodically based on the seasons, time of day, or other factors. For these types of scenarios a dynamic re-partitioning solution would improve transaction processing time.

A simple dynamic data partitioning approach would be to collect salient workload properties as the workload is being executed. Periodically, a data partitioning algorithm can be run using recently collected workload information. Then, data can be live migrated to transform the existing partition into the new one recommended by the algorithm.

SpearDB's core-edge architecture and transaction protocol ensure all transactions pass through the core at some point. This facilitates the data collection requirement to ascertain the properties of the current workload. The core additionally has knowledge of the locations of all tuple copies in order to be able to perform the proper routing of requests during transaction commit and propagation phases. Add to this the larger provisioning of computing resources at the core and it easily becomes the preferred location to perform the gathering of workload statistics and the periodic running of primary copy placement algorithm. The goal of this is to have an approximate view of what the workload looks like at the moment and hopefully for the near future. Exact precision isn't the goal, since such a reactive system would only provide an approximation of the future behavior of the workload.

The data movement operations discussed in Section 4.5.1 can then be used to perform the migration from the actual placement towards the desired placement. This may be done gradually and/or at times of low usage in order to minimize contention. It may also be done in the order of most to least accessed keys in order to increase gains early, as the migration to the desired placement doesn't need to fully complete before re-running the static partitioning algorithms on a fresh window of workload information. The desired placement will simply change and the migration towards it will become an ongoing process.

Chapter 5

SpearDB Prototype Evaluation

A SpearDB prototype has been built according to specifications described in Chapter 3 and Chapter 4. This chapter presents an evaluation of the prototype using several micro-benchmarks and a more realistic application workload.

SpearDB's various transaction commit strategies are compared in Section 5.2 in order to show their tolerance to network latency. The goal is to show the potential latency improvement of workloads if most transactions could be committed by latency tolerant strategies. A centralized database baseline is introduced in Section 5.3 as well as a *locality* workload metric. SpearDB's performance is then compared against this simpler baseline system for workloads of different localities in order to illustrate the kinds of workloads where SpearDB is particularly useful. A second baseline system is introduced in Section 5.4 based on the distributed Walter [88] database system. This baseline is again compared to SpearDB for different workloads in order to showcase the benefits of the modified PSI protocols and the addition of a central core site. Finally, in Section 5.5, SpearDB and the two baseline systems are evaluated using a more realistic application workload based on a key-value version of the RUBiS benchmark [84].

5.1 Experimental Setup

Experiments were performed on a cluster of servers, each with dual Intel Xeon E5-2620v2 6-core CPUs, 64 GB of DDR3 memory and an Intel S3700 200GB SATA3 SSD for persistent storage, running Ubuntu Server Linux, release 14.04.1 LTS. Connectivity is via 10Gbps Ethernet.

This cluster is used to simulate a core/edge setting by introducing a controllable amount of latency between servers using *tc*, a Linux network traffic control utility. Equal delays have been added on the ingress and egress paths to simulate the effect of geographical distance. When latency numbers are reported in the results and discussions, they are expressed as total round trip latency.

The SpearDB prototype is implemented in Java. The prototype uses Apache Thrift version 0.9.3 to provide the client API as well as for inter-server communications. This allows for the automatic generation of client libraries for different programming languages. Test clients were also implemented in Java. The underlying site databases for all of the experiments presented in this section use Berkeley DB Java Edition version 5.0.73.

SpearDB was configured to run with Thrift's TThreadPoolServer flavor. Clients connect to SpearDB through Thift RPC through an underlying TCP socket and using the CompactProtocol for marshaling.

Each experiment consists of an initial warm-up phase, during which performance is not measured, followed by the measurement phase. For experiments in which multiple clients connect to SpearDB concurrently, a final wind-down phase occurs after the measurement phase. During the wind-down phase, clients continue to perform operations at the same rate as they did during the measurement phase. This ensures that all clients experience a constant level of contention from other clients during their entire measurement phase. The experiment completes once all clients have entered the wind-down phase. All experiments were repeated 20 times and reported performance metrics are means over the 20 runs. Where space allows, 99% confidence intervals for these means are also included.

5.2 Micro-Benchmark: Commit Strategies

SpearDB implements four transaction commit strategies, and chooses one for each transaction depending on the locations of the primary copies of the records that the transaction updates. This experiment illustrates the performance of these four strategies, as a function of the latency between the edge sites and the core.

Figure 5.1 illustrates the core/edge scenario simulated for this experiment. Four SpearDB servers were used: one for the core and one for each of the three edge sites. Using *tc*, an L millisecond round trip latency has been added between two of the edges and the core, and $3L$ milliseconds between the core and the far edge. A single closed-loop SpearDB client ran on a fifth server, submitting one transaction at a time to the local edge, with no think time between requests. Each transaction consists of two PUT requests followed by COMMIT_TX.

Primary copies of different records are located at each of the four SpearDB sites. By choosing which two records a transaction updates, the commit strategy that SpearDB will use for the transaction can be controlled. This is summarized in Table 5.2. For each commit strategy, the mean client-observed transaction response time is measured.

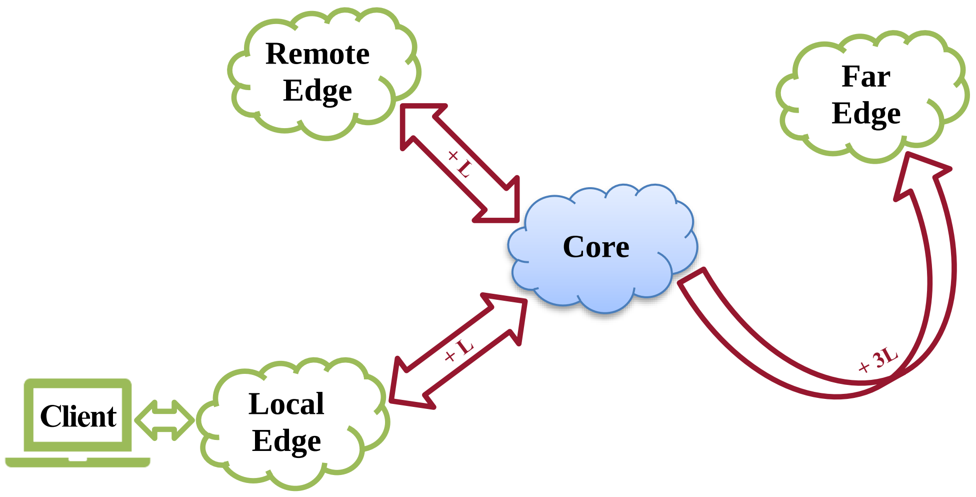


Figure 5.1: Commit Strategies Experiment Setup

Table 5.2: Commit Strategies Experiment: Data Placement

Record Placement	**Resulting Commit Strategy**	**Series Name**
both records at the local edge	local commit	local
both records at the core	core commit	core
both records at the remote edge	remote commit	remote
a record at the local edge and another at the core	2PC between the local edge and the core	2pc l-c
a record at the core and another at the remote edge	2PC between the core and a remote edge	2pc c-r
a record at the core and another at the far edge	2PC between the core and an edge that is far away	2pc r-f

Figure 5.3 shows the results of this experiment, and summarizes the important performance properties of SpearDB's commit protocol. When the transaction is geographically localized at the client's edge (the *local* series in Figure 5.3), it commits locally, and transaction response times are low and insensitive to network latencies. If the transaction's updates are limited to the local edge and the core (the *core* and *2pc l-c* series in Figure 5.3) SpearDB will commit the transaction using a single synchronous message round-trip between the local edge and the core, resulting in L milliseconds of added latency. In the worst case, when the transaction updates records mastered at one or more remote edges, the added latency due to the commit protocol is equal to that of a single round trip from the local edge to the furthest remote edge, via the core site.

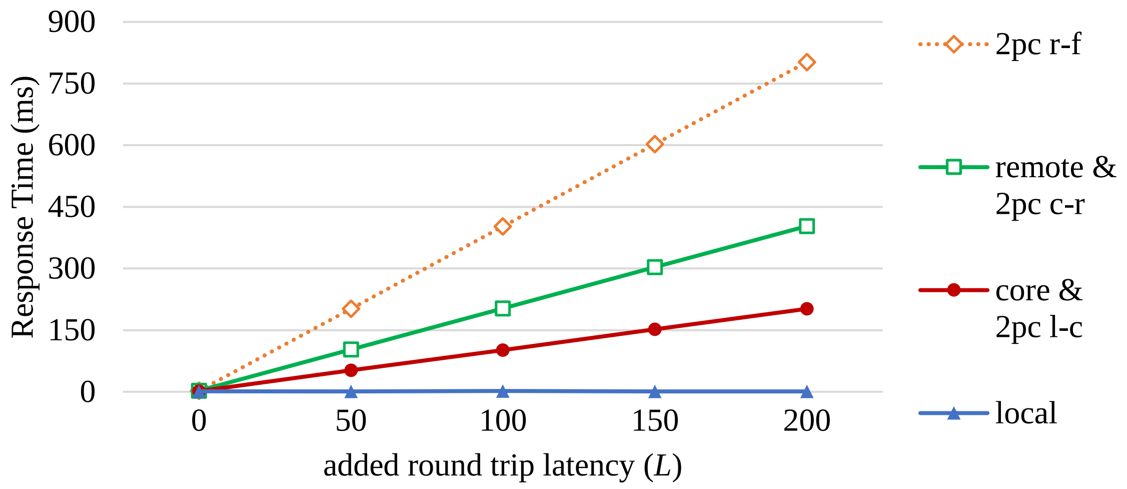


Figure 5.3: Effects of Commit Strategy and Network Latency (L) on Transaction Response Times

5.3 Micro-Benchmark: Locality

SpearDB is designed for workloads in which clients frequently access data located in their geographical region. To determine how a workload's data access pattern affects SpearDB's performance, a simple *locality* metric is introduced: It measures the ratio of transactions in a workload that are able to commit locally. SpearDB is then tested with workloads that have different locality values. In these experiments, the placement of records is fixed and the data access patterns of clients are varied in order to evaluate the entire spectrum of locality values.

The setup for these experiments consists of a core and 3 edges as shown on the left side of Figure 5.4. There is one client per edge and L ms of artificial round trip latency is injected between each edge and the core. The right side of Figure 5.4 describes an alternative setup that is referred to as the *centralized baseline*. The same 3 edges are connected to the core without any added latency. Instead, the same L ms of artificial latency is injected between the clients and the edges they connect to. This setup should exhibit the same performance as a centralized application and database that is accessed by remote clients, while enabling the use of the same data access and storage procedures as well as the same number of machines and resources for a fair comparison.

The clients are all run concurrently, but each client independently determines its own transactional workload and only issues one transaction at a time. Each transaction consists of the following sequence of operations: BEGIN_TX, PUT, PUT, COMMIT_TX. Transactions are implemented as stored procedures, invoked by a CALL from the client, so that each transaction requires

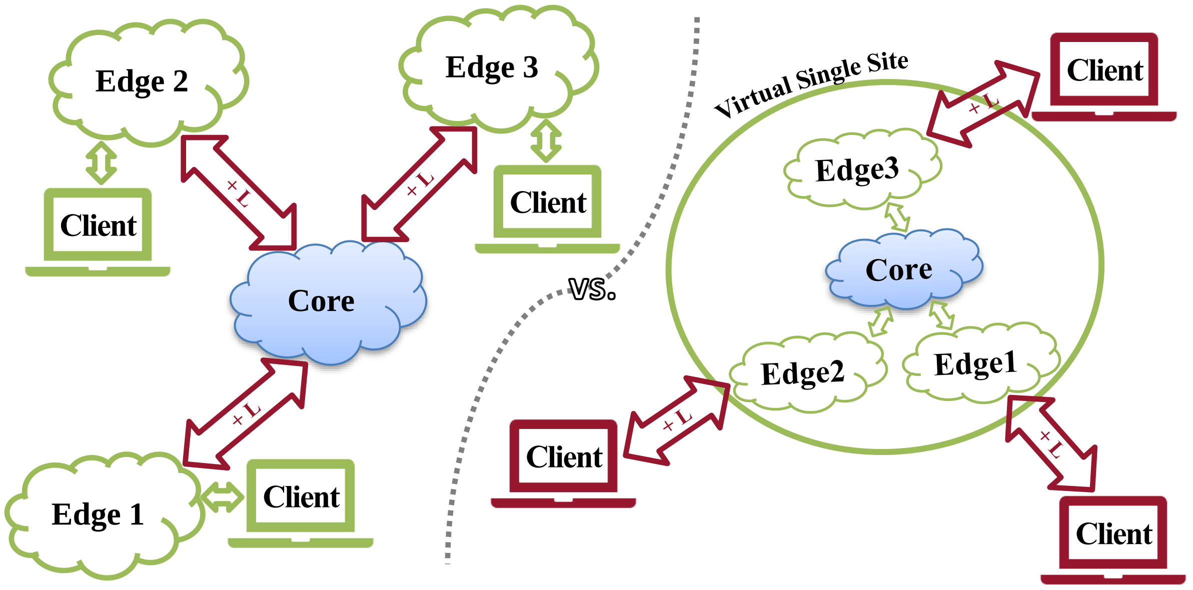


Figure 5.4: Locality Experiment Setup

a single round trip between the client and SpearDB. The empty readset of these transactions is provided as a readset hint as described in Section 4.3. As a result, SpearDB will execute the stored procedure at the the edge and perform all 4 API calls there.

The test database for this experiment includes 200 records, with 50 records mastered at each of the four sites. With probability P, a transaction will PUT two randomly selected records at the local edge. With probability $(1 - P)/2$, a transaction will PUT two randomly selected records at the core. Otherwise, the transaction will PUT two records selected randomly from among all 200 records in the system such that at most one record is at the local edge and at most one record is at the core. Thus, the expected locality of this workload will be P.

Figure 5.5 illustrates the average transaction response times for SpearDB and the centralized baseline, with $L = 10\,\text{ms}$ and locality (P) varying from from 0 to 1. Under SpearDB, the mean transaction response time depends on the locality, with stronger locality resulting in lower response times, as expected. The centralized baseline's performance is not sensitive to locality. Compared to the centralized baseline, SpearDB has a latency advantage for locality levels above 0.3. For workloads with low locality values, it can be guaranteed that SpearDB performs at least as well as a centralized solution by using a low-locality-optimized placement strategy that places most/all data items at the core.

Additional experiments have been performed for different latency values with L ranging from 10 ms to 100 ms. These experiments have all resulted in very similar results, with the main

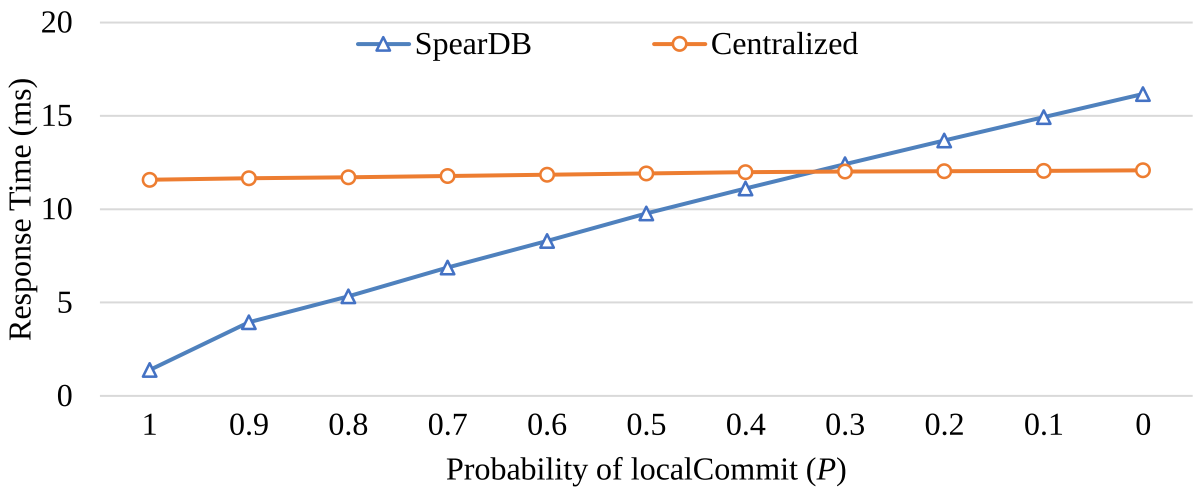


Figure 5.5: Goodput as a Function of Locality

difference being a linear scaling of the absolute latency values. Thus, as expected, when locality is high, SpearDB's advantage over the centralized baseline increases with L.

In addition to transaction response times, abort rates have also been measured. These are shown in Figure 5.6. Since the experimental setting includes only a single client per site, local transactions will never write-write conflict with one another, as there will be at most one local transaction at each site at any time. Thus, abort rates for both SpearDB and the centralized

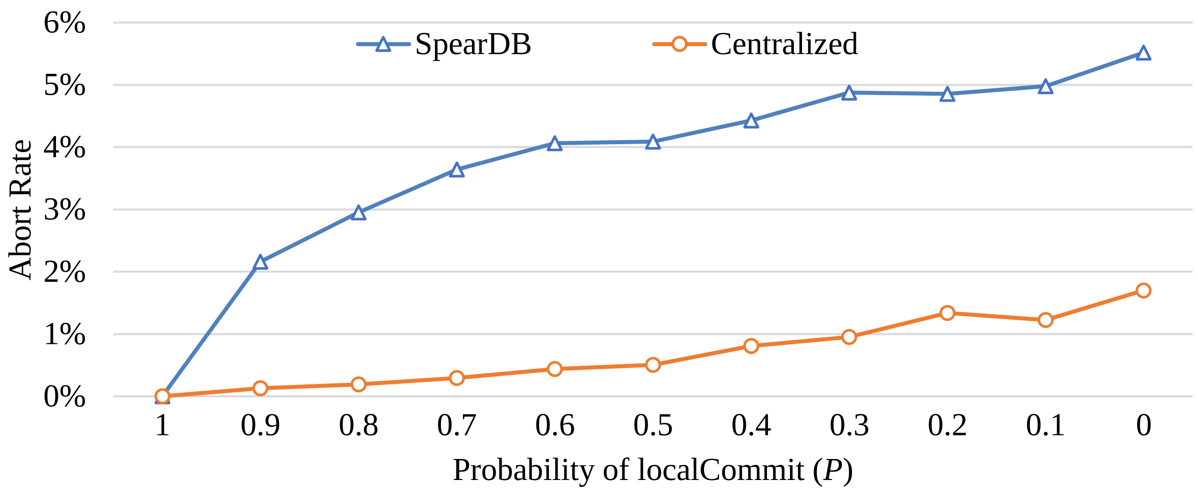


Figure 5.6: Transaction Abort Rate as a Function of Locality

baseline are zero when $P = 1$. However, non-local transactions can conflict with other transactions. Hence, the abort rate for both systems increases as locality (P) decreases. The transaction aborts observed in these experiments are due to local transactions conflicting with locks set by non-local, distributed transactions during commit processing (Figure 4.9). At edge sites, such locks are held for at least the time required for one message round trip between the edge and the core. So, abort rates for SpearDB are higher than those for the centralized baseline, since the edges are far from the core.

These elevated abort rates are a drawback of providing a strong consistency guarantee in a geographically distributed setting. Although PSI is able to avoid transaction synchronization (and hence aborts) in many cases, coordination is still necessary when transactions conflict or are causally related. Nonetheless, the increase in abort rates is small, despite the fact that transaction conflicts are likely because of the very small test database. Furthermore, the abort rate penalty is lowest for the high-locality workloads that SpearDB targets. Finally, since SpearDB yields significant latency savings compared to the centralized solution when locality is high, it is faster to simply retry an aborted local transaction at the edge than to perform all transactions at the core, as happens in the baseline.

5.4 Micro-Benchmark: Walter

In Section 5.3 SpearDB was compared to a centralized baseline. In this section, a similar benchmark is presented that compares SpearDB to a geo-distributed baseline based on Walter [88]. Walter was designed to be deployed across geographically distributed sites, like SpearDB. However, in the original Walter design, all sites are peers - there is no designated core site, like SpearDB's. Walter attempts to execute transactions locally, at the site at which the transaction arrives. If it is unable to do so because the required data is not mastered locally, the originating site coordinates a cross-site distributed protocol to commit the transaction.

To create a baseline for SpearDB, Walter has been adapted for the asymmetric core/edge setting. This baseline will be referred to as *core-aware Walter*. Core-aware Walter provides the same global consistency (PSI) as both SpearDB and the original Walter. Like Walter, core-aware Walter executes transactions at the originating edge when possible. When an edge site is unable to process a transaction locally, that edge coordinates Walter's two-phase commit protocol with other sites involved in the transaction. However, all communication between the originating edge and other sites is routed through the core. The originating edge sends outgoing messages to the core, which then distributes them to other sites as necessary. Similarly, return messages are collected at the core before being returned to the originating edge. In contrast, distributed commits in SpearDB are coordinated by the core site, rather than the originating edge.

This experiment uses the server configuration shown in the left side of Figure 5.4, with three edge servers, one core server, and a single client sending requests to each edge. $L = 10\,\text{ms}$ of

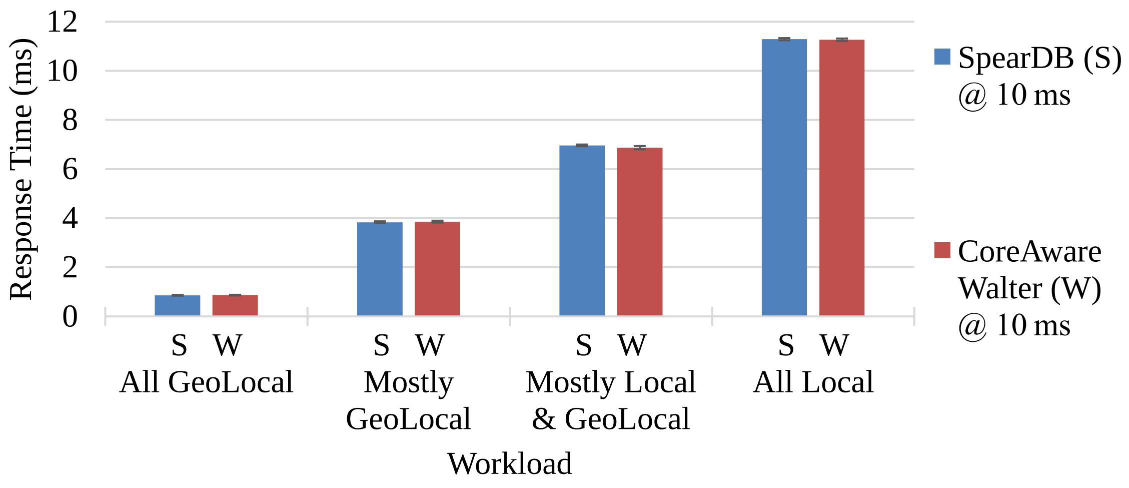


Figure 5.7: SpearDB and Walter Response Times

artificial message round trip latency has been injected between the edges and the core.

The same database from the locality micro-benchmark (Section 5.3) is reused. Workload transactions each perform two PUT operations. Clients generate four types of transactions - corresponding to the four types of commit processing performed by SpearDB, by controlling which records are updated by the transaction. Several different workloads have been tested; each representing a different mix of transaction types. For each workload, transaction response times and abort rates have been measured. For these simple update-only transactions, SpearDB and core-aware Walter had very similar transaction response times as seen in Figure 5.7. This is to be expected given that all workloads used in this experiment exhibit the same number of cross-site network messages in both SpearDB and the baseline. Hence, the focus of this experiment is on abort rates, which are shown in Figure 5.8. Abort rates differ, because SpearDB's more complex commit protocols allow it to acquire fewer locks and hold on to them for shorter periods of time.

Each pair of bars in Figures 5.7 and 5.8 corresponds to a different workload. For the *All GeoLocal* workload, 100% of transactions are able to commit locally, at the originating edge. Since both SpearDB and Walter can commit locally without any network communication and since there's a single client per edge, there is no contention and there are no aborted transactions.

The *Mostly GeoLocal* workload, consists of 85% local transactions, 5% of transactions that update a pair of records at a remote edge, and 10% of transactions that update one record at the originating edge and one at the core. This represents a workload with strong geographic locality, but with a few transactions originating at an unexpected edge (i.e. a user accessing an account while traveling), and a small number of un-localizable objects, placed at the core. Both SpearDB and core-aware Walter experience aborts under this workload, but SpearDB's rate is about half

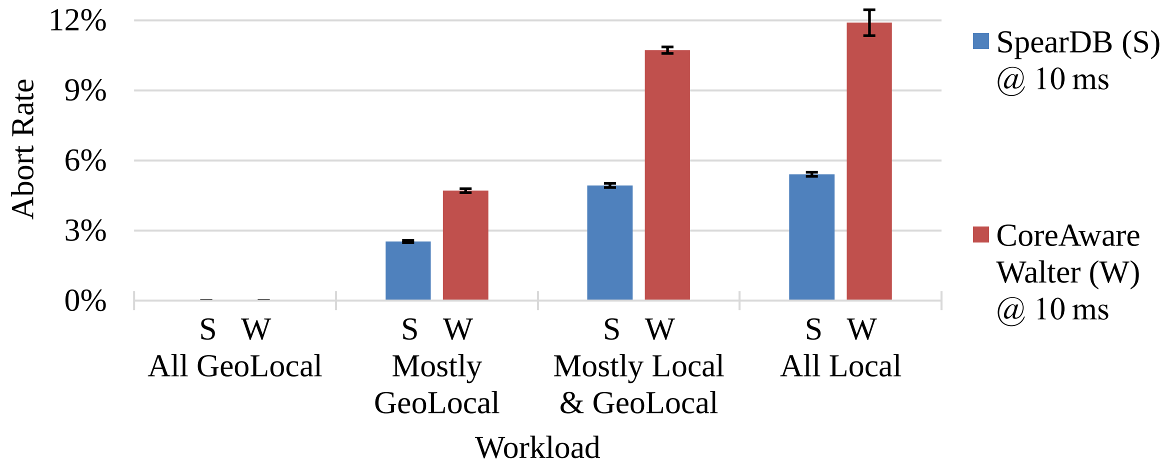


Figure 5.8: SpearDB and Walter Abort Rates

of the baseline's. There are two reasons for this: First, SpearDB uses a remote single site commit for the remote-edge transactions, while the baseline uses a distributed commit coordinated by the local edge. This requires setting locks at the remote edge, which can cause local transactions at the remote edge to abort. For the core/edge transactions, both SpearDB and the baseline use a distributed commit, but the baseline holds its locks for longer than SpearDB, increasing the risk of aborts. In both systems, such transactions start their commit procedure by acquiring a lock on the record at the local edge. If this succeeds, the core is contacted and it acquires a lock on the core record. However, in SpearDB the core will coordinate the transaction, committing it and releasing its core lock immediately, while Walter needs to keep the core lock until the local edge (acting as the coordinator) commits the transaction.

The *Mostly Local and GeoLocal* workload includes 60% local edge transactions, 5% remote edge transactions, 30% transactions that update two records at the core, and 5% core/edge transactions. Abort rates are higher for both systems here, because there are fewer local transactions. However, SpearDB's rate is much lower than the baseline's because it is able to perform single-site commits for all but the core/edge transactions.

Finally, the *All Local* workload considers a mix of 40% local, 30% core, and 30% remote edge transactions. For this workload, all commits in SpearDB are single site, whereas the baseline requires distributed commit for more than half of the transactions. SpearDB still experiences some transaction aborts because of write-write conflicts between transactions that originate at different edges, but its rate is much lower than the baseline's rate as there are no aborts due to locking.

5.5 Evaluation Using RUBiS

This section presents an evaluation of SpearDB using a more realistic workload based on the RUBiS benchmark [84]. SpearDB is compared to both the centralized baseline presented in section 5.3 and to core-aware Walter, which was described in section 5.4.

5.5.1 RUBiS

RUBiS is a simulated auction application that includes a database describing users, items, and bids. The web application implements operations that allow users to browse, search, view, buy, place bids and comments on items, register new items and users, and view user accounts. RUBiS includes a session model that generates a workload consisting of sequences of these operations.

Small changes to RUBiS were required in order to adapt it for use as a SpearDB benchmark: First, since RUBiS had been written for relational databases, the database schema needed to be adapted. In addition, all database interactions were re-implemented to use SpearDB's transactional key-value interface. Table 5.9 summarizes the new SpearDB schema created for RUBiS. Different record types are distinguished by prepending a unique string denoting the record type to each record's key.

Second, the notion of locality has been introduced to the application. RUBiS had already modeled "regions", and each user is associated with a particular region. Locality has been introduced by associating each RUBiS region with a SpearDB edge site. RUBiS operations are submitted at the submitting user's region's edge site, and items registered by a user are associated with the user's region. An additional *locality* parameter has also been introduced to the RUBiS session model. Whenever a region needs to be selected during a user's session, the user's own region is selected with probability *locality*, otherwise some other region is selected at random. For example, when users search for items within a region, they will search within their own regions with probability *locality*; otherwise they search in a remote region.

Table 5.9: SpearDB Schema for RUBiS

Record Type	**Key**	**Primary Copy**	**Secondary Copy**
category list	single record	core	all edges
region list	single record	core	all edges
user	nickname	user's region's edge	core
item (incl. bids)	item id	item's seller's region's edge	core
item list	region+category	region's edge	core

5.5.2 Experiment Design

A total of 11 SpearDB servers, representing 10 edge sites and 1 core site, have been used in these experiments. The goal has been to model a geo-distributed system at a continental scale based on the geography of the United States. Major population centers spanning the country have been chosen to represent the servers. Kansas City, KS - a relatively central city in the list has been chosen to be the core site. As in previous experiments, *tc* has been used to introduce synthetic round trip latency between edge servers and the core, as shown in Table 5.10. These latencies have been obtained from WonderNetwork's[1] global ping statistics. All three systems (SpearDB and the two baselines) have been tested using the round trip latencies shown in the "Latency" column of Table 5.10. Additionally, the core-aware Walter baseline has been tested using the scaled down latencies shown in the "Scaled Latency" column. These have been obtained by scaling the first set of latencies by a factor of approximately 0.66. The scaling factor has been computed by dividing the average edge-to-edge latency by the average edge-to-core-to-edge latency for all possible combinations of 2 (different) edges. The scaled latency experiments are intended to approximate how Walter would perform if it were allowed direct edge-to-edge communication, rather than forcing all such communication through the core.

Table 5.10: RUBiS Experiment Network Round Trip Latencies

City	**Latency**	**Scaled Latency**
Atlanta, GA	30 ms	20 ms
Austin, TX	18 ms	12 ms
Chicago, IL	15 ms	10 ms
Detroit, MI	22 ms	14 ms
Kansas City, KS	0 (CORE)	0 (CORE)
New York, NY	31 ms	20 ms
Orlando, FL	40 ms	27 ms
San Francisco, CA	42 ms	28 ms
Seattle, WA	53 ms	35 ms
St. Louis, MO	8 ms	5 ms
Washington, DC	33 ms	22 ms

A RUBiS database consisting of 25,000 users and 50,000 items has been used, with data placed as described in Table 5.9. Ten RUBiS clients ran concurrently at each edge (100 clients in total). Each client generated requests according to the RUBiS bidding workload (which includes both read-only and update transactions), with 1 second think time between requests. Each RUBiS operation has been implemented as a SpearDB stored procedure. For the two baselines,

[1]https://wondernetwork.com/

the procedure executes at the site at which the request arrives. For SpearDB, the readset hint mechanism is used to control whether the procedure executes at the arrival edge or at the core. To ensure each system (SpearDB and the two baselines) was tested with the same workloads, the random seeds used by the clients for workload generation have been fixed.

In each experiment, the response time for each RUBiS request has been measured. These measurements do not include the time required for HTML page generation - which is database-agnostic. This configuration is such that both SpearDB and the two baseline systems are lightly loaded, so that latencies due to contention for server resources are negligible. Aborted transaction counts have also been recorded. However, abort rates were insignificant for both SpearDB and the two baselines because data contention is very low for this workload: the hottest (i.e. most accessed) tuple in the workload is only accessed by 0.8% of transactions.

5.5.3 Results

Figure 5.11 shows the mean request response time for each system for each of the 17 types of RUBiS operations that use the database. For these experiments, the RUBiS workload *locality* parameter was set to 80%. The percentage below each operation indicates its frequency of occurrence in the workload. In the figure, the operations have been grouped into four categories, according to the database access characteristics of the operations. The last graph in Figure 5.11 shows the frequency-weighted overall average request response time for the entire workload.

Both SpearDB and core-aware Walter are able to handle operations in the "Read Globally Replicated Index" category entirely at the submitting edge, thus avoiding the edge-to-core latency that affects the centralized baseline. These are simple operations that essentially consist of a single GET request. Response times for both SpearDB and core-aware Walter are less than 0.5 ms.

Operations in the "Read Single User And/Or Item Information" category are also read-only, and involve obtaining information about one user or one item (or both) from the database. If the required user or item is local, then both SpearDB and core-aware Walter can handle the request at the submitting edge. This happens close to 80% of the time, because of the workload locality parameter setting. For non-local requests, SpearDB's read-hinting mechanism will cause it to execute the stored procedure at the core, where (by design) the required user or item data will be found. This results in a single edge-to-core round trip for non-local requests. Core-aware Walter handles non-local reads by directing them to the *master copy* of the required user or item record, which will be at a remote edge. Hence, core-aware Walter experiences higher latencies for non-local requests. This penalty is higher for "Put Comment" because it involves two GET requests, each of which results in an edge-to-edge round trip.

Operations in the "Update Transactions" category perform both GET and PUT operations. The "Register User" and "Register Item" pages are responsible for creating new users and items respectively. This involves creating a new record and inserting it in the database. For both

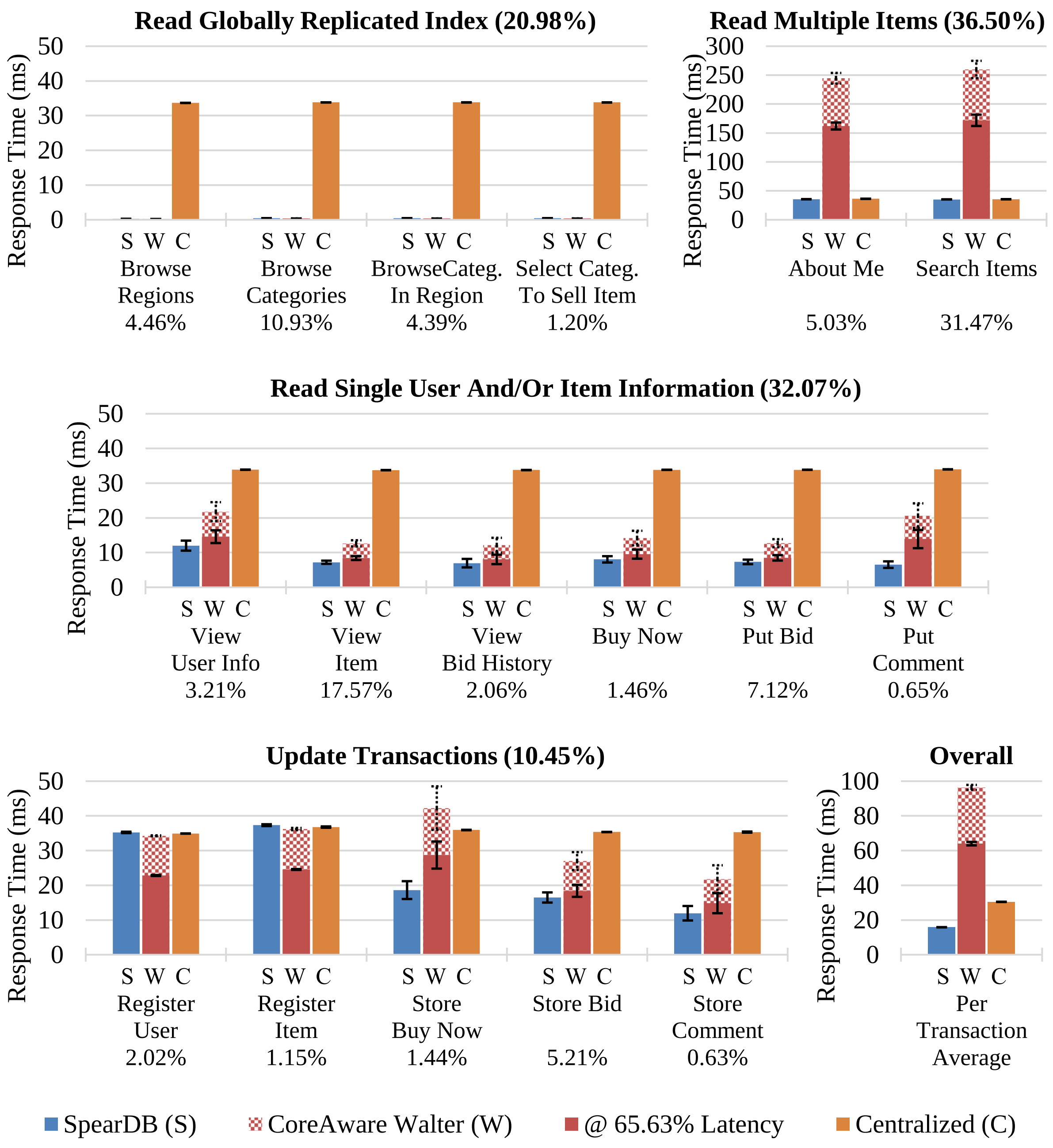


Figure 5.11: RUBiS Database Response Time by Page and Page Type @ 80% Locality

SpearDB and Walter, this results in a non-local commit requiring a message exchange between the edge and core. Thus, both of these systems, as well as the centralized baseline, have mean response times that depend on edge-to-core message latency, which ranges from 8 ms to 53 ms in this experiment, depending on the edge (see Table 5.10).

The "Store Comment" and "Store Bid" pages add a new comment or bid, respectively. This is done by reading, modifying and writing back records for a user and (in the case of a bid) an item. If these records are local, both SpearDB and Walter will perform the entire transaction at the edge. If they are remote, SpearDB's readset hints will move execution of the entire transaction to the core and, at commit time, a remote commit will occur at the edge of the user/seller receiving the comment/bid. This amounts to one round trip to the core and another between the core and a remote edge. Core-aware Walter, on the other hand, will execute the transaction at the local edge, will need to go out to a remote edge for the read(s), and will perform a two phase commit with the same remote edge(s). This results in either 2 or 3 times more network round trips than SpearDB.

The "Store Buy Now" operation allows a user to buy an item. It is similar to the previous two in that it reads, modifies and writes back records for an item and a user. Additionally, if the entire stock quantity of the item is depleted as a result of this purchase, the item needs to be removed from the item list index. SpearDB uses a readset hint to determine whether to perform the operation's stored procedure at the local edge (if the user and item are local) or ship it to the core. In contrast, core-aware Walter always executes the operation at the local edge, and requires multiple round trips to a remote edge to read user and item information when they are not local.

The final group in Figure 5.11 is made up of pages that require reading multiple items. For core-aware Walter, each non-local GET results in a message round-trip to the site that holds the primary copy of the requested record, resulting in high latencies. For SpearDB, the clients cannot provide read hints for these operations since their readsets are not known in advance. SpearDB's execution strategy in this case is to ship the stored procedure to the core, where all reads are from the core's database. Thus, both SpearDB and the centralized baseline execute these operations with a single edge-to-core round trip. This execution strategy is enabled by SpearDB's asymmetric design, which insists on a core site with a complete copy of the database.

All of the results in Figure 5.11 were obtained with workload locality set at 80%. The RUBiS experiment has also been performed with other locality settings, ranging from 50% to 95%. Figure 5.12 shows the frequency-weighted mean operation response time for each system, as a function of *locality*. As expected, the performance of both SpearDB and core-aware Walter improves with increasing locality. For this workload, locality must be very high before core-aware Walter outperforms the centralized baseline. SpearDB performs better than the centralized baseline for all of the locality values tested. The most important factors in SpearDB's performance relative to core-aware Walter are (a) its ability to handle non-local reads at the core, rather than at remote edges, and (b) its ability to ship CALL operations for execution at the core.

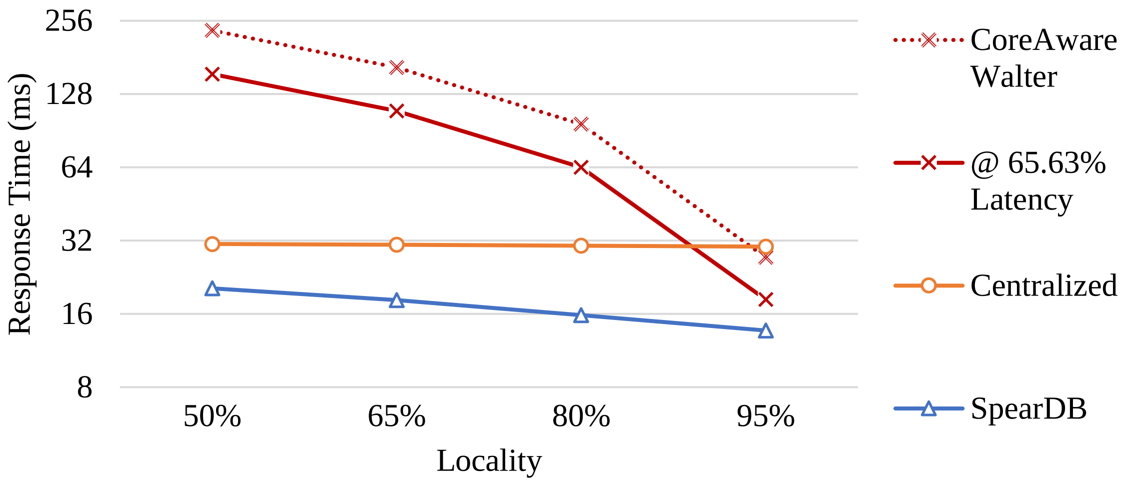


Figure 5.12: RUBiS Database Average Response Time by Locality

Chapter 6

Data Partitioning - Primary Copy Placement

Transactions in SpearDB can be viewed as having two critical phases: *execution* and *commit*; followed by a non-critical *propagation* phase. During the execution phase, both primary and secondary copies of tuples are used to answer GET requests. In the commit phase, only the primary copies are involved in the protocol. The propagation phase happens in a lazy manner after (and if) the transaction has been committed in the system and does not contribute to client-perceived latency. For this reason, the focus of data partitioning will be solely for the benefit of the execution and commit phases.

Both the commit and execution protocols in SpearDB are designed to minimize the number of cross-site messages, which are the main source of latency in SpearDB's transaction processing. These protocols work with any valid placement of primary and secondary copies (i.e. assignment of primary/secondary copies to particular edges or the core). However different placements affect transaction execution times. The goal of this chapter is to present algorithmic approaches to data placement that try to minimize the overall execution time of a given transactional workload.

The commit phase of transactions is only concerned with the location of primary copies, while secondary copies play no role. The execution phase, on the other hand, treats all copies (primary and secondary) the same - in other words, for the execution phase primary copies also play the role of secondary copies. This introduces a dependency between the placement of secondary copies and primary copies. For this reason, and also for the sake of simplifying this complex problem, data partitioning is split into two separate partitioning problems in this thesis: primary and secondary copy placement. Both are approached as static off-line optimization problems, where the entire workload is known apriori. The remainder of this chapter is dedicated to solutions for primary copy placement. Secondary copy placement is discussed in Chapter 7.

6.1 Primary Copy Placement

The problem of static primary copy placement in SpearDB can be expressed as follows: given a workload trace and a network topology, determine a placement of primary copies for which the latency of network messages generated by the commit phases of all transactions (the *cost*) is minimized. The goal is to choose a location (either the core or one specific edge site) for the primary copy of each tuple in the database.

More formally, let D represent the database (which is a set of tuples) and E represent the set of edge sites. Along with D and E, a workload trace W and network topology N make up the input to the primary copy placement problem. The workload trace W consists of a set of transactions. Each transaction $tx \in W$ is described by a triple $\langle s, w, ws \rangle$, where the start edge site $s \in E$ indicates the edge at which the transaction request arrives, the weight w represents the number of times the transaction occurs in the workload, and the writeset $ws \subseteq D$ indicates which tuples from the database the transaction updates. A dot notation is used to refer to the elements of a given transaction, e.g. $tx.ws$ refers to the writeset of a transaction tx. The network topology N defines the network latencies between the core and each edge site in E. These latencies are expressed in terms of round trip time. The notation $l(e)$ is used to represent the network latency between the core and edge $e \in E$. Table 6.1 summarizes all the components of the input to the primary copy placement problem.

The output of primary copy placement is a mapping p that maps each tuple in the database to either an edge or the core ($p : D \to E \cup \{\text{core}\}$), indicating where each tuple's primary copy should be located.

Finally the goal of the primary copy placement algorithm is to choose the output function p that minimizes the objective function $\text{cost}(W, N, p)$ defined in equation (6.1).

Table 6.1: Primary Copy Placement Input Components

Component	**Description**
D	the database - a set of tuples
E	the set of edge sites
N	the network, where $l(e)$ denotes the round trip network latency from edge e to the core
W	a workload consisting of transactions
$\forall tx \in W$	a transaction $tx = \langle s, w, ws \rangle : \begin{cases} tx.s \in E & \text{start site} \\ tx.w & \text{weight} \\ tx.ws \subseteq D & \text{writeset} \end{cases}$

$$\text{cost}(W, N, p) = \sum_{tx \in W} \left[tx.w \cdot \text{cost}(tx, N, p) \right] \tag{6.1}$$

The objective function as defined in equation (6.1) is generic and consists of the sum of all individual transaction costs multiplied by the transaction weights. The individual transaction cost, however, is system specific. It models the total latency of all network messages generated by the commit phase of transaction tx given the network topology N and the data placement defined by the primary copy placement output function p. For the SpearDB database, the transaction cost function is defined in equation (6.2).

$$\text{cost}(tx, N, p) = \begin{cases} 0 & \text{if } tx.ws = \varnothing \text{ or } \forall t \in tx.ws, p(t) = tx.s \\ l(tx.s) & \text{if } \begin{cases} \forall t \in tx.ws, p(t) \in \{tx.s, \text{core}\} \\ \text{and } \exists t_c \in tx.ws : p(t_c) = \text{core} \end{cases} \\ l(tx.s) + \max\limits_{\substack{e \in E - \{tx.s\}: \\ (\exists t \in tx.ws : p(t) = e)}} l(e) & \text{if } \exists t \in tx.ws : p(t) \notin \{tx.s, \text{core}\} \end{cases} \tag{6.2}$$

The complexity of the SpearDB transaction cost function as defined in equation (6.2) stems from the multiple ways of committing transactions in SpearDB. For local commit or an empty writeset, the cost is 0 since there are no network messages generated. If all tuples in the writeset are either local or at the core and at least one tuple in the writeset is at the core then the cost is the latency between the local edge and the core since the transaction can be committed with a single network message to the core. This case includes both SpearDB's core commit strategy and a two-phase commit between the local edge and the core only. Finally, if at least one tuple in the writeset is at a remote site (i.e. not local and not at the core), then the cost function reflects the latency between the local edge and the core plus the largest latency between any remote sites with primary copies in the writeset and the core. This case includes both SpearDB's remote commit strategy (for which there will only be one remote site with primary copies in the writeset) and the general two-phase commit case. The largest latency is used since in SpearDB's two-phase commit all remote edges are messaged concurrently.

Table 6.2: Example Primary Copy Placement Input and Corresponding Output

Workload (W)			**Network (N)**	
Start Site	**Weight**	**Writeset**	**Edge**	**Latency**
Edge1	40	TupleA	Edge1	10 ms
Edge2	40	TupleB	Edge2	10 ms
Edge3	60	TupleA, TupleB	Edge3	10 ms

Placement (p)	
Tuple	**Site**
TupleA	Edge1
TupleB	Edge2

Table 6.2 presents an example primary copy placement problem and its solution. The input network consists of 3 edges, each a 10 ms round trip time from the CORE. The input workload consists of 3 transactions. The first occurs 40 times in the workload, starts at Edge1 and updates TupleA. The second occurs 40 times, starts at Edge2 and touches TupleB. The third occurs 60 times, starts at Edge3 and touches both TupleA and TupleB.

The algorithm's task is to assign each of the two tuples to any of the 4 possible sites (the core or one of the 3 edges). This needs to be done in such a way that minimizes the sum of all network message latencies generated by the commit phase of the 3 transactions. The full search space for this example is 16 possible placements. The optimal placement (obtained by exhaustively searching this small search space) is presented in Table 6.2 and consists of placing TupleA at Edge1 and TupleB at Edge2.

For this optimal placement (p), the 80 occurrences (total) of the first two transactions can all commit locally without generating any network messages:

$$\text{cost}(tx_1, N, p) = 0$$
$$\text{cost}(tx_2, N, p) = 0$$

The 60 occurrences of the last transaction (starting at Edge3) all commit via 2PC: Each of these occurrences require a round trip from Edge3 to the Core (10 ms) and two *concurrent* round trips from the Core to Edge1 and Edge2 respectively (10 ms each). The total latency of each 2PC commit here is then:

$$\text{cost}(tx_3, N, p) = 10\,\text{ms} + \max(10\,\text{ms}, 10\,\text{ms}) = 20\,\text{ms}$$

Thus, the total cost of the optimal placement is:

$$\text{cost}(W, N, p) = 40 \cdot \text{cost}(tx_1, N, p) + 40 \cdot \text{cost}(tx_2, N, p) + 60 \cdot \text{cost}(tx_3, N, p)$$
$$\text{cost}(W, N, p) = 40 \cdot 0 + 40 \cdot 0 + 60 \cdot 20\,\text{ms} = 1200\,\text{ms}$$

Several algorithms that can produce approximate solutions are discussed in Section 6.2, Section 6.3, Section 6.4 and Section 6.5. An experimental evaluation of the algorithms proposed is presented in Section 6.6 followed by some comments and observations for primary copy placement in Section 6.7.

6.2 A Simple Affinity Approach to Primary Copy Placement

Because transaction writesets in a workload can partially overlap, choosing to improve the commit phase of one transaction could cause slower commit phases for others. This section presents a simple baseline algorithm that ignores transactional tuple groupings and only looks at the access site(s) and frequency of each tuple individually. Ignoring transactional groupings substantially simplifies the problem of primary copy placement.

In this simple affinity approach, the primary copy for a tuple is placed at the edge which accesses it most frequently for writes *if* the *edge write ratio* is greater than a given `THRESHOLD` parameter. The edge write ratio for a tuple t and edge e is defined in equation (6.4) and can have values between 0 and 1 (inclusive). The edge write ratio represents the ratio of writes at an edge to total writes at all edges for a given tuple. Where "writes" is defined in equation (6.3) as the number of individual write requests for a tuple t that originate at a site s.

$$\text{writes}(t, s) = \sum_{tx \in W : t \in tx.ws \text{ and } tx.s = s} tx.w \tag{6.3}$$

$$\text{edgeWriteRatio}(t, e) = \begin{cases} 0 & \text{if writes}(t, s) = 0, \forall s \in E \\ \frac{\text{writes}(t,e)}{\sum_{s \in E} \text{writes}(t,s)} & \text{otherwise} \end{cases} \tag{6.4}$$

Algorithm Affinity for solving primary copy placement problem
Input: database `D`, workload `W`, the set of all edges `E` and an edge write ratio `THRESHOLD`
Output: primary copy placement `p`

```
for each tuple t in database D # initialize tCounts and output o
  p[t] ← CORE # default output is core placement
  tCounts[t][CORE] ← 0
  for each edge e in edge set E
    tCounts[t][e] ← 0
for each transaction tx in workload W # count site accesses for all tuples
  for each tuple t in writeset tx.ws
    tCounts[t][tx.s]+= tx.w

for each tuple t in database D # main placement loop
  max ← maximum value of tCounts[t][s] ∀ s ∈ E ∪ {CORE}
  maxSite ← site s for which tCounts[t][s] = max
  total ← sum of tCounts[t][s] ∀ s ∈ E ∪ {CORE}
  if total > 0 and max / total > THRESHOLD then p[t] ← maxSite
return p
```

Figure 6.3: Simple Affinity Primary Placement Algorithm

Figure 6.3 describes the simple affinity placement algorithm in pseudocode. The given `THRESHOLD` parameter acts as the minimum edge write ratio that needs to be reached in order to place a primary copy at an edge. If this `THRESHOLD` isn't reached, the primary copy is placed at the core as the tuple is accessed enough times at other edges to make a more central placement worthwhile (i.e. the writes originating at the write-dominant edge for the tuple don't outweigh all other writes originating elsewhere).

In a simplified network topology, where all edges are equidistant from the core, choosing a value for `THRESHOLD` can be done based on the transaction cost function defined in equation (6.2). Assuming each edge is N ms from the core, local commit would be free, core commit would cost N ms, and the other commit strategies would cost $2N$ ms. Thus, moving a tuple from the core to edge e would save a cost of N ms for all writes originating at edge e but would cost an additional N ms for all other writes. So in an equidistant edge network topology a value of `THRESHOLD` $= 0.5$ is ideal. For different network topologies, making `THRESHOLD` a parameter provides additional flexibility to either favor more or less aggressive edging of tuples.

This is very similar to the way non-transactional systems such as PNUTS [17] handle data placement. The advantage to this method is its simplicity and scalability: a single pass over the workload is required and one vector counter for every tuple (the vector counts tuple accesses from each site). The drawback is that a lot of relevant input information, such as transactional groupings and network latencies between the edges and the core, is ignored. This can lead to poor placement decisions in some workloads.

For the example input described in Table 6.2, simple affinity would conclude that both TupleA and TupleB are accessed 60 times from Edge3 and only 40 times from another edge (Edge1 for TupleA and Edge2 for TupleB). So for any value of `THRESHOLD` < 0.6 the algorithm would place both tuples at Edge3. This placement is not optimal and has a cost of $2 \cdot 40 \cdot 20\,\text{ms} = 1600\,\text{ms}$ compared to the optimal cost of 1200 ms. For values of `THRESHOLD` ≥ 0.6, simple affinity would place both tuples at the core. This is another sub-optimal placement which has a cost of $(40 + 40 + 60) \cdot 10\,\text{ms} = 1400\,\text{ms}$.

6.3 A Greedy Approach to Primary Copy Placement

This section describes a greedy algorithm for solving primary copy placement. It relies on a simple refinement procedure that starts from an existing placement and either "*edges*" or "*cores*" a single transaction. The edging of a transaction consists of (re)locating its entire writeset (each tuple in the writeset) to the transaction start site. Similarly, "coring" relocates each tuple in a transaction's writeset to the core. The intuition behind this refinement procedure is that it brings one transaction towards one of its two lowest cost methods of committing. Edging is particularly effective on tuples that are mostly accessed from one site and coring is very effective on hot groups of tuples that are often accessed together but from multiple edges by multiple transactions.

Algorithm Greedy for solving primary copy placement problem

Input: database D, workload W, network N, the set of all edges E, and an initial placement pInit

Output: primary copy placement p

```
pMin ← pInit # start from the initial placement given
repeat # refinement loop
  p ← pMin
  for each transaction tx in workload W # inner loop
    e ← p modified so that ∀ t ∈ tx.ws, e[t] ← tx.s  # edged tx placement
    if cost(W, N, e) < cost(W, N, pMin) then pMin ← e
    c ← p modified so that ∀ t ∈ tx.ws, c[t] ← CORE  # cored tx placement
    if cost(W, N, c) < cost(W, N, pMin) then pMin ← c
until pMin = p # didn't find any better placement at this stage
return p
```

Figure 6.4: Greedy Primary Placement Algorithm

Thus the greedy algorithm, which is described in Figure 6.4, starts off with an existing data placement that is given as an additional parameter. This may be randomly generated, or everything could be placed at the core, or it could be the actual current placement of data in a running DBMS, or even the placement result generated by a different algorithm such as simple affinity presented in Section 6.2. The greedy algorithm consists of several iterations, each of which considers every possible refinement (i.e. the edging as well as coring of every transaction in the workload) and chooses the one (or one of the ones) which results in the placement with the lowest cost. At the end of the iteration, the chosen placement replaces the initial placement given and another iteration begins. This is repeated until no more refinement is possible (i.e. every possible refinement results in a placement with a cost that's higher or the same as the placement before refinement).

One nice property of this greedy algorithm is that every iteration produces a new valid placement with an improved (i.e. lower) cost. This allows the algorithm to be stopped early if runtime is a limiting factor and the solution computed up to the stopping point is guaranteed to be better than the starting placement. It is also the case that a solution obtained by different means may be improved upon by running this greedy algorithm with said solution (placement) as input. And should the greedy algorithm be unable to improve upon a given placement, a single iteration is required to reach this conclusion.

The worst case complexity of each iteration step, as presented in the pseudocode in Figure 6.4, is $O(n^2)$, where n is the size of the workload. This assumes the cost function, which is the same cost function defined in equation (6.1), requires another pass through the entire workload to

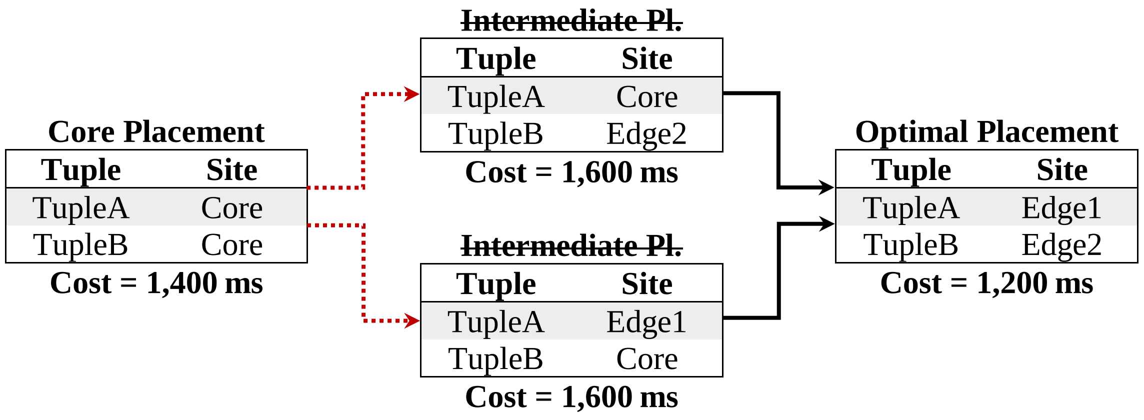


Figure 6.5: Example of Greedy Primary Copy Placement Stuck in Local Minimum

compute the cost of each individual transaction. An optimized version of the same algorithm has been implemented for experiments and is presented in Appendix C. In the optimized version, there are some initialization steps required and the worst case complexity is still $O(n^2)$. However, in practice, the algorithm runs faster by reducing the time required to compute $\text{cost}(W, N, p)$ after the first iteration.

As a greedy algorithm, there is no guarantee of an optimal result. In the example presented before in Table 6.2, this greedy algorithm gets stuck in the local minimum of a placement that has both TupleA and TupleB at the core. This core placement isn't optimal as it has a cost of $(60 + 40 + 40) \cdot 10\,\text{ms} = 1400\,\text{ms}$ compared to the optimal cost of 1200 ms. And in order to progress towards the optimal placement, both the transaction starting at Edge1 and the transaction starting at Edge2 need to be edged. However the greedy algorithm can only edge one transaction per step. Unfortunately, edging either one of these transactions without edging the other produces a placement of higher cost. Thus, these search paths are never followed to the next step that, as shown in Figure 6.5, would have produced the global optimum.

6.4 Primary Copy Placement as Hypergraph Partitioning

Primary copy placement is inherently a partitioning problem: tuples need to be assigned to partitions (i.e. sites) with the goal of minimizing a cost that is determined by their partition assignments. As such, existing partitioning techniques may be used to solve the primary placement problem. One particular partitioning problem that primary copy placement can be expressed as is hypergraph partitioning. The key insight here is that hyperedge weights can be used to model transaction commit costs in such a way that running a min-cut partitioning algorithm on the hypergraph corresponds to finding the minimum cost placement.

A hypergraph is simply a generalization of a graph in which edges can join any number of vertices as opposed to just two. Formally a hypergraph is a pair consisting of two sets: a set of vertices and a set of hyperedges; where each hyperedge is a non-empty subset of the set of vertices. Similar to weighted graphs, a weighted hypergraph is a hypergraph where all hyperedges have a numeric weight associated with them. In this document, the term *edge* is used to refer to a computational site at the edge of the cloud. In order to avoid confusion, the same term is never used as shorthand for "hyperedge" within this document.

The problem of min-cut hypergraph partitioning is a known NP-hard problem [47, 41] which consists of dividing the vertices of a hypergraph into disjoint partitions by removing some of the hyperedges. For the version of min-cut hypergraph partitioning used in this document, the number of partitions is given as input and some of the vertices are pre-assigned to specific partitions in which they are required to stay. The goal is to perform the partitioning that requires the cheapest total hyperedge cut (i.e. the sum of the weights of the hyperedges removed is minimized).

Thus, in order to solve primary copy placement via hypergraph partitioning the following 3-step procedure is required: First, the primary copy placement problem needs to be expressed as a hypergraph. This procedure and the reasoning behind the hyperedge weights used in building the hypergraph are explained in Section 6.4.1. Second, a min-cut hypergraph partitioning algorithm needs to be run on the hypergraph obtained in the first step. Finally, the third step involves converting the results of hypergraph partitioning into a solution to the primary copy placement problem. The last two steps are described in detail in Section 6.4.2.

6.4.1 Building the Hypergraph

This section describes the process of converting a primary copy placement problem into a hypergraph. First, every site in the system (all edges in E and the core) and every tuple in the database D are represented as vertices. Then, for every transaction in the workload ($tx \in W$), several hyperedges as constructed as follows: An *inner hyperedge* connects all the vertices representing tuples in the transaction's writeset ($\forall t \in tx.ws$). Then, for every site in the system, an additional hyperedge is constructed connecting the site's vertex with all vertices representing tuples in the transaction's writeset. The hyperedge that contains the core vertex will be referred to as the *core hyperedge*, the one containing the local site's vertex (i.e. vertex corresponding to tx.s) will be referred to as the *local hyperedge*, and all remaining hyperedges will be referred to as *remote hyperedges*. Figure 6.6 illustrates the construction of such a hypergraph for a single transaction with two tuples in its writeset. One remote hyperedge is drawn in Figure 6.6 for illustrative purposes. It has an index i intended to represent that there may be more than one

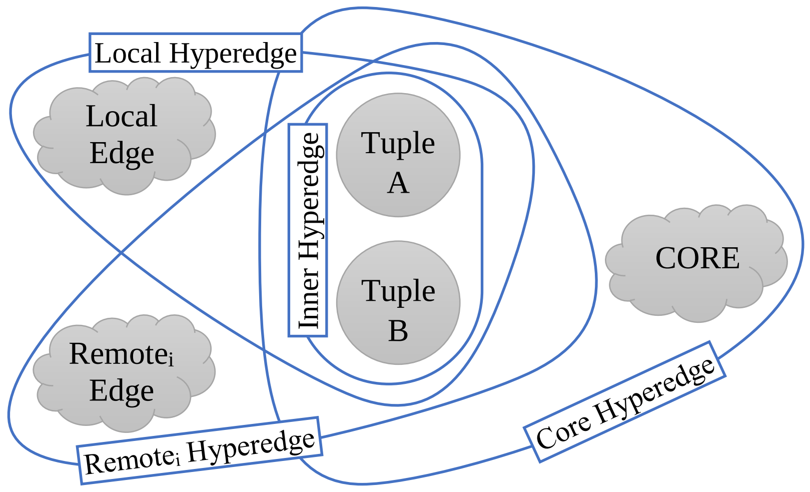


Figure 6.6: A Two-Tuple Transaction Expressed as a Hypergraph for Primary Copy Placement

such hyperedge required, one for each remote edge. Note that for transactions with a single tuple in the writeset, an inner hyperedge doesn't exist because hyperedges need to connect at least two vertices. For the same reason read-only transactions can be skipped completely when building this hypergraph.

The presence of a hyperedge in this hypergraph indicates that all tuple nodes inside it should be placed at the same site; and this site should be that corresponding to the edge or core vertex inside the hyperedge if such a node is present. Thus the hyperedges in Figure 6.6 describe possible co-location of the tuples in the transaction's writeset. By pre-assigning every site vertex in the hypergraph to a distinct partition, running min-cut hypergraph partitioning forces the algorithm to choose where a transaction writeset is located. Any outcome can have at most one of the hyperedges that include site vertices per transaction writeset. This follows from the fact that all site vertices have been pre-assigned to different partitions.

Since the local, core and remote hyperedges all contain a site vertex, at most one of them may survive the cut. Furthermore, because the inner hyperedge's set of tuples is a subset of all other hyperedges' sets of tuples for the same transaction, if any other hyperedge is present, it also means that the inner hyperedge must be present as well; and conversely if the inner hyperedge is cut, all other hyperedges pertaining to the same transaction are also cut. Thus, 4 possible outcomes are distinguished for different combinations of hyperedges being present or cut for one transaction. These are described in the first column of Table 6.7.

Each of the 4 outcomes result in a different placement (second column of Table 6.7), and hence a different transaction execution cost (third column of Table 6.7). Thus, the key idea is to

Table 6.7: Outcome of All Possible Hyperedge Combinations for a Single Transaction

Hyperedge Present	**Placement Outcome** (p)	**Cost** $\mathbf{cost}(tx, N, p)$
Local + Inner	$p(t) = tx.s, \forall t \in tx.ws$	0
Core + Inner	$p(t) = \text{core}, \forall t \in tx.ws$	$l(tx.s)$
Remote$_i$ + Inner	$p(t) = \text{Remote}_i, \forall t \in tx.ws$	$l(tx.s) + l(\text{Remote}_i)$
(none)	(no placement outcome) can only conclude that: $\exists t_1, t_2 \in tx.ws : p(t_1) \neq p(t_2)$	$\approx l(tx.s) + \frac{\sum_{i=1}^{\lvert E\rvert-1} l(\text{Remote}_i)}{\lvert E\rvert-1}$

set the sum of all missing (cut) hyperedges in each of the 4 cases to the transaction execution cost that stems from the tuple placement resulting from all the present edges in the case. For each transaction, this results in a system of $|E|+2$ equations in $|E|+2$ variables. These variables are the hyperedge weights, which can thus be obtained by solving this system of equations.

For the first 3 combinations of present hyperedges described in Table 6.7, the placement outcome p is straightforward and the transaction execution cost is easily determined from equation (6.2). In the last case, when all hyperedges belonging to the transaction have been cut, there is no clear placement outcome, which makes the transaction execution cost indeterminate. An approximation needs to be used instead.

If all hyperedges have been cut, the implication is that not all tuples in the transaction's writeset are located at the same site. The actual location where these tuples end up being is unknown at this stage in the algorithm. Other transactions that have overlapping writesets will determine their location after the min-cut partitioning algorithm is run. This can either mean $\text{cost}(tx, N, p) = l(tx.s)$ if part of the writeset ends up being at the local edge and the rest at the core; or $\text{cost}(tx, N, p) = l(tx.s) + \max l(\text{Remote}_i), \forall i \in \overline{1, |E|-1}, \exists t \in tx.ws : p(t) = \text{Remote}_i$ otherwise (i.e. if at least one tuple in the writeset ends up being at a remote edge). In this case, an approximate cost is used, which is obtained by ignoring the less expensive former cost function outcome and using only the more expensive latter cost. Furthermore since the set of remote edges where all tuples in the writeset end up being placed (Remote$_i$) is also unknown, the maximum network latency of these edges is approximated to be the average remote edge latency for the entire network: $\left[\sum_{i=1}^{|E|-1} l(\text{Remote}_i)\right] / \left(|E|-1\right)$. Because these costs are approximations of the true cost of executing the transaction, the placement determined by the hypergraph partitioning may not be optimal.

Having obtained a cost for all 4 outcomes (last column of Table 6.7), the system of equations that needs to be solved to determine the hyperedge weights can now be defined. Let L, C and I represent the hyperedge weights of the local, core and inner hyperedges respectively and let $R_i, i \in \overline{1, |E|-1}$ represent the hyperedge weights of all $|E|-1$ remote edges. Equation (6.5) is then the resulting system.

$$\begin{cases} C + \sum_{i=1}^{|E|-1} R_i = 0 \\ L + \sum_{i=1}^{|E|-1} R_i = l(tx.s) \\ L + C - R_j + \sum_{i=1}^{|E|-1} R_i = l(tx.s) + l(\text{Remote}_j), \forall j \in \overline{1, |E|-1} \\ L + C + I + \sum_{i=1}^{|E|-1} R_i = l(tx.s) + \frac{\sum_{i=1}^{|E|-1} l(\text{Remote}_i)}{|E|-1} \end{cases} \tag{6.5}$$

Solving the system in equation (6.5) provides the hyperedge costs described in equation (6.6).

$$\begin{cases} L = l(tx.s) + \frac{\sum_{i=1}^{|E|-1} l(\text{Remote}_i)}{|E|} \\ C = \frac{\sum_{i=1}^{|E|-1} l(\text{Remote}_i)}{|E|} \\ R_j = \frac{\sum_{i=1}^{|E|-1} l(\text{Remote}_i)}{|E|} - l(\text{Remote}_j), \forall j \in \overline{1, |E|-1} \\ I = \frac{\sum_{i=1}^{|E|-1} l(\text{Remote}_i)}{|E| \cdot (|E|-1)} \end{cases} \tag{6.6}$$

In the cost function for the entire workload (defined in equation (6.1)), the transaction costs are multiplied by the transaction weight ($tx.w$). This fact has been so far omitted in order to simplify the explanation of hyperedge weights. Thus, the hyperedge weight values obtained for L, C, R_x and I need to be multiplied by the transaction weight ($tx.w$) in order to obtain their correct values. This needs to be done for every transaction in the workload. At this point the hypergraph is fully constructed from the input of primary copy placement.

As an example, the hypergraph constructed from the input workload described in Table 6.2 is shown in Figure 6.8. Vertices in this figure are represented as circles if they correspond to tuples and cloud shapes if they correspond to sites. Their corresponding entity is clearly labeled. Hyperedges in Figure 6.8 are represented either as closed curves surrounding the vertices they contain; or, in the case of two-vertex hyperedges, a simple line connecting the two vertices is used for representational clarity. Hyperedge costs are displayed in a rectangle on the hyperedge perimeter. Finally, based on the transaction used to generate a hyperedge, one of three line styles and colors are used as described in the legend underneath the hypergraph.

6.4.2 Hypergraph Partitioning and Retrieving the Placement

Once the workload is defined as a hypergraph and weights are assigned to edges, a min-cut hypergraph partitioning algorithm can be run. All the hypergraph vertices that represent sites need to be pre-assigned to distinct partitions. This ensures sites are never co-located in the same partition. Furthermore, the number of output partitions needs to be set as the number of sites ($|E| + 1$). This ensures that every partition contains exactly one site vertex and that every tuple

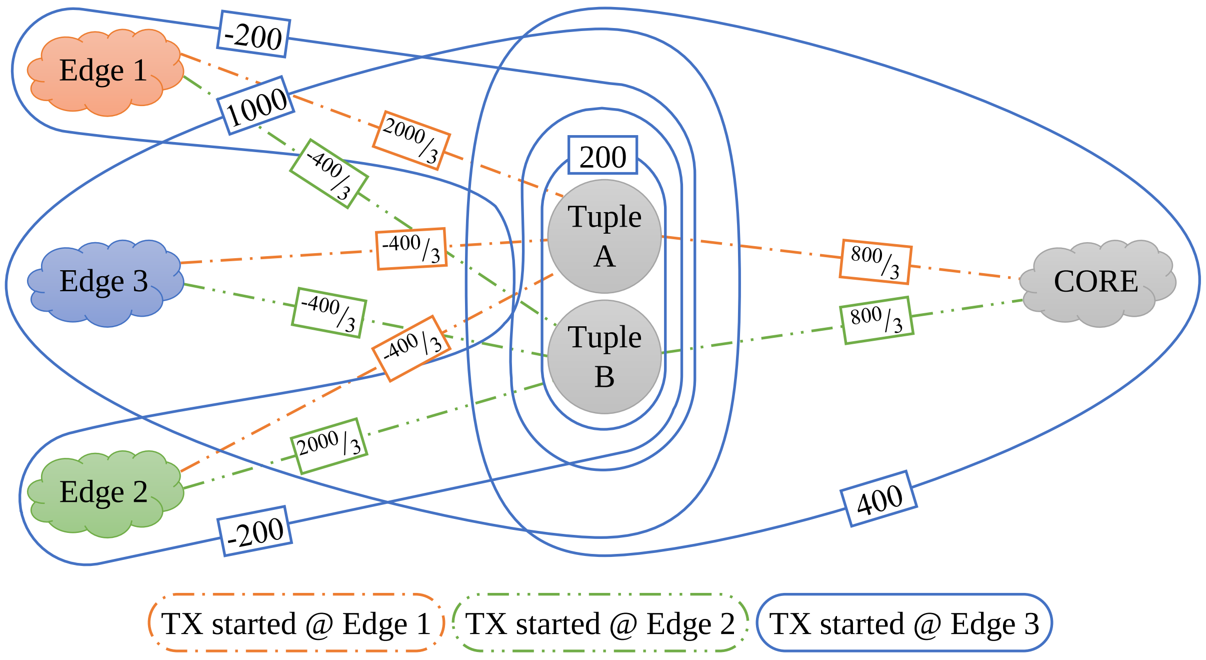


Figure 6.8: Hypergraph for the Example Workload and Network Defined in Table 6.2

vertex, which the algorithm *must* place in on of the $|E| + 1$ partitions, ends up co-located with one such site vertex.

An off-the-shelf solver such as hMETIS [42, 41] can then perform the actual partitioning. The final result of partitioning is the list of partitions, each of which will have exactly one site vertex and 0 or more tuple nodes. The site node in every partition can then determine the location of all the tuple nodes in the same partition, if there are any.

Internally, min-cut hypergraph partitioning is meant to achieve this result by removing (cutting) the minimum total weight of hyperedges required to achieve a valid partitioning. For example, in the hypergraph presented in Figure 6.8: the optimal way to achieve a valid partitioning is to cut every single hyperedge except the two that weigh $\frac{2000}{3}$ ms as shown on the left side of Figure 6.9. The resulting (minimum possible) total cost of the cut in this example is then 1200 ms. Since a hyperedge connecting Tuple A to Edge 1 is left uncut this means Tuple A will appear in the same partition as Edge 1 thus determining its position. For similar reasons Tuple B will be placed at Edge 2. And as explained in Table 6.2, this particular placement is indeed the optimal one for this example.

Unfortunately, as other authors have discovered as well [19], hypergraph partitioning tools aren't widely studied and don't seem to be very mature: Even for the 6 nodes and 13 hyperedges

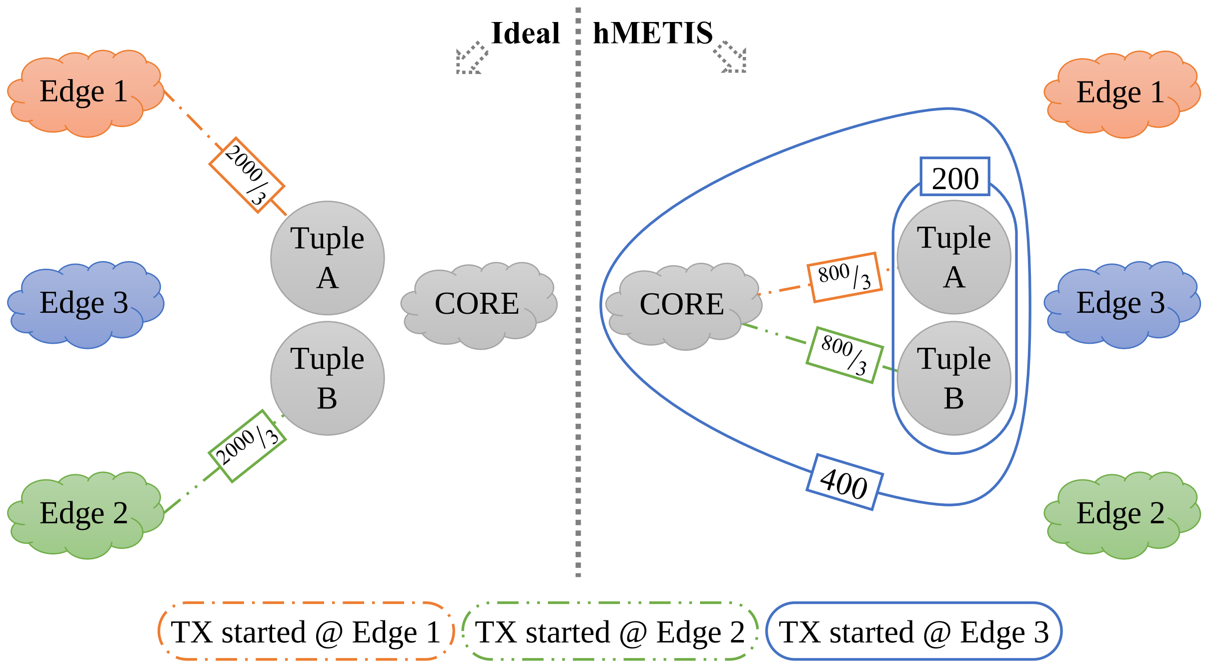


Figure 6.9: Remaining Hyperedges After Min-Cut for Hypergraph Defined in Figure 6.8

in Figure 6.8, hMETIS is unable to find the optimal min-cut with any combination of the available algorithm tuning parameters. The result provided places both Tuple A and Tuple B in the same partition as the core. Such a placement implies all hyperedges that contain an edge vertex are cut. The only hyperedges left are the internal hyperedge and the 3 core hyperedges, as shown on the right side of Figure 6.9. The total cost of such a cut is 1400 ms which is larger than the minimal cut. These poor results can also be seen in the other experiments presented in Section 6.6. Several different hypergraph formulations have been tried, however all produced similarly poor results in practice.

6.5 A Linear Programming Solution to Primary Copy Placement

Another way to simplify the primary copy placement problem is to only consider a binary commit strategy: transactions are either able to commit locally or they are not. This approach bundles core, remote and 2PC commit strategies as a single choice (non-local commit). The cost of executing a transaction tx under this model is 0 for local commit just as before. For non-local commits, at least one message needs to be exchanged, resulting in a higher cost than local. The minimum value of this cost is used: the latency of this one message to the core. Thus, the transaction cost function originally defined in equation (6.2) can be simplified to the form in equation (6.7).

$$\text{cost}(tx, N, p) = \begin{cases} 0 & \text{if } tx.ws = \varnothing \text{ or } \forall t \in tx.ws, o(t) = tx.s \\ l(tx.s) & \text{otherwise} \end{cases} \tag{6.7}$$

The goal of primary copy placement under the simplified cost function in equation (6.7) essentially becomes the edging of as many transactions as possible while still observing network topology restrictions. This can be expressed in terms of a 0-1 integer linear program that tries to maximize the latency savings of a placement in relation to a base placement where no transaction is edged.

$$\begin{aligned} \text{maximize} \sum_{tx \in W} & [tx.w \cdot l(tx.s) \cdot x_{tx}] \\ \text{subject to } & x_{tx_1} + x_{tx_2} \leq 1, \forall tx_1, tx_2 \in W \text{ such that} \\ & tx_1.ws \cap tx_2.ws \neq \varnothing \text{ and } tx_1.s \neq tx_2.s \\ \text{and } & x_{tx} \in \{0,1\}, \forall tx \in W \end{aligned}$$

Figure 6.10: Simplified Cost Primary Copy Placement Problem

Figure (6.10) shows the formulation of such a linear program. The decision variables for this optimization problem are x_{tx} where $x_{tx} = 1 \iff tx$ is edged and $x_{tx} = 0$ otherwise. The only placement constraint this linear program imposes is that at most one of any two transactions with overlapping writesets and different start sites may be edged. If this were not observed, then all tuples common to both writesets would need to be placed at both transactions' start sites at the same time - which is impossible in a single-primary copy system.

The linear program first needs to be solved. This can be done by one of the many off-the-shelf LP solvers. Once solved, converting the list of x_{tx} values into a primary copy placement can be achieved by the following simple procedure: For each edged transaction, place all of the tuples in its writeset at its start site:

$$p(t) = tx.s, \forall tx : x_{tx} = 1 \text{ and } \forall t \in tx.ws$$

All other tuples that remain after this operation are placed at the core.

The example presented in Table 6.2 can be expressed as the following linear program:

$$\begin{aligned} \text{maximize } & 400x_1 + 400x_2 + 600x_3 \\ \text{subject to } & x_1 + x_3 \leq 1 \\ \text{and } & x_2 + x_3 \leq 1 \\ \text{and } & x_1, x_2, x_3 \in \{0, 1\} \end{aligned}$$

There are only 8 possible combinations of values for x_1, x_2 and x_3. The optimal solution is $\{x_1 = 1, x_2 = 1, x_3 = 0\}$ which maximizes the value of the objective function to a value of 800 and satisfies the two constraints. The interpretation of this solution is that transactions 1 and 2 need to be edged. More specifically, TupleA would be placed at transaction 1's starts site (Edge1) and TupleB at transaction 2's start site (Edge2). This is the same optimal placement presented in Table 6.2.

6.6 Experimental Evaluation of Primary Copy Placement

All of the algorithms described in Sections 6.2, 6.3, 6.4 and 6.5 are evaluated in this section using several workloads. The goal is to determine the quality of the produced solutions and the scalability of the algorithms on larger workloads. The quality of the solution is simply determined by its cost as defined in equation (6.1) and is inversely proportional to it.

The algorithms have been implemented in JavaScript (ECMAScript 6th edition) for the Node.js runtime. All experiments have been performed on a server with dual Intel Xeon E5-2620v2 6-core CPUs, 64 GB of DDR3 memory and an Intel S3700 200GB SATA3 SSD for persistent storage, running Ubuntu Server Linux, release 14.04.1 LTS. The Hypergraph partitioning part of the algorithm described in Section 6.4 has been performed using the hMETIS solver, version 1.5.3 [42]. The linear program described in Section 6.5 has been solved by using the IBM ILOG CPLEX Interactive Optimizer version 12.7.0.0 [39].

The testing methodology involves starting from a workload and network topology. There are three such inputs considered which will be described throughout this section. The affinity, greedy, hypergraph and linear programming algorithms are all run on these inputs to produce a valid placement. For the greedy algorithm, two different initial placements were considered. The first is all-core, which places all tuples at the core site. This will be referred to as "Greedy-Core" in the experiments. The second is the highest quality (i.e. lowest cost) solution produced by the other three algorithms. It will be referred to as "Greedy-???" in the experiments, where ??? will be replaced by the name of the algorithm that produced the input initial placement. Random starting placements have also been tried (best out of 10) with the greedy algorithm, but the resulting solutions were the lowest quality among the greedy algorithm solutions, so they haven't been included in the graphs. An all-core placement is also considered as a baseline.

6.6.1 Locality

For the first experiment, the same locality workload and network described in Section 5.3 are used. The goal is to evaluate solution qualities and algorithm runtimes for workloads spanning the entire locality spectrum and to identify any correlation between the locality parameter and these values. In this setup, there are 3 edges in total and all of them have a core round trip latency of 10 ms. Each of the edges, as well as the core, have a bag of 50 tuples associated with them for a total of 200 tuples in the database. 5000 transactions are generated for each of the 3 edges given a workload locality parameter P in the following manner: With probability P, a transaction's writeset will consist of 2 tuples picked uniformly at random from the bag associated with the transaction's start site. With probability $(1 - P)/2$, a transaction's writeset will consist of 2 tuples picked uniformly at random from the bag associated with the core. Otherwise, the transaction's writeset will contain 2 tuples selected uniformly at random from among all 200 available in the database such that at most one is from the bag associated with the transaction's start site and at most one is from the bag record is at the local edge and at most one record is at the core.

As explained in Section 5.3, a higher locality means that more local commits should be possible. The locality parameter P is an approximation of the workload's locality if all the tuples were to be assigned to the site associated with the bag they belong to - the natural placement. However, the primary copy placement algorithms have no knowledge of these bags of tuples, nor the natural placement. All they receive as input are the workload and the network topology. They then attempt to find the best placement with just this given input. As such, the expectation is that the optimal solution is as least as good as the natural one and that the quality of the optimal solution should be directly proportional to the locality parameter P. However, unlike the toy example in Table 6.2, a search of the entire solution space is infeasible and there is no other known method at this point to actually determine the optimal solution for most of these workloads. As such, the experiments in this section only compare among the proposed solutions.

Figure 6.11 shows the solution quality results of this first experiment. As expected, most algorithms' solution quality scaled with the locality parameter P. All algorithms found the zero cost solution for the workload with $P = 1$, but the hypergraph and greedy algorithm with a core starting placement provided relatively poor solutions for $P < 0.9$. The linear programming (LP) and affinity approaches performed worse than the baseline core placement for some low locality workloads, however the simple affinity method seems to have performed the best in most cases. As such, its output has been used as the starting placement for the greedy algorithm as well. However, the greedy algorithm wasn't able to improve on the solutions affinity had found.

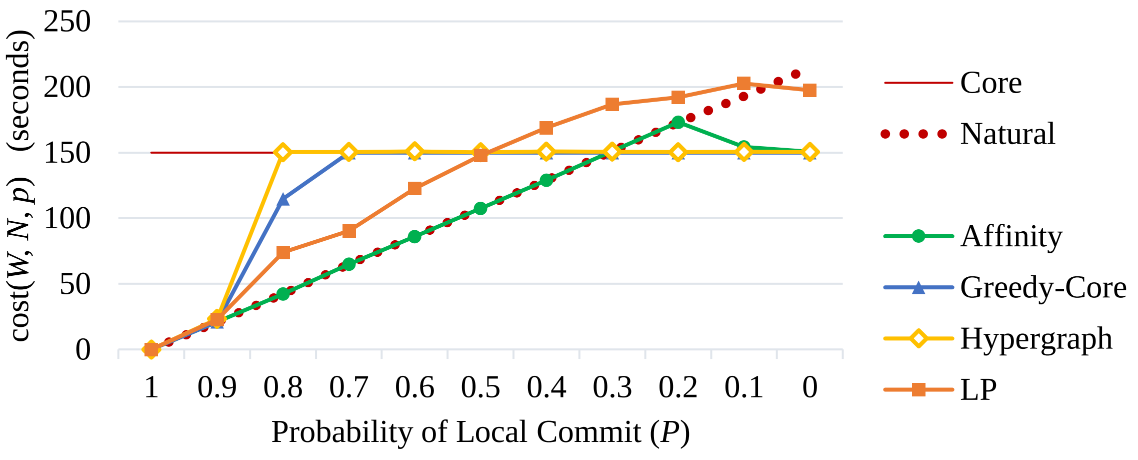


Figure 6.11: Primary Copy Solution Quality Comparison for the Locality Workload

An interesting observation is that none of the proposed algorithms have been able to provide a solution that's better than both the natural and core placements.

The runtimes of all algorithms used in this first experiment are shown in Figure 6.12. Notice the linear programming (LP) algorithm series runs off the top of the chart and isn't shown in the figure for values of P smaller than 0.9. The runtimes for this algorithm are much longer than the others, taking as much as 22 minutes to finish a run for $P = 0.1$.

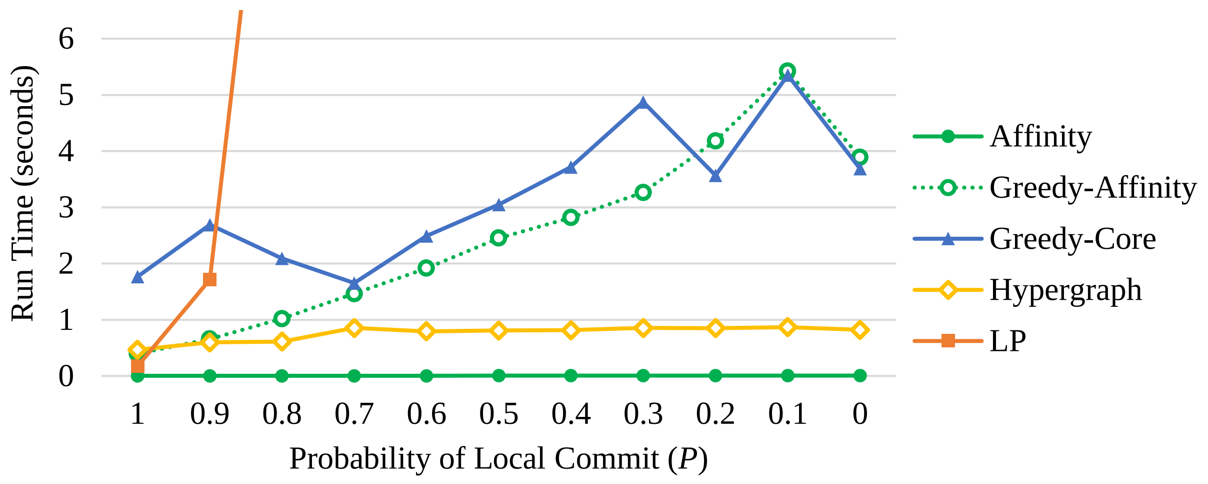


Figure 6.12: Primary Copy Algorithm Runtime Comparison for the Locality Workload

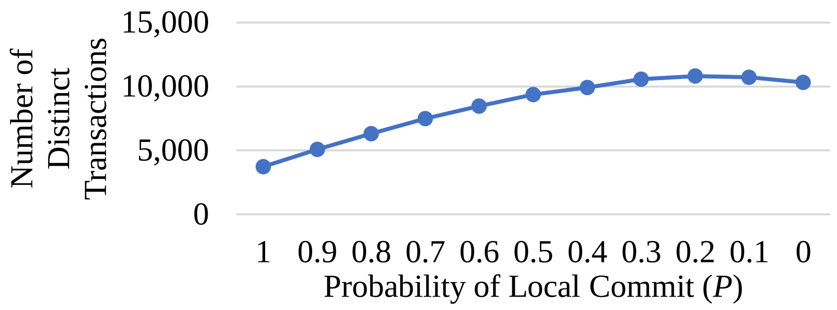


Figure 6.13: Locality Workload Size as a Function of P

The affinity approach is the fastest in all runs due to its simplicity. The hypergraph approach takes roughly the same amount of time for workloads with $P \leq 0.7$. Both the hypergraph and linear programming approaches spend the vast majority of their solving time within their respective off-the-shelf solvers. The input construction and solution interpretation for both these solvers takes up a negligible amount of time.

Figure 6.12 also seems to suggest that there is a correlation between the locality parameter P and the runtime of the greedy, linear programming and hypergraph (for $P > 0.7$) algorithms. However, this is not a direct correlation. Because 15 thousand transactions are being generated using combinations of 200 tuples and 3 start sites, many of these transactions end up being duplicates. All such duplicates are merged together in a single transaction of a higher weight (i.e. the merged transaction weight is equal to the number of transactions merged). Due to the transaction construction mechanism, there is a correlation between P and the final size of the workload which is shown in Figure 6.13. It is then the size of the workload that correlates with the algorithm runtimes in the cases of LP and Greedy-Affinity, the latter of which only performs one iteration as it is unable to improve upon Affinity's solution as already mentioned. The Greedy-Core algorithm also slows down when the workload increases, but has some slower runs at high values of P because in those cases it is able to refine the original core placement over several iterations.

6.6.2 Hot/Cold Tuples

The second experiment considers a workload consisting of a mix of popular (hot) tuples and less frequently accessed (cold) tuples. The goal of this experiment is to be able to vary the tuple contention of the workload and measure the effects on the solution quality of the primary copy placement algorithms. Informally, the tuple contention of a workload is the amount of overlap that occurs between the writesets of transactions in the workload that have different starting sites. The expectation here is that less contention makes a partitioning easier resulting in better quality solutions.

The workload for this scenario is constructed in the following manner: The database contains 1000 tuples. Each of the 10 edges in the network gets assigned a different zipfian distribution of these tuples. More specifically, the same parameter S is used for all edge zipfian distributions in order to randomly pick index values from 1 to 1000; but each edge has a different randomly assigned ordering (permutation) of the set of tuples these indexes refer to. 500 transactions are generated for each of the 10 edges by picking a number of tuples from the edges' distribution to make up the writeset. The number of tuples in the writeset of a transaction is chosen uniformly at random to be 1, 2 or 3.

Because of the properties of zipfian distributions, a larger value of S means transactions at each edge are likely concentrated on a smaller set of tuples that are being accessed more frequently (hot tuples). The smaller the set of tuples commonly touched by transactions at an edge, the less likely it is that these transactions' writesets overlap with other edges' transactions' writesets. This results in a lower level of tuple contention for the overall workload.

Figure 6.14 shows the solution quality results of this experiment. The hypergraph approach doesn't provide good quality solutions relative to the other approaches. Furthermore its solution quality doesn't improve for higher values of S as expected. This is likely due to the poor performance of hMETIS [42] as mentioned in Section 6.4.2. The linear programming approach does improve at higher values of S, but does so slower than affinity and greedy. Affinity and greedy are the best performers in this experiment, with greedy starting from a core placement being marginally better. The output of the affinity algorithm has also been used as a starting place-

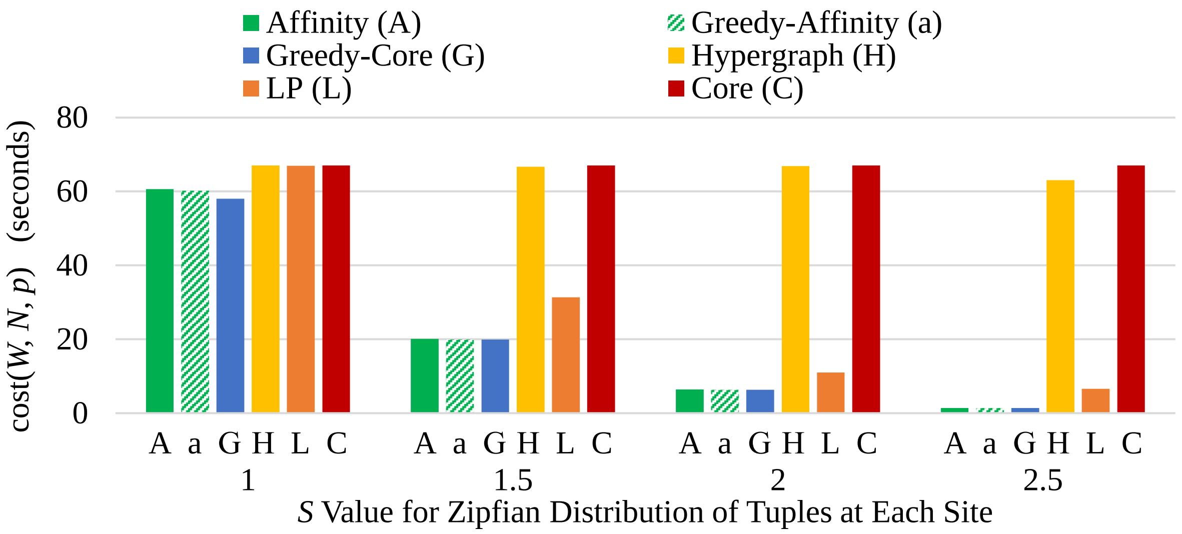


Figure 6.14: Primary Copy Solution Quality Comparison for the Hot/Cold Tuples Workload

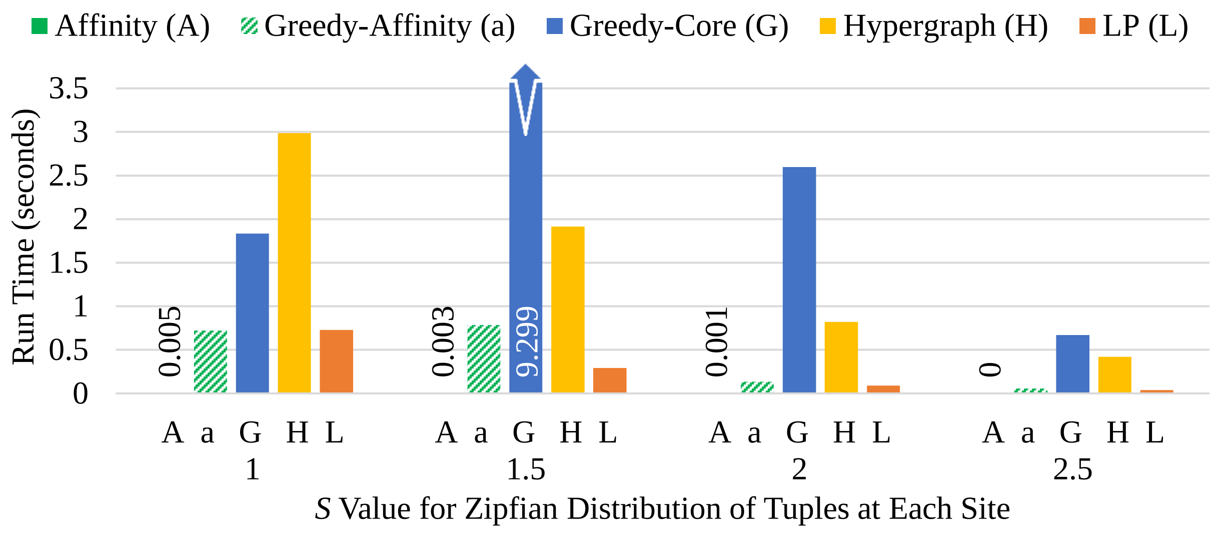


Figure 6.15: Primary Copy Algorithm Runtime Quality Comparison for the Hot/Cold Tuples Workload

ment for the greedy algorithm. This is represented by the Greedy-Affinity series in Figure 6.14). The improvement that Greedy-Affinity achieves over Affinity in this experiment is an average cost reduction of only 0.5%, which isn't perceivable in Figure 6.14.

The second experiment's runtime results are shown in Figure 6.15. The Affinity algorithm's runtimes are shown in the graph as numbers as they are too small to register visually. The arrow on the Greedy-Core series for $S = 1.5$ indicates the runtime value, which is also presented as a number in the graph, extends outside the axis bounds. In the case of $S = 2.5$, Affinity's runtime is shorter than the 1 ms precision of the timer used to measure these.

Most algorithms, with the exception of Greedy-Core scale with the workload size, similar to the experiment in Section 6.6.1. Figure 6.16 shows that as the number of hot tuples increases (with S), the effective database size used for transactions decreases, leading to more duplicates and fewer distinct transactions in the workload. In these relatively smaller workloads, when compared to the experiment in Section 6.6.1, the relationship between the Hypergraph and LP algorithms is seen more clearly. LP runtimes grow exponentially with the workload size, while Hypergraph runtimes grow linearly with the workload size up to a point beyond which they stop growing. Figure 6.17 shows the number of iterations it took to reach a solution that may no longer be improved by the greedy algorithm. The Greedy-Core series runtimes are correlated with this number to a higher degree than they are to workload sizes.

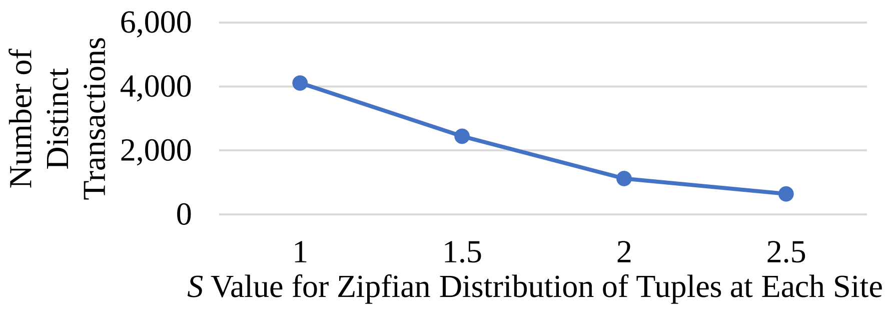


Figure 6.16: Hot/Cold Tuples Workload Size as a Function of S

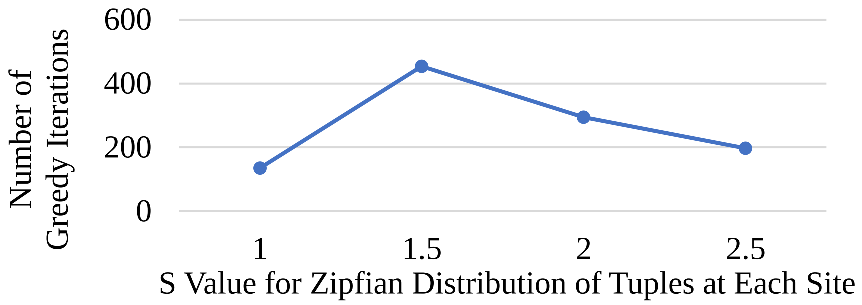


Figure 6.17: Greedy-Core Iterations as a Function of S for the Hot/Cold Tuples Workload

6.6.3 RUBiS, Once Again

The third experiment considers a more realistic benchmark. There are no particular expectations with this experiment as the goal is simply to observe the quality of the primary copy placement solutions for a more realistic workload and network setting. For this purpose, the RUBiS benchmark [84] is once again used. More specifically, the problem input used in this experiment is obtained from the SpearDB workload derived from RUBiS, which was presented in Section 5.5.

The topology from the “Latency” column of Table 5.10 is used for the network part of the input. The edge latencies in Table 5.10 represent a geo-distributed system of major population centers at a continental scale based on the geography of the United States.

The workload part of the input is constructed from the workload trace of an actual run of the RUBiS experiment in Section 5.5 with 80% locality - this is one of the 20 runs (selected at random) used in the results presented in Figure 5.11. The workload traces (lists of SpearDB transactions) of the RUBiS experiment from all edges have then been taken and put together in a single list ordered by the wall-clock time (i.e. real world time as opposed to CPU time) of the start of transactions. All information from these workload traces other than transactions' start sites and writesets were then removed. This included removing any transactions without

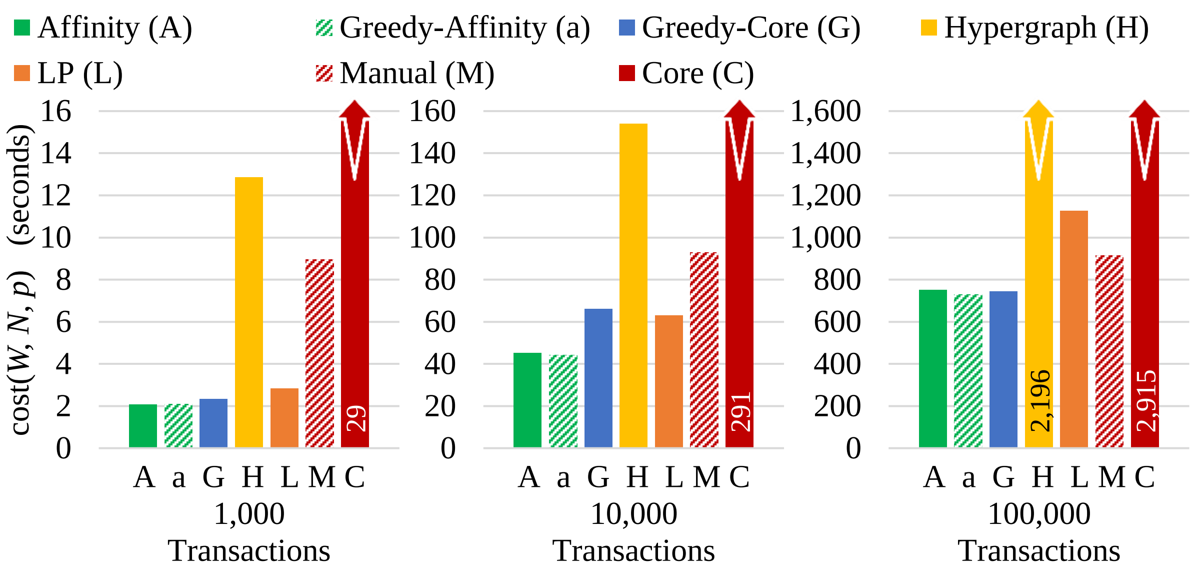


Figure 6.18: Primary Copy Solution Quality Comparison for Subsets of the Rubis Workload

updates as they are not relevant to primary copy placement. The first thousand, 10 thousand and 100 thousand transactions that remained in the list were then used to make up three individual workload inputs. In each of the three cases the database was simply the union of all transaction writesets in the workload input used.

Figure 6.18 shows the experiment's solution quality results for all three cases. The core placement baseline is shown in the figure, but isn't represented to scale due to its higher cost. Arrows are used to indicate when values fall outside the bounds of the axis and a numeric representation is used instead. The original manual placement schema described in Table 5.9 has been used as an additional baseline primary copy placement solution (the Manual series in the figure). The affinity based solution is again the top contender and the average cost reduction achieved for Greedy-Affinity over Affinity is 1.8%. More importantly, Affinity, as well as both greedy based approaches provide better solutions in terms of quality than the manual baseline. This enforces the need to develop algorithmic data placement solutions over relying on domain experts for complex workloads.

An interesting observation is that both baselines scale almost perfectly linearly with the size of the workload. The optimal partitioning likely follows a similar trend. However, the algorithmic solutions' relative quality to these baselines becomes increasingly poorer as the workload grows larger. The likely reason is that the missed optimization opportunities of these approximate algorithms get compounded.

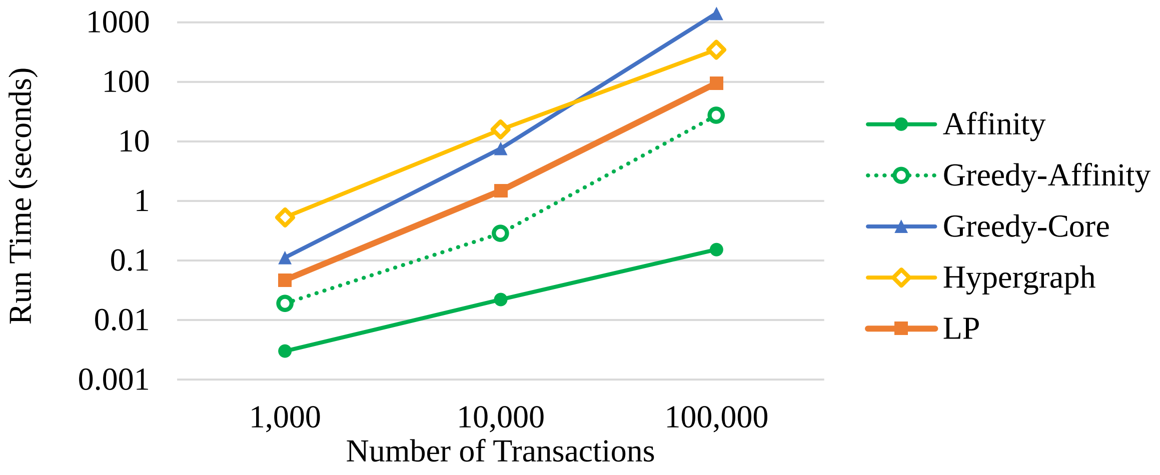


Figure 6.19: Primary Copy Algorithm Runtime Scaling for the Rubis Workload

The algorithm runtimes for this experiment are presented in Figure 6.19. Note that both axis in the figure use a logarithmic scale. The greedy algorithms scale poorly for large workloads, but the others seem to scale close to linear in this case.

In this experiment, the size of the workload is intentionally varied, not just as a result of merging transactions. The RUBiS workload has very few duplicate transactions - between 0.1% and 1.6% among the three workloads sizes observed. Furthermore, contention (writeset overlap) is much lower than in the previous experiments and workload sizes grow up to an order of magnitude larger. The Hypergraph and LP solvers exhibit different runtime characteristics for this type of workload. Hypergraph runtimes continue scaling with the workload size in this case, while the LP solver is able to solve much larger workloads much faster - probably due to the decreased contention. Most of the runtime these two algorithms spend to find the solution is again taken up by the dedicated solvers. All other input and output parsing for Hypergraph and LP take up a negligible amount of time.

6.6.4 Greedy Iterations

This section takes a closer look at the iterations of the greedy algorithm. Appendix C describes the optimized greedy primary copy placement algorithm used in the prototype implementation. The first iteration step of this algorithm includes some data structure initializations that help reduce computation time of the cost function in subsequent iteration steps of the algorithm. As a result, the first iteration is expected to be longer than subsequent ones. The aim of this section is to observe the practical runtime and solution quality improvement of individual iteration steps.

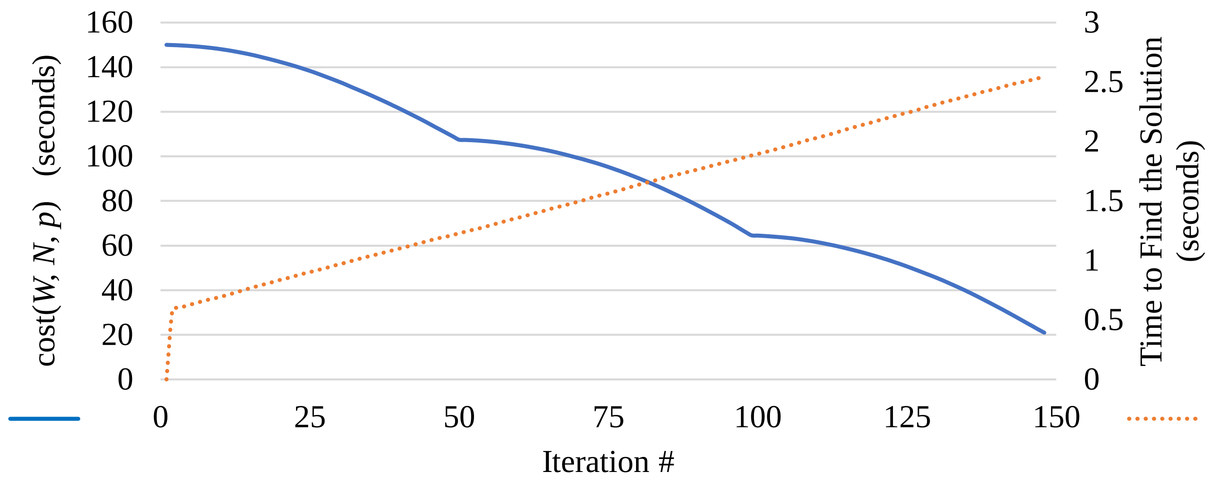


Figure 6.20: Example Greedy Solution Quality and Solving Times as a Function of Algorithm Iterations

Figure 6.20 shows a representative result obtained from running the greedy algorithm with a core starting placement on the locality workload with $P = 0.9$. This is the same workload presented before in Figure 6.11 and was chosen because of the high number steps (iterations) it took to reach a relatively good quality solution.

The number of steps/iterations is plotted on the x-axis of Figure 6.20 while the cost of the placement reached so far is on the left hand y-axis and the total cumulative time to reach the current placement is on the right hand y-axis. As expected the solution quality improves with every iteration (cost goes down). The improvement is close to linear with time. The three consecutive curvatures that make up the shape of the solution quality are due to the fact that this run of the algorithm consists of three consecutive groups of 49 transactions with the same start site (but a different start site per group) being edged. On the other y-axis, the time to finish the first step of the greedy algorithm is clearly larger than subsequent steps which all take roughly the same amount of time as each other. The close-to-linear improvement of the solution quality and linearly increasing total time to solution after the first step have been observed in virtually all other runs of the greedy algorithm.

6.7 Primary Copy Placement Conclusions

Solving the primary copy placement problem consists of identifying an optimal disjoint partitioning of the tuples of a database based on geographic access patterns of tuples by transactions in order to minimize transaction processing time. Several algorithmic solutions have been proposed and the quality of their results empirically tested. An important outcome of these tests is that some of the algorithmic partitioners have provided higher quality solutions than a manual expert placement of tuples for the RUBiS workload, as presented in Figure 6.18.

The surprising conclusion is that a very simple heuristic (affinity - presented in Section 6.2) that disregards much of the relevant input information of the problem outperforms more complex approaches in practical testing in runtime, but also in solution quality - which is unexpected. Although the affinity technique has shown its limitations on a small example (Table 6.2) when compared to the other techniques, in virtually all larger scale experiments from Section 6.6 it has outperformed them. Part of this result may however be attributed to the poor performance of the 3rd party solvers available for hypergraph partitioning.

The only other contender that should be considered in practice, according to the experiments performed in Section 6.6, is the greedy algorithm presented in Section 6.3. The greedy algorithm may be run with a core starting placement to mitigate the few cases observed where affinity is unable to produce solutions of better quality than an all-core placement. Greedy can also attempt to improve on the affinity solution by using it as a starting placement. A small improvement has been observed in a few experimental cases in Section 6.6 and a single iteration is required for cases where the solution may not be further improved. Furthermore this property of the greedy algorithm to potentially improve upon solutions may be used with the results of any other primary copy placement solvers that may be developed in the future.

Chapter 7

Secondary Copy Placement

The execution phase of transaction processing in a database like SpearDB is influenced by the location of all copies of tuples - both primary and secondary. For the former, Chapter 6 has described the problem of optimal placement of primary copies with the goal of minimizing network latency during the commit phase of transactions. This chapter looks into the problem of static secondary copy placement, which aims to add secondary copies of tuples at each edge site. The goal is to minimize the latency of the execution phases of transactions executed at that edge. The constraints are not adding more than a specified amount of network traffic between the edge and the core, while also imposing an edge-specific maximum total size of all copies.

To present secondary copy placement, the same notation is used that was introduced in Table 6.1. However, for secondary copy placement, further information about the database tuples that are read by each transaction is required. Thus, each transaction tx in the workload has a readset ($tx.rs \subseteq D$) in addition to its writeset ($tx.ws$). Additionally, all tuples t in the database have a size: size(t). All components of the input to the secondary copy placement problem are summarized in Table 7.1. The output of secondary copy placement is a set of tuples S_e for which secondary copies will be installed at edge e.

In order to understand the goal of secondary copy placement (the objective function), it is important to observe the consequences of adding a secondary copy for a tuple $t \in D$ at an edge e. Placing a secondary copy of t at e has the following effects:

- Whenever t is read at edge e it will be served locally, saving an entire round trip to the core - which is equivalent to $l(e)$ amount of latency and size(t) - the tuple size - amount of bandwidth;
- When t is written to by transactions (anywhere in the system), there are no additional messages involved in executing or committing the transaction; However, during the prop-

Table 7.1: Secondary Copy Placement Input Components

Component	**Description**
D	the database - a set of tuples
E	the set of edge sites
$e \in E$	an edge - for which the problem is defined
$l(e)$	the round trip network latency from edge e to the core
$maxB_e$	the maximum bandwidth threshold for edge e
$maxT_e$	the maximum allowed total size of tuple copies (primary + secondary) at edge e
W	a workload consisting of transactions
$\forall tx \in W$	a transaction $tx = \langle s, w, rs, ws \rangle : \begin{cases} tx.s \in E & \text{start site} \\ tx.w & \text{weight} \\ tx.rs \subseteq D & \text{readset} \\ tx.ws \subseteq D & \text{writeset} \end{cases}$
size(t), $t \in D$	the size (in bytes) of tuple t
$p : D \rightarrow E \cup \{\text{core}\}$	a primary copy placement

agation phase of the transaction additional network traffic between the core and the edge is required to propagate the update of t to the secondary copy at the edge.

These consequences can be expressed formally as follows. Let $L_e(t)$ and $B_e(t)$ denote the latency savings and added network traffic respectively of placing a secondary copy of tuple t at edge e. Then the latency savings are simply the number of local reads of the tuple multiplied by the round trip time from edge e to the core. This is expressed in equation (7.1). The added bandwidth is expressed in equation (7.2) as the tuple size multiplied by difference between the numbers of global writes of the tuple (which add network traffic in the propagation phase) and its local reads (which save traffic in the execution phase).

$$L_e(t) = \left(\sum_{tx \in W : tx.s = e \text{ and } t \in tx.rs} tx.w \right) \cdot l(e) \tag{7.1}$$

$$B_e(t) = \left[\left(\sum_{tx \in W : t \in tx.ws} tx.w \right) - \left(\sum_{tx \in W : tx.s = e \text{ and } t \in tx.rs} tx.w \right) \right] \cdot \text{size}(t) \tag{7.2}$$

Thus, the secondary copy placement problem at edge e can be expressed as a 0-1 integer linear program. The decision variables $x_t \in \{0, 1\}$ (one for each tuple $t \in D$ that doesn't have a primary copy at e) indicate whether tuple t has a copy at edge e ($x_t = 1$) or not ($x_t = 0$).

$$\begin{aligned}
\text{maximize} \quad & \sum_{t \in D : p(t) \neq e} L_e(t) \cdot x_t \\
\text{subject to} \quad & \sum_{t \in D : p(t) \neq e} B_e(t) \cdot x_t \leq maxB_e \\
\text{and} \quad & \sum_{t \in D : p(t) \neq e} \text{size}(t) \cdot x_t + \sum_{t \in D : p(t) = e} \text{size}(t) \leq maxT_e \\
\text{and} \quad & x_t \in \{0, 1\}
\end{aligned}$$

Figure 7.2: Secondary Copy Placement Problem at Edge e

The goal is to choose values for all decision variables x_t such that the sum of all latency savings for all tuples t that have been chosen ($x_t = 1$) is maximized, while the added bandwidth for the same tuples doesn't exceed $maxB_e$ and the total tuple size at edge e doesn't exceed $maxT_e$. The program, which is shown in Figure 7.2, can be solved by an off-the-shelf linear program solver.

Bear in mind that since primary copies also act as secondary copies during the execution phase of transactions, there is never a need to place a secondary copy of a tuple at a site that already has a primary copy of that tuple. As such these tuples can be excluded from consideration altogether (i.e. there are no decision variables x_t for tuples with primary copies at edge e). Thus, secondary copy placement is dependent on the result of primary copy placement, which is the reason a primary copy placement p is also required as input. Furthermore, if there are any additional limitations imposed on the set of viable tuples (e.g. imposed manual placement of a small subset of tuples; legal restrictions on the geographical location of sensitive data; etc.), the problem may be generalized to accommodate these limitations by fixing the value of a subset of decision variables x_t to either 0 (meaning that a secondary copy may not be placed at this edge) or 1 (meaning that a secondary copy must be placed at this edge).

The problem of secondary copy placement can be solved independently for each edge site. However the entire global workload W is required as input so the problem is not completely independent from what occurs in the rest of the system. In order to come up with the complete global list of secondary copies, a total of $|E|$ different instances of the problem need to be solved - one for every edge. Notice that secondary copy placement is not required at the core in a system like SpearDB, which holds a full copy of the database there.

Table 7.3: Example Secondary Copy Placement Setting

Workload

Start Site	**Weight**	**Readset**	**Writeset**
Edge1	55	TupleA	TupleB
Edge1	45	TupleB	TupleA
Edge1	25	TupleC	-
Edge2	55	TupleA	TupleA
Edge2	20	TupleA, TupleB	TupleC

Thresholds: $maxB_{\text{Edge1}} = maxB_{\text{Edge2}} = 40\,\text{bytes}$
$maxT_{\text{Edge1}} = maxT_{\text{Edge2}} = 1\,\text{GB}$

Network Topology

Edge	**Latency**
Edge1	10 ms
Edge2	10 ms

Primary Copies

Tuple	**Site**
TupleA	Edge2
TupleB	CORE
TupleC	CORE

7.1 Secondary Copy Placement Example

This section presents a small example of secondary copy placement. Consider the setting presented in Table 7.3 which includes a workload, network topology, primary copy placement and threshold bandwidth values for all edges. From it, two instances of secondary copy placement can be extracted. For Edge1, its associated latency is $l(\text{Edge1}) = 10\,\text{ms}$ and the maximum bandwidth threshold for the edge is $maxB_{\text{Edge1}} = 40\,\text{bytes}$, while the maximum total tuple size is $maxT_{\text{Edge1}} = 1\,\text{GB}$. The entire workload and primary copy placement from Table 7.3 are used as input as well. They will be referred to as W and p respectively. There are no primary copies at this edge. The final input component to this first instance of the problem is the database $D = \{\text{TupleA}, \text{TupleB}, \text{TupleC}\}$.

The second instance of the problem is defined for Edge2 and is similar to the first. Its input consists of the following: $l(\text{Edge2}) = 10\,\text{ms}$, $maxB_{\text{Edge2}} = 40\,\text{bytes}$, $maxT_{\text{Edge2}} = 1\,\text{GB}$, D, W and p. The relevant portion of the primary copy placement p for this instance of the problem is $p(\text{TupleA}) = \text{Edge2}$.

The values for all latency savings as defined in equation (7.1) and added network traffic as

Table 7.4: Added Traffic and Latency Savings for Example Input in Table 7.3

Tuple	**Edge1**		**Edge2**	
t	$L_{\text{Edge1}}(t)$	$B_{\text{Edge1}}(t)$	$L_{\text{Edge2}}(t)$	$B_{\text{Edge2}}(t)$
TupleA	$55 \cdot 10 = 550\,\text{ms}$	$(45+55) - 55 = 45$	$\cancel{(55+20) \cdot 10}$	$\cancel{(45+55) - (55+20)}$
TupleB	$45 \cdot 10 = 450\,\text{ms}$	$55 - 45 = 10$	$20 \cdot 10 = 200\,\text{ms}$	$55 - 20 = 35$
TupleC	$25 \cdot 10 = 250\,\text{ms}$	$20 - 25 = -5$	$0 \cdot 10 = 0\,\text{ms}$	$20 - 0 = 20$

defined in equation (7.2) are summarized in Table 7.4. For simplicity, the size of all tuples in the database was assumed to be 1 byte - small enough to allow the entire database to fit within the maximum 1 GB threshold of the edges. The values for $L_{\text{Edge2}}(\text{TupleA})$ and $B_{\text{Edge2}}(\text{TupleA})$ are crossed out in Table 7.4 because TupleA should be excluded from consideration as a secondary copy candidate for Edge2 due to the fact that it has a primary copy there: $p(\text{TupleA}) = \text{Edge2}$.

For Edge1 a maximum of 800 ms of latency can be saved by installing secondary copies of TupleA and TupleC there. As Table 7.4 shows, the additional network traffic incurred for these two secondary copies would be $45 - 5 = 40 = maxB_{\text{Edge1}}$ bytes. Notice that TupleC in particular is responsible for 250 ms of latency savings while at the same time reducing network traffic. This is to be expected in cases where a tuple is read locally more often than it is globally written to.

For Edge2, TupleA needs to be taken out of consideration because it already has a primary copy at this edge. What is left is an option between TupleB and TupleC since both of them cannot be replicated at this edge without incurring a bandwidth penalty greater then $maxB_{\text{Edge2}}$. TupleB is the better choice since it provides a greater latency benefit of 200 ms. Again notice that TupleC is a special case here: since it is never read at Edge2, there would be no latency savings from installing a secondary copy.

7.2 Problem Complexity

To understand the source of complexity in the secondary copy placement problem, consider the following simple greedy solution. First, choose the tuple t with the minimum $\frac{B_e(t)}{L_e(t)}$ value such that the total additional network bandwidth for all chosen tuples is smaller than the threshold $maxB_e$. For simplicity, also assume that the maximum tuple size threshold $maxT_e$ is large enough as to never be exceeded. Then repeat the process until a choice isn't possible anymore without violating the bandwidth constraint. This strategy works for negative values of $B_e(t)$ as well. For the corner case of $L_e(t) = 0 \Rightarrow B_e(t) \geq 0$, the value of $\frac{B_e(t)}{L_e(t)} = +\infty$ as there is no point in adding a secondary copy that just increases bandwidth without any benefit to latency.

At first glance it might seem like this algorithm should provide an optimal solution. Using it to solve the input defined in Table 7.3 would order the tuples based on their $\frac{B_E(t)}{L_E(t)}$ values.

Table 7.5: Latency Savings per Added Bandwidth Unit for Example Input in Table 7.3

t	$\frac{B_{\text{Edge1}}(t)}{L_{\text{Edge1}}(t)}$	$\frac{B_{\text{Edge2}}(t)}{L_{\text{Edge2}}(t)}$
TupleA	≈ 0.082 ms	~~≈ 0.033 ms~~
TupleB	≈ 0.022 ms	0.175 ms
TupleC	-0.02 ms	∞

For Edge1, this order would be TupleC, followed by TupleB and then by TupleA as can be seen in Table 7.5. However, after TupleC and TupleB are chosen, there would be no more room in the bandwidth budget for TupleA. Thus, this greedy algorithm would reach the conclusion that $S'_{\text{Edge1}} = \{\text{TupleC}, \text{TupleB}\}$. This is a different solution than the optimal found before ($S'_{\text{Edge1}} \neq S_{\text{Edge1}}$). While both solutions include TupleC, the solution S_{Edge1} includes TupleA, while S'_{Edge1} includes TupleB. Since $L_{\text{Edge1}}(\text{TupleA}) > L_{\text{Edge1}}(\text{TupleB})$ as seen in Table 7.4, the greedy algorithm solution S'_{Edge1} is worse.

The problem of secondary copy placement is in fact a variation on a well known NP-complete problem called the *0-1 knapsack problem*. Theorem 7.3 states that any instance of 0-1 knapsack can be reduced to secondary copy placement in polynomial time. The corollary of this reduction is that secondary copy placement is NP hard.

Theorem 7.3 *Any instance of the 0-1 knapsack problem can be reduced to secondary copy placement in polynomial time.*

Proof: The 0-1 knapsack problem can be formulated as follows: Given a set of n items numbered from 1 to n, each with a positive weight w_i and a positive value v_i along with a maximum weight capacity C,

$$\begin{aligned} \text{maximize } & \sum_{i=1}^{n} v_i \cdot x_i \\ \text{subject to } & \sum_{i=1}^{n} w_i \cdot x_i \leq C \\ \text{and } & x_i \in \{0,1\} \end{aligned}$$

As seen in Table 7.1, a secondary copy placement problem requires as input: a database D, a set of edges E, an edge for which the problem is defined e and its latency to the core $l(e)$, a workload W, a primary copy placement p, and maximum bandwidth and total tuple size thresholds: $maxB_e$ and $maxT_e$, respectively. Assume an arbitrary instance of the 0-1 knapsack problem is provided. From it, a secondary copy placement problem is formulated as follows:

- $D = \{t_1, t_2, ..., t_n\}, \text{size}(t_i) = 1\,\text{byte}$, for all tuples ($n$ tuples of size 1 byte in the database);
- $E = \{e_1, e_2\}$ (there are 2 edges in the system);
- $l(e_1) = 1\,\text{ms}$ (edge e_1 for which the problem is defined is 1 ms away from the core);
- $maxB_e = C$
- $maxT_e = n\,\text{bytes}$

- $W = \{tx_1, tx_2, ..., tx_n, tx'_1, tx'_2, ..., tx'_n\}$, where:
    - $tx_i.s = e_1$ and $tx'_i.s = e_2, \forall i \in \overline{1,n}$
    - $tx_i.rs = \{t_i\}$ and $tx'_i.rs = \varnothing, \forall i \in \overline{1,n}$
    - $tx_i.ws = tx'_i.ws = \{t_i\}, \forall i \in \overline{1,n}$
    - $tx_i.w = v_i$ and $tx'_i.w = w_i, \forall i \in \overline{1,n}$
- $p(t) = \text{core}, \forall t \in D$ (all primary copies are at the core - i.e. not at edge e_1);

The complexity of this conversion is $O(n)$ - where n is the size (i.e. number of items) of the 0-1 knapsack problem instance. Thus the conversion can occur in polynomial time.

The problem of secondary copy placement is defined in Figure 7.2. The latency savings formula from equation (7.1) can be substituted into the objective function; and the added bandwidth formula from equation (7.2) can be substituted into the constraints of the secondary copy placement problem. Then, renaming the decision variables to y_t to avoid confusion provides the following equivalent formulation:

$$\begin{aligned}
\text{maximize} &\sum_{t \in D} \left(\sum_{\forall tx \in W : tx.s = e \text{ and } t \in tx.rs} tx.w \right) \cdot l(e) \cdot y_t \\
\text{subject to} &\sum_{t \in D} \left[\left(\sum_{\forall tx \in W : t \in tx.ws} tx.w \right) - \left(\sum_{\forall tx \in W : tx.s = e \text{ and } t \in tx.rs} tx.w \right) \right] \cdot \text{size}(t) \cdot y_t \leq maxB_e \\
\text{and} &\sum_{t \in D} \text{size}(t) \cdot y_t + 0 \leq maxT_e \\
\text{and } & y_t \in \{0,1\}, \forall t \in D
\end{aligned}$$

Then, substituting the values obtained from the 0-1 knapsack problem into this linear program the following formulation is obtained:

$$\begin{aligned}
\text{maximize} &\sum_{i=1}^{n} \left(tx_i.w \cdot y_{t_i} \right) \\
\text{subject to} &\sum_{i=1}^{n} \left[\left(tx'_i.w + \cancel{tx_i.w} - \cancel{tx_i.w} \right) \cdot y_{t_i} \right] \leq C \\
\text{and } & n \leq n \\
\text{and } & y_{t_i} \in \{0,1\}, \forall i \in \overline{1,n}
\end{aligned}$$

Since $tx_i.w = v_i$ from the way it has been defined, the objective function of this instance of secondary copy placement is the same as the arbitrary instance of 0-1 knapsack from which it has been defined. Similarly, since $tx'_i.w = w_i$ and $n \leq n$ is always true, this instance of secondary copy placement also has the same constraints as the arbitrary instance of 0-1 knapsack. Thus an optimal solution to this secondary copy placement instance ($S_e = \{t_i : y_{t_i} = 1\}$) can be easily converted to an optimal solution to the 0-1 knapsack problem by the following (polynomial time) function:

$$x_i = \begin{cases} 0 & \text{if } t_i \notin S_e \\ 1 & \text{if } t_i \in S_e \end{cases}, \forall i \in \overline{1, n}$$

Q.E.D.

Chapter 8

Conclusions

This thesis provides a solution to the problem of latency caused by the distance between end users and the data required by the applications they use. The solution takes the form of a database management system for distributed applications at the edge of the cloud. A prototype implementation of this database management system has been built and is called SpearDB. SpearDB is a geographically distributed, replicated database service that takes advantage of users' proximity to the edge of the cloud in order to optimize request response times. SpearDB makes it possible for applications, as well as the data that they rely on, to be pushed to the edge of the cloud, where clients can communicate with the application quickly. SpearDB provides such edge-dispersed applications with location-transparent access to a shared, persistent database.

SpearDB supports transactions and provides strong consistency guarantees for them in the form of parallel snapshot isolation (PSI). It is designed to be used by applications with geographically localized access patterns. That is, applications for which clients in different geographic regions make heavy use of different parts of the database.

The prototype implementation has been evaluated using several micro-benchmarks in a server cluster in which synthetic network latency can be injected in order to simulate the effect of a controllable amount of geographic distance between servers. A key-value version of the RUBiS auction service application has been developed for additional evaluation of SpearDB. For this application, it is shown that running RUBiS on SpearDB cuts RUBiS request response times in half, relative to a centralized baseline that did not exploit the edge. This improvement was not uniform across the application. Some RUBiS operations are more localizable than others. SpearDB speeds such operations up dramatically. Other operations are not as easy to localize at an edge. For such cases, performance with SpearDB is about the same as in the centralized case.

The performance of SpearDB relies on the workload and data placement of tuples at edges. So, in order to increase performance, automatic partitioning algorithms have been proposed for placing primary and secondary copies. Primary copy placement is approached first - independently from secondary copy placement. Several algorithms are proposed to solve primary copy placement, including a greedy heuristic that can improve upon an existing data placement obtained by any other means.

In addition to several micro-benchmarks, the RUBiS auction service workload is used again to provide an input for the proposed partitioning algorithms. The quality of the solutions provided as output by these algorithms have been ranked and compared between one another as well as relative to a centralized baseline. Overall, a simple placement heuristic that ignores transactional tuple groupings has proven to be the best approach in the empirical testing, outperforming more complex algorithms in both speed and, more importantly, solution quality. Furthermore, for the RUBiS workload, several of the proposed algorithmic approaches have provided better quality solutions than a domain expert manual data placement.

The problem of secondary copy placement, which is approached last, is dependent on the primary copy placement partitioning. This problem is proven to be NP-hard as well as a generalization of the NP-Complete 0-1 knapsack problem.

Overall, the two tiered core-edge approach and commit protocols proposed have shown themselves to be very adept at improving client perceived latency in a geo-distributed setting. SpearDB performs very well on the edge-localizable workloads for which it has been built. Furthermore, because of the introduction of the core in the system, even non-localizable workloads can be executed under latency conditions that aren't any worse than a monolithic database system given the right data partitioning. The transactional mechanism, which has been designed for general purpose use, has proven robust and easy to use when programming the test applications; and Parallel Snapshot Isolation guarantees provide strong consistency which makes reasoning about transactions and interactions between database operations very straight forward.

SpearDB has been designed to scale out by adding more edges to the system. This thesis doesn't address the problem of scaling out the individual sites aside from allowing a wide variety of database systems - which include distributed solutions - to be used for data storage.

References

[1] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, and Harbinder Bhogan. Volley: Automated data placement for geo-distributed cloud services. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '10, pages 17–33, San Jose, CA, USA, April 2010. USENIX Association.

[2] Marcos K. Aguilera, Wojciech Golab, and Mehul A. Shah. A practical scalable distributed B-tree. *Proceedings of the VLDB Endowment*, 1(1):598–609, August 2008.

[3] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, SOSP '07, pages 159–174, Stevenson, WA, USA, October 2007. ACM.

[4] Amazon. Amazon web services. https://aws.amazon.com/.

[5] Khalil Amiri, Sanghyun Park, Renu Tewari, and Sriram Padmanabhan. DBProxy: A dynamic data cache for web applications. In *Proceedings of the 19th International Conference on Data Engineering*, ICDE '03, pages 821–831, Bangalore, India, March 2003. IEEE Computer Society.

[6] Ashok Anand, Chitra Muthukrishnan, Steven Kappes, Aditya Akella, and Suman Nath. Cheap and large CAMs for high performance data-intensive networked systems. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '10, pages 433–448, San Jose, CA, USA, April 2010. USENIX Association.

[7] Catalin Alexandru Avram, Kenneth Salem, and Bernard Wong. Latency amplification: Characterizing the impact of web page content on load times. In *33rd IEEE International Symposium on Reliable Distributed Systems Workshop on Planetary-Scale Distributed Systems*, W-PSDS '14, pages 20–25, Nara, Japan, October 2014. IEEE Computer Society.

[8] Jason Baker, Chris Bond, James Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing

scalable, highly available storage for interactive services. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*, CIDR '11, pages 223–234, Asilomar, CA, USA, January 2011. CIDR Conference.

[9] Michael Till Beck, Martin Werner, Sebastian Feld, and Thomas Schimper. Mobile edge computing: A taxonomy. In *Proceedings of the 6th International Conference on Advances in Future Internet*, AFIN '14, pages 48–54, Lisbon, Portugal, November 2014. IARIA.

[10] Anna Bouch, Allan Kuchinsky, and Nina Bhatti. Quality is in the eye of the beholder: meeting users' requirements for internet quality of service. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, CHI '00, pages 297–304, The Hague, The Netherlands, April 2000. ACM.

[11] Michael Butkiewicz, Harsha V. Madhyastha, and Vyas Sekar. Understanding website complexity: measurements, metrics, and implications. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, SIGCOMM '11, pages 313–328, Berlin, Germany, November 2011. ACM.

[12] David G. Campbell, Gopal Kakivaya, and Nigel Ellis. Extreme scale with full SQL language support in Microsoft SQL Azure. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 1021–1024, Indianapolis, IN, USA, June 2010. ACM.

[13] Emmanuel Cecchet, Veena Udayabhanu, Timothy Wood, and Prashant Shenoy. BenchLab: An open testbed for realistic benchmarking of web applications. In *Proceedings of the 2nd USENIX Conference on Web Application Development*, WebApps '11, pages 37–48, Portland, OR, USA, June 2011. USENIX Association.

[14] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 205–218, Seattle, WA, USA, November 2006. USENIX Association.

[15] Shuyi Chen, Kaustubh R. Joshi, Matti A. Hiltunen, William H. Sanders, and Richard D. Schlichting. Link gradients: Predicting the impact of network latency on multitier applications. In *Proceedings of the 28th IEEE International Conference on Computer Communications*, INFOCOM '09, pages 2258–2266, Rio de Janeiro, Brazil, April 2009. IEEE Computer Society.

[16] Parvathi Chundi, Daniel J. Rosenkrantz, and Sekharipuram S. Ravi. Deferred updates and data placement in distributed databases. In *Proceedings of the 12th International Conference on Data Engineering*, ICDE '96, pages 469–476, New Orleans, LA, USA, February 1996. IEEE Computer Society.

[17] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, August 2008.

[18] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '12, pages 251–264, Hollywood, CA, USA, October 2012. USENIX Association.

[19] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: A workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1-2):48–57, September 2010.

[20] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-Store: A scalable data store for transactional multi key access in the cloud. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 163–174, Indianapolis, Indiana, USA, June 2010. ACM.

[21] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with snapshot isolation. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, pages 715–726, Seoul, Korea, September 2006. VLDB Endowment.

[22] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, Stevenson, WA, USA, October 2007. ACM.

[23] Bailu Ding, Lucja Kot, Alan Demers, and Johannes Gehrke. Centiman: Elastic, high performance optimistic concurrency control by watermarking. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 262–275, Kohala Coast, HI, USA, August 2015. ACM.

[24] Philip Dixon. Shopzilla's site redo - you get what you measure. In *Proceedings of the 2009 Velocity Web Performance and Operations Conference*, San Jose, CA, USA, June 2009. O'REILLY.

[25] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124):5–5, August 2004.

[26] International Organization for Standardization. Information processing systems – Database language – SQL. Standard, ISO/IEC 9075, Geneva, Switzerland, June 1987.

[27] Guoqiang Gao, Ruixuan Li, Weijun Xiao, and Zhiyong Xu. Distributed caching strategies in peer-to-peer systems. In *Proceedings of the 13th IEEE International Conference on High Performance Computing and Communications*, HPCC '11, pages 1–8, Banff, AB, Canada, September 2011. IEEE Computer Society.

[28] Lei Gao, Mike Dahlin, Amol Nayate, Jiandan Zheng, and Arun Iyengar. Application specific data replication for edge services. In *Proceedings of the 12th International Conference on World Wide Web*, WWW '03, pages 449–460, Budapest, Hungary, May 2003. ACM.

[29] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas E. Anderson. Scalable consistency in Scatter. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 15–28, Cascais, Portugal, October 2011. ACM.

[30] Lukasz Golab, Marios Hadjieleftheriou, Howard Karloff, and Barna Saha. Distributed data placement to minimize communication costs via graph partitioning. In *Proceedings of the 26th International Conference on Scientific and Statistical Database Management*, SSDBM '14, pages 20:1–20:12, Aalborg, Denmark, June 2014. ACM.

[31] Lukasz Golab, Marios Hadjieleftheriou, Howard J. Karloff, and Barna Saha. Distributed data placement via graph partitioning. *CoRR*, abs/1312.0285, December 2013.

[32] Google. Google cloud platform. https://cloud.google.com.

[33] Google. SPDY: An experimental protocol for a faster web. http://www.chromium.org/spdy/spdy-whitepaper.

[34] Ilya Grigorik (Google). Building faster websites: Crash course on web performance. In *In Proceedings of the 2013 O'REILLY fluent conference: JavaScript & Beyond*, Fluent '13, San Francisco, CA, USA, May 2013. O'REILLY.

[35] James Hamilton. The cost of latency. http://perspectives.mvdirona.com/2009/10/31/TheCostOfLatency.aspx, October 2009.

[36] Kartik Hosanagar and Yong Tan. Cooperative cashing? an economic analysis of document duplication in cooperative web caching. *Information Systems Research*, 23(2):356–375, June 2012.

[37] HP. HP performance center.

[38] IBM. Transparent application scaling with IBM DB2 pureScale. Technical report, IBM, October 2009.

[39] IBM Analytics. From business insight to business action - combining the power of IBM predictive analytics and IBM decision optimization. Technical report, IBM, April 2015.

[40] IBM Redbooks. *Websphere Edge Server New Features and Functions in Version 2*. Vervante, April 2002.

[41] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: Application in VLSI domain. In *Proceedings of the 34th Annual Design Automation Conference*, DAC '97, pages 526–529, Anaheim, CA, USA, June 1997. ACM.

[42] Karypis Lab. hMETIS - Hypergraph & Circuit Partitioning. http://glaros.dtc.umn.edu/gkhome/metis/hmetis/overview, November 1998.

[43] Ron Kohavi, Randal M. Henne, and Dan Sommerfield. Practical guide to controlled experiments on the web: Listen to your customers not to the hippo. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '07, pages 959–967, San Jose, CA, USA, August 2007. ACM.

[44] Ron Kohavi and Roger Longbotham. Online experiments: Lessons learned. *Computer*, 40(9):103–105, September 2007.

[45] Erdinc Korpeoglu, Cetin Sahin, Divyakant Agrawal, Amr El Abbadi, Takeo Hosomi, and Yoshiki Seo. Dragonfly: Cloud assisted peer-to-peer architecture for multipoint media streaming applications. In *Proceedings of the 6th IEEE International Conference on Cloud Computing*, CLOUD '13, pages 269–276, Santa Clara, CA, USA, June 2013. IEEE Computer Society.

[46] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 113–126, Prague, Czech Republic, April 2013. ACM.

[47] Dorothy Kucar, Shawki Areibi, and Anthony Vannelli. Hypergraph partitioning techniques. *Dynamics of Continuous, Discrete and Impulsive Systems*, 11(2-3a):339–367, 2004.

[48] K. Ashwin Kumar, Amol Deshpande, and Samir Khuller. Data placement and replica selection for improving co-location in distributed environments. *CoRR*, abs/1302.4168, February 2013.

[49] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, April 2010.

[50] Tom Leighton. Improving performance on the internet. *Communications of the ACM*, 52(2):44–51, February 2009.

[51] Justin J. Levandoski, David B. Lomet, Mohamed F. Mokbel, and Kevin Zhao. Deuteronomy: Transaction support for cloud data. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*, CIDR '11, pages 123–133, Asilomar, CA, USA, January 2011. CIDR Conference.

[52] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '12, pages 265–278, Hollywood, CA, USA, October 2012. USENIX Association.

[53] Jiexing Li, Jeffrey Naughton, and Rimma V. Nehme. Resource bricolage for parallel database systems. *Proceedings of the VLDB Endowment*, 8(1):25–36, September 2014.

[54] Zhichun Li, Ming Zhang, Zhaosheng Zhu, Yan Chen, Albert G. Greenberg, and Yi-Min Wang. WebProphet: Automating performance prediction for web services. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '10, pages 143–158, San Jose, CA, USA, April 2010. USENIX Association.

[55] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: a memory-efficient, high-performance key-value store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 1–13, Cascais, Portugal, October 2011. ACM.

[56] Chia Feng Lin, Muh-Chyi Leu, Chih-Wei Chang, and Shyan-Ming Yuan. The study and methods for cloud based CDN. In *Proceedings of the 3rd International Conference on Cyber-enabled Distributed Computing and Knowledge Discovery*, CyberC '11, pages 469–475, Beijing, China, October 2011. IEEE Computer Society.

[57] Yi Lin, Bettina Kemme, Marta Patiño-Martínez, and Ricardo Jiménez-Peris. Enhancing edge computing with database replication. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*, SRDS '07, pages 45–54, Beijing, China, October 2007. IEEE Computer Society.

[58] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, Cascais, Portugal, October 2011. ACM.

[59] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '13, pages 313–328, Lombard, IL, USA, April 2013. USENIX Association.

[60] Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. Low-latency multi-datacenter databases using replicated commit. *Proceedings of the VLDB Endowment*, 6(9):661–672, July 2013.

[61] Marissa Mayer. What google knows. In *Proceedings of the 3rd Annual Web 2.0 Summit*, San Francisco, CA, USA, November 2006. O'REILLY.

[62] Hemant K. Mehta, Priyesh Kanungo, and Manohar Chandwani. Distributed database caching for web applications and web services. In *Proceedings of the International Conference & Workshop on Emerging Trends in Technology*, ICWET '11, pages 510–515. ACM, February 2011.

[63] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting more concurrency from distributed transactions. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14, pages 479–494, Broomfield, CO, USA, October 2014. USENIX Association.

[64] Rimma Nehme and Nicolas Bruno. Automated partitioning design in parallel database systems. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 1137–1148, Athens, Greece, June 2011. ACM.

[65] Neustar. Benefits of external load testing: Identify bottlenecks and improve customer experience. Whitepaper, Neustar, 2012. https://www.neustar.biz/resources/whitepapers/benefits-of-external-load-testing.

[66] NSERC Strategic Network. Smart applications on virtual infrastructures. http://www.savinetwork.ca.

[67] Lucas Nussbaum and Olivier Richard. A comparative study of network link emulators. In *Proceedings of the 2009 Spring Simulation Multiconference*, SpringSim '09, pages 85:1–85:8, San Diego, CA, USA, March 2009. Society for Computer Simulation International.

[68] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*, ATEC '99, pages 43–43, Monterey, CA, USA, June 1999. USENIX Association.

[69] Oracle. Berkeley DB Java Edition. http://www.oracle.com/technetwork/database/berkeleydb/berkeley-db-je-ds-066564.pdf.

[70] Oracle. Oracle Real Application Clusters 11g Release 2. Technical report, Oracle, November 2010.

[71] Oracle. Oracle NoSQL Database. Technical report, Oracle, September 2011.

[72] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, January 2010.

[73] Andrea Passarella. A survey on content-centric technologies for the current internet: CDN and P2P solutions. *Computer Communications*, 35(1):1–32, January 2012.

[74] Stacy Patterson, Aaron J. Elmore, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. Serializability, not serial: Concurrency control and availability in multi-datacenter datastores. *Proceedings of the VLDB Endowment*, 5(11):1459–1470, July 2012.

[75] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 61–72, Scottsdale, AZ, USA, May 2012. ACM.

[76] Guillaume Pierre and Maarten van Steen. Globule: A collaborative content delivery network. *IEEE Communications Magazine*, 44(8):127–133, August 2006.

[77] F. Ping, X. Li, C. McConnell, R. Vabbalareddy, and J. H. Hwang. Towards optimal data replication across data centers. In *Proceedings of the 31st International Conference on Distributed Computing Systems Workshops*, ICDCSW '11, pages 66–71, Minneapolis, MN, USA, June 2011. IEEE Computer Society.

[78] Fan Ping, Jeong-Hyon Hwang, XiaoHu Li, Chris McConnell, and Rohini Vabbalareddy. Wide area placement of data replicas for fast and highly available data access. In *Proceedings of the 4th International Workshop on Data Intensive Distributed Computing*, DIDC '11, pages 1–8, San Jose, CA, USA, June 2011. ACM.

[79] Himabindu Pucha and Saumitra M. Das. DynCoDe: An architecture for transparent dynamic content delivery. In *Proceedings of the 12th International World Wide Web Conference*, WWW '03 (Posters), Budapest, Hungary, May 2003. ACM.

[80] Josep M. Pujol, Vijay Erramilli, Georgos Siganos, Xiaoyuan Yang, Nikos Laoutaris, Parminder Chhabra, and Pablo Rodriguez. The little engine(s) that could: Scaling online social networks. In *Proceedings of the 2010 ACM SIGCOMM Conference on Internet Measurement Conference*, SIGCOMM '10, pages 375–386, Melbourne, Australia, November 2010. ACM.

[81] Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. SWORD: Scalable workload-aware data placement for transactional workloads. In *Proceedings of the 16th International*

Conference on Extending Database Technology, EDBT/ICDT '13, pages 430–441, Genoa, Italy, March 2013. ACM.

[82] Ramakrishnan Rajamony and Elmootazbellah (Mootaz) Elnozahy. Measuring client-perceived response time on the WWW. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, USITS '01, pages 16–16, San Francisco, CA, USA, March 2001. USENIX Association.

[83] Srinivasan Suman Ramkumar, Lee Jae Woo, Batni Dhruva L., Schulzrinne Henning G., Jing Gao Haibin Cheng, and Pang-Ning Tan. ActiveCDN: Cloud computing meets content delivery networks. Technical Report CUCS-045-11, Department of Computer Science, Columbia University, January 2011.

[84] Rice University. RUBiS: Rice University Bidding System. http://rubis.ow2.org.

[85] Eric Schurman (Amazon) and Jake Brutlag (Google). The user and business impact of server delays, additional bytes, and HTTP chunking in web search. In *Proceedings of the 2009 Velocity Web Performance and Operations Conference*, San Jose, CA, USA, June 2009. O'REILLY.

[86] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS '11, pages 386–400, Grenoble, France, October 2011. Springer-Verlag.

[87] Artyom Sharov, Alexander Shraer, Arif Merchant, and Murray Stokely. Take me to your leader!: Online optimization of distributed storage configurations. *Proceedings of the VLDB Endowment*, 8(12):1490–1501, August 2015.

[88] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, Cascais, Portugal, October 2011. ACM.

[89] Liwen Sun, Michael J. Franklin, Sanjay Krishnan, and Reynold S. Xin. Fine-grained partitioning for aggressive data skipping. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1115–1126, Snowbird, UT, USA, June 2014. ACM.

[90] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment*, 8(3):245–256, November 2014.

[91] Muhammad Mukarram Bin Tariq, Amgad Zeitoun, Vytautas Valancius, Nick Feamster, and Mostafa H. Ammar. Answering what-if deployment and configuration questions with wise. In *Proceedings of the 2008 ACM SIGCOMM Conference on Data Communication*, SIGCOMM '08, pages 99–110, Seattle, WA, USA, August 2008. ACM.

[92] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems*, PDIS '94, pages 140–149, Austin, TX, USA, September 1994. IEEE Computer Society.

[93] The Apache Software Foundation. Apache HBase. `https://hbase.apache.org/`.

[94] The SQLite Consortium. SQLite. `https://sqlite.org`.

[95] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12, Scottsdale, AZ, USA, May 2012. ACM.

[96] Khai Q. Tran, Jeffrey F. Naughton, Bruhathi Sundarmurthy, and Dimitris Tsirogiannis. JECB: A join-extension, code-based approach to OLTP data partitioning. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 39–50, Snowbird, UT, USA, June 2014. ACM.

[97] Alexandru Turcu, Roberto Palmieri, and Binoy Ravindran. Automated data partitioning for highly scalable and strongly consistent transactions. In *Proceedings of the 7th ACM International Conference on Systems and Storage*, SYSTOR '14, pages 1–11, Haifa, Israel, June 2014. ACM.

[98] Danny De Vleeschauwer and Dave C. Robinson. Optimum caching strategies for a telco CDN. *Bell Labs Technical Journal*, 16(2):115–132, September 2011.

[99] Limin Wang, Vivek S. Pai, and Larry L. Peterson. The effectiveness of request redirection on CDN robustness. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '02, pages 345–360, Boston, MA, USA, December 2002. USENIX Association.

[100] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. Demystifying page load performance with WProf. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '13, pages 473–486, Lombard, IL, USA, April 2013. USENIX Association.

[101] Jianbin Wei and Cheng-Zhong Xu. Measuring client-perceived pageview response time of internet services. *IEEE Transactions on Parallel and Distributed Systems*, 22(5):773–785, May 2011.

[102] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[103] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 292–308, Farmington, PA, USA, November 2013. ACM.

[104] Zhen Ye, Shanping Li, and Xiaozhen Zhou. GCplace: Geo-cloud based correlation aware data replica placement. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 371–376, Coimbra, Portugal, March 2013. ACM.

[105] Chun Yuan, Yu Chen, and Zheng Zhang. Evaluation of edge caching/offloading for dynamic content delivery. *IEEE Transactions on Knowledge and Data Engineering*, 16(11):1411–1423, November 2004.

[106] Victor Zakhary, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. DB-Risk: The game of global database placement. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, SIGMOD '16, pages 2185–2188, San Francisco, CA, USA, June 2016. ACM.

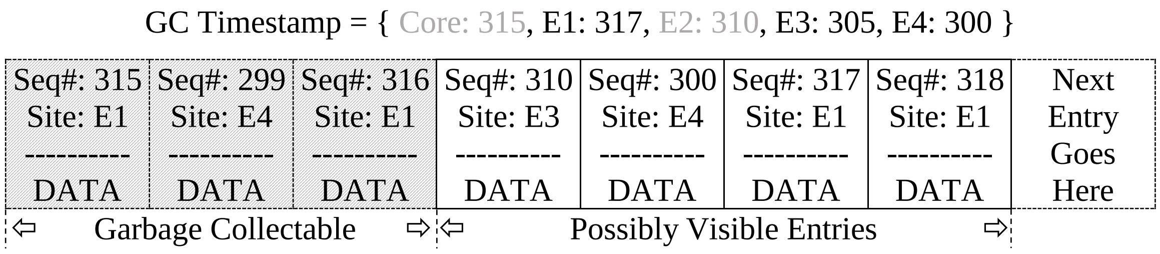
[107] Marek Zawirski, Nuno Preguiça, Sérgio Duarte, Annette Bieniusa, Valter Balegas, and Marc Shapiro. Write fast, read in the past: Causal consistency for client-side applications. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 75–87, Vancouver, BC, Canada, December 2015. ACM.

[108] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 276–291, Farmington, PA, USA, November 2013. ACM.

Appendices

Appendix A: Site Database API

This appendix describes in more detail the functionality that an underlying SpearDB site database needs to provide. A site database needs to provide (through the use of a driver) several simple API operations: READ, WRITE and FLUSH, as well as READALL and MERGE, as described in Table 3.4.



GC Timestamp = { Core: 315, E1: 317, E2: 310, E3: 305, E4: 300 }
Seq#: 315 Site: E1 ---------- DATA
Seq#: 299 Site: E4 ---------- DATA
Seq#: 316 Site: E1 ---------- DATA
Seq#: 310 Site: E3 ---------- DATA
Seq#: 300 Site: E4 ---------- DATA
Seq#: 317 Site: E1 ---------- DATA
Seq#: 318 Site: E1 ---------- DATA
Next Entry Goes Here
Garbage Collectable
Possibly Visible Entries

Figure A.1: SiteDB Entry List Example for some Key

The underlying database needs to be able to store a version list for every key. Figure A.1 provides an example of such a list which may contain 1 or more entries. Each version includes a commit timestamp, which consists of the commit site and a sequence number (for the commit site), in addition to the actual data (i.e. the "value" associated with the key). Thus, database systems that can naturally store multi-version data and can iterate over the versions (e.g. BerkeleyDB, HBase) are a good fit, but any storage engine capable of storing an ordered list is sufficient in this regard.

The driver needs to implement several API operations. The operations required for additional copy installation at an edge are described in Section A.ii. For regular transaction processing, the following API operations are required:

READ(key, vts): This function needs to retrieve the last version in the ordered version list of a given key that has a commit timestamp smaller than or equal the given input vector timestamp parameter, vts. A commit timestamp, ⟨site, seqno⟩, is comparable to a vector timestamp vts, by comparing the commit timestamp's sequence number to the vector timestamp's component corresponding to the commit timestamp's site: i.e. seqno compared to vts[site]. If an entry with a commit timestamp smaller than or equal to vts does not exist in the version list, or if there isn't a version list at all for the given key, NULL is returned - signifying that this site database doesn't have knowledge of this key.

WRITE(set of ⟨key, value⟩ pairs, site, seqno): For each ⟨key, value⟩ pair in the input, this function needs to install a new version entry at the end of the ordered version list of key. The version entry to be installed consists of the value, along with the commit timestamp ⟨site, seqno⟩ that is constructed from the last two input parameters. All these changes need to be made atomically, but there need not be any durability guarantees on the changes in case of failure at this point.

FLUSH(): This function's purpose it to ensure that any changes that have been made by **all** previous WRITE calls become durable in the face of a system failure. If a failure occurs while this function is running there needs to be a guarantee that upon recovery, changes made by each individual WRITE call are either fully restored or not at all in order to preserve system-wide atomicity.

The system provides some additional guarantees regarding the way the READ and WRITE operations may be called. These guarantees are intended to simplify driver development. First, it is guaranteed that there are no concurrent calls to WRITE. Subsequent calls to WRITE append new versions to the ends of ordered version lists using system provided commit timestamps. These commit timestamps are guaranteed to be built in such a way that all resulting ordered version lists will have a splitting point (between elements) with the following property: For any vector timestamp vts that may be used in READ operations, all commit timestamps in entries preceding the splitting point are smaller than or equal to vts and all following entries have commit timestamps greater than vts.

For example, using these guarantees, a driver implementation of the READ function could first retrieves the version list associated with the key. Then, while traversing the list backwards (i.e. from right to left in the list depiction given in Figure A.1), the first entry with a commit timestamp smaller than or equal to vts can be returned without finishing the list traversal. For example, if vts = { Core: 315, E1: 316, E2: 310, E3: 310, E4: 300 } for the entry list in Figure A.1, the entry with sequence number 300 and site E4 will be retrieved after traversing the last 3 elements.

The WRITE function can take advantage of the "no concurrency" guarantee in order to implement its atomicity requirements at the driver level, atop an underlying database system that doesn't support atomic multi-write operations. A simple example of this would be to use a database-backed WRITE-counter in the driver. All version list entries appended by the WRITE would be tagged with the value of this counter plus one. Then READ operations would be programmed to ignore any entries tagged with a value smaller than the current counter. The counter itself would only be incremented at the end of the WRITE operation, thus making all modifications atomically visible to READs.

The FLUSH operation has been separated from WRITE in order to allow for batch disc flushes (an expensive long running operation) in order to increase concurrency. Some database systems like BerkeleyDB [68] explicitly allow disc flush operations to be invoked separately from database changes that are cached in memory. However, if WRITE operations for the site database already guarantee a satisfactory level of durability, then FLUSH may be implemented as a no-op. Furthermore, SpearDB's global durability guarantees are essentially the same as those provided by this function coupled with the underlying database. For example, a volatile in-memory implementation with a no-op FLUSH function provides no durability guarantees (but is a valid implementation fully usable by SpearDB).

A.i Garbage Collection

The driver for the underlying database also needs to take care of garbage collection. In order to do this, SpearDB provides the driver with access to a garbage collectible timestamp (GC Timestamp in Figure A.1) for the current site as well as two flags: garbage_collect_on_read and garbage_collect_on_write. Based on the value of these two flags, garbage collection should happen when reading, writing, or both. Specifically, garbage collection would only happen for the entry list(s) currently being operated on by READ or WRITE operations.

The GC Timestamp value is computed based on the current site's timestamp and the current running transactions' timestamps. Its value indicates the lowest sequence number (for each site) that a currently running or future transaction (at the local site) can have. For the edges, only the locally running transactions' timestamps need to be considered since READ operations will never come in from other sites. For the core, all global running transactions' timestamps are relevant so the edges' GC Timestamps are periodically sent to the core to update the core GC Timestamp. Out of all entries with a commit timestamp lower than the GC Timestamp, only the last one (in traversal order) needs to be kept. All others may be garbage collected (i.e. removed from the entry list) as there aren't any running transactions left in the system with database snapshots older than the values kept.

A.ii New Copy Operations

This section describes the two API operations required for the process of installing a new secondary copy which is described in Section 4.5.1.

READALL(key): This function returns the entire version list associated with the key given as input. This will then be passed on as the history parameter to a MERGE operation running on a different database.

MERGE(key, history): This function merges the local database's version list associated with the given key with a foreign version history provided as the second parameter. Both the history parameter and the local version list for key are lists of triples of the form $\langle$value, site, seqno$\rangle$. The merge operation require that duplicate entries that appear in both lists be removed from the history list first. Finally, the merge ordering is also strict: the remainder of the history list (after duplicate removal) is *prepended* to the local version list for key.

The system guarantees that no two MERGE operations on the same key will be requested concurrently on the same database. However, calls to the WRITE function that include modifications to the same key are allowed to occur concurrently and the driver must ensure that both operations execute in parallel. This requirement could easily be relaxed by introducing additional locking in SpearDB before calls to WRITE and MERGE operations. However, WRITE appends data on one end of the list, while MERGE prepends it on the other. Furthermore, the changes these operations are required to make may be interleaved (i.e. they needn't happen in a strict serial order). For this reason it is more efficient to make these operations thread safe at the driver level.

Additionally, MERGE may also interact with the garbage collection mechanism described in Section A.i. This may happen if several WRITEs that modify the key happen before the MERGE operation is called. If one or more of the WRITEs trigger garbage collection, old entries at the beginning of the base list will be removed. This will cause a gap in the version history to appear once MERGE prepends the history list to the base. In this case, however, the gap doesn't present a problem, since the entries prepended are even older than the garbage collected ones. Thus they will never be seen by any running transaction and they will simply be removed on the next garbage collection cycle.

Appendix B: SpearDB PSI Guarantee

This appendix shows that the SpearDB pseudocode presented in Chapter 4 observes the three PSI properties introduced in Section 3.3. Theorem B.1 in Section B.i claims that write-write conflicts are disallowed in SpearDB. Theorem B.3 in Section B.ii states that the SpearDB pseudocode observes the causality property of PSI. Finally, Section B.iii describes SpearDB's compliance with the transaction-consistent database snapshot property.

The proofs in this appendix only account for regular transactional processing. Data movement operations are omitted for the proofs' simplicity. The interaction between data movement operations and regular transactional processing operations is described in Section 4.5.1, along with the reasons why the former don't affect the correctness of the latter.

B.i No Write-Write Conflicts

Theorem B.1 states that concurrent transactions cannot have write-write conflicts. More specifically it makes the equivalent claim that two committed transactions that have updated a common key cannot be concurrent. Figure B.1 is used throughout the proof of this theorem. It contains an equivalent pseudocode version of the relevant parts of the transaction commit protocol extracted from Figures 4.6 and 4.9. Only the checks relevant to this proof and pertaining to a single tuple key have been left in this version of the pseudocode as well as only the branches that lead to successful transaction committal.

Theorem B.1 (No Write-Write Conflicts) *Let tx_1 and tx_2 be any pair of committed transactions whose writesets contain a common key. Then $tx_1.startVTS[tx_2.commitSite] \geq tx_2.seqNo$* ***or*** *$tx_2.startVTS[tx_1.commitSite] \geq tx_1.seqNo$.*

Proof by contradiction:
Suppose that tx_1 and tx_2 both update tuple key, and $\mathrm{tx}_1.\mathrm{startVTS}[\mathrm{tx}_2.\mathrm{commitSite}] < \mathrm{tx}_2.\mathrm{seqNo}$ and $\mathrm{tx}_2.\mathrm{startVTS}[\mathrm{tx}_1.\mathrm{commitSite}] < \mathrm{tx}_1.\mathrm{seqNo}$.

There are 3 cases to consider based on the first commit of the transactions (i.e. not including propagation to other sites):

<u>**Case 1:**</u> both tx_1 and tx_2 are committed using `singleSiteCommit`

In this case, each transaction must be committing at the site that has the primary copies of all the tuples in their writeset. Since key belongs to both transactions' writesets it follows that both transactions are committed at the same site S (which contains the primary copy of the tuple key) $\Rightarrow \mathrm{tx}_1.\mathrm{commitSite} = \mathrm{tx}_2.\mathrm{commitSite} = \mathrm{CORE}.\mathrm{pCopy}[\mathrm{key}] = \mathrm{S}$.

```
function singleSiteCommit(tx)
  if key not locked at S # check for locks (used by 2PC)
  and key unmodified since tx.startVTS  # enforce no concurrent updates
    tx.seqNo ← ++S.seqNo # assign commit version
    WRITE(⟨⟨key, ?⟩⟩, S, tx.seqNo) # store new versions of keys to database

function twoPhaseCommit(tx)
  remote async prepare(tx) @ CORE.pCopy[key]
  wait for vote from CORE.pCopy[key]
  if vote is YES
    tx.seqNo ← ++CORE.seqNo
    WRITE(⟨⟨key, ?⟩⟩, CORE, tx.seqNo) # store to CORE's database
  ...
  # eventually return to twoPhaseCommit's calling function: coreCommit
  # and execute the following:
  remote async installTrans(tx) @ CORE.pCopy[key]

function installTrans(tx)
  WRITE(⟨⟨key, ?⟩⟩, tx.commitSite, tx.seqNo) # store to site database
  release lock on key held by tx at S

function prepare(tx)
  if key not locked at S # check for locks (used by 2PC)
  and key unmodified since tx.startVTS  # enforce no concurrent updates
    lock key for tx
    return YES
```

Figure B.1: Equivalent pseudocode portions of all transaction commit branches at a site S or the CORE for a single key transaction - code fragments are extracted from Figures 4.6, 4.7, 4.8 and 4.9.

Without loss of generality, assume that tx_1 goes through `singleSiteCommit`'s critical section first. Inside this critical section, a new sequence number is generated at site S on line 4 of Figure B.1. Then, a new version of key is installed in the database on line 5 which essentially modifies key. Since both transactions eventually commit, they both must successfully complete the "no concurrent update" check on line 3. However the only way for tx_2 to pass this test when it enters the critical section after tx_1 is if $tx_2.\text{startVTS}$ includes tx_1's modification of key with sequence number $tx_1.\text{seqNo}$, i.e. $tx_2.\text{startVTS}[S] \geq tx_1.\text{seqNo}$, **a contradiction**.

Case 2: one transaction is committed using `singleSiteCommit` and the other `twoPhaseCommit`

Without loss of generality, assume that tx_1 is committed using `singleSiteCommit` and tx_2 is committed using `twoPhaseCommit`. Since key belongs to both transactions' writesets it follows that tx_1 is committed at the site S that contains the primary copy of the tuple key. While tx_2 is committed at the core, but the prepare stage of two phase commit must obtain a lock from the same site S $\Rightarrow tx_1.\text{commitSite} = \text{CORE}.\text{pCopy}[\text{key}] = S$.

Thus, both transactions have to go through a critical section at site S. As seen in Figure B.1, tx_1's critical section is in the `singleSiteCommit` function, while tx_2's is in the `prepare` function. Based on the order in which these transactions enter their respective critical sections, two additional sub-cases are distinguished:

Case 2a: tx_1 executes its critical section at site S before tx_2

This case is the same situation as the one described in **Case 1**: tx_1 modifies key in its critical section first, and then tx_2, in order to eventually commit must pass the "no concurrent updates" check on line 24 which requires that $tx_2.\text{startVTS}$ includes tx_1's modification of key with sequence number $tx_1.\text{seqNo}$, i.e. $tx_2.\text{startVTS}[S] \geq tx_1.\text{seqNo}$, **a contradiction**.

Case 2b: tx_2 executes its critical section at site S before tx_1

In this case, after tx_2 executes the critical section inside the `prepare` function, key is locked at site S as a result of the operation on line 25. If tx_1 where to enter its critical section at this point, it would fail its lock check on line 2 and abort. However, since tx_1 has to continue on to successfully commit, the only time it may pass the lock check is after key is unlocked. This happens on line 20 only after remote propagation of tx_2 and after line 19 which installs a new version of key in S' database. Thus, tx_1 can only (successfully) enter its critical section after tx_2's modification of key with sequence number $tx_2.\text{seqNo}$. This case is then similar to **Case 1**, since, in order to eventually commit, tx_1 must also pass the "no concurrent updates" check on line 3 which requires that $tx_1.\text{startVTS}$ includes tx_2's modification of key with sequence number $tx_2.\text{seqNo}$, i.e. $tx_1.\text{startVTS}[tx_2.\text{commitSite}] \geq tx_2.\text{seqNo}$, **a contradiction**.

Case 3: both tx_1 and tx_2 are committed using `twoPhaseCommit`

Since key belongs to both transactions' writesets it follows that both transactions need to obtain a lock from the site S that contains the primary copy of the tuple key for the prepare stage of two phase commit $\Rightarrow \text{CORE}.\text{pCopy}[\text{key}] = S$.

Without loss of generality, assume that tx_1 goes through `prepare`'s critical section at site S first. This is now similar to the situation described in **Case 2b**. After tx_1 executes the critical section inside the `prepare` function, key is locked at site S as a result of the operation on line 25. If tx_2 where to enter its critical section at this point, it would fail its lock check on line 23 and `prepare` would not vote YES. However, since tx_2 needs to obtain a YES vote in order to continue on to successfully commit, the only time it may pass the lock check is after key is unlocked. This happens on line 20 only after remote propagation of tx_1 and after line 19 which installs a new version of key in S' database. Thus, tx_2 can only (successfully) enter its critical section after tx_1's modification of key with sequence number $\mathrm{tx}_1.\mathrm{seqNo}$. This case is then similar to **Case 1**, since, in order to eventually commit, tx_2 must also pass the "no concurrent updates" check on line 24 which requires that $\mathrm{tx}_2.\mathrm{startVTS}$ includes tx_1's modification of key with sequence number $\mathrm{tx}_1.\mathrm{seqNo}$, i.e. $\mathrm{tx}_2.\mathrm{startVTS}[\mathrm{tx}_1.\mathrm{commitSite}] \geq \mathrm{tx}_1.\mathrm{seqNo}$, **a contradiction**.

Q.E.D.

B.ii Causality

Lemma B.2 states that for any two distinct committed transactions (tx_i and tx_j) that both update a common tuple (key), tx_i and its updates are installed in site S' database before tx_j if and only if tx_i sees tx_j.

Lemma B.2 (Ordered History) $\forall$ *key, S,* tx_i, tx_j: $tx_i \neq tx_j$ *and*
$(\exists\ \langle key,\ val_i\rangle \in tx_i.updates\text{: } S.database[key][i] = \langle val_i,\ tx_i.commitSite,\ tx_i.seqNo\rangle)$ *and*
$(\exists\ \langle key,\ val_j\rangle \in tx_j.updates\text{: } S.database[key][j] = \langle val_j,\ tx_j.commitSite,\ tx_j.seqNo\rangle)$
$\Rightarrow i < j \iff tx_j.startVTS[tx_i.commitSite] \geq tx_i.seqNo$

In this notation, S.database[key] is an *ordered* list of versions for the tuple represented by key. A second index is thus used to refer to the position within this ordering.

Proof of $i < j \Leftarrow tx_j.startVTS[tx_i.commitSite] \geq tx_i.seqNo$:

Suppose $\exists$ key, S, tx_i, tx_j: $\mathrm{tx}_i \neq \mathrm{tx}_j$ and
$(\exists\ \langle \mathrm{key},\ \mathrm{val}_i\rangle \in \mathrm{tx}_i.\mathrm{updates}\text{: } \mathrm{S.database[key][i]} = \langle \mathrm{val}_i,\ \mathrm{tx}_i.\mathrm{commitSite},\ \mathrm{tx}_i.\mathrm{seqNo}\rangle)$ and
$(\exists\ \langle \mathrm{key},\ \mathrm{val}_j\rangle \in \mathrm{tx}_j.\mathrm{updates}\text{: } \mathrm{S.database[key][j]} = \langle \mathrm{val}_j,\ \mathrm{tx}_j.\mathrm{commitSite},\ \mathrm{tx}_j.\mathrm{seqNo}\rangle)$ and
$\mathrm{tx}_j.\mathrm{startVTS}[\mathrm{tx}_i.\mathrm{commitSite}] \geq \mathrm{tx}_i.\mathrm{seqNo}$ and $i \geq j$

Regardless of the method by which tx_x came to be committed at site S (`singleSiteCommit`, `installTrans` or `twoPhaseCommit` as seen in Figure B.2), the following equivalent commands (commit sequence) would have been executed in order:

```
1   wait until S.commitVTS ≥ tx_x.startVTS
2   S.database[key].append(⟨val_x, tx_x.commitSite, tx_x.seqNo⟩)
3   ...
4   S.commitVTS[tx_x.commitSite] ← tx_x.seqNo
```

Whichever transaction (tx_i or tx_j) executes line 2 first is going to be installed first in site S' database. Assume, without loss of generality, that tx_j will attempt to execute its commit sequence before tx_i. If tx_j were to advance past the wait condition on line 1 the following must be true:
$\text{S.commitVTS} \geq \text{tx}_j\text{.startVTS} \Rightarrow \text{S.commitVTS}[\text{tx}_i\text{.startSite}] \geq \text{tx}_j\text{.startVTS}[\text{tx}_i\text{.startSite}]$

Furthermore, from the original supposition that tx_j sees tx_i, the following must also be true:
$\text{tx}_j\text{.startVTS}[\text{tx}_i\text{.startSite}] \geq \text{tx}_i\text{.seqNo} \Rightarrow \text{S.commitVTS}[\text{tx}_i\text{.startSite}] \geq \text{tx}_i\text{.seqNo}$

Which, if true, implies that tx_i would have finished executing its commit sequence at the end of which (line 4) $\text{S.commitVTS}[\text{tx}_i\text{.startSite}]$ is incremented to $\text{tx}_i\text{.seqNo}$. Which also means that tx_i would have completed its database write on line 2. Thus, before tx_j may advance past the wait condition on line 1, tx_i must execute its database write on line 2. This implies $i < j$ which **contradicts** the original supposition.

Q.E.D.

Proof of $i < j \Rightarrow tx_j.startVTS[tx_i.commitSite] \geq tx_i.seqNo$:

Suppose $\exists$ key, S, tx_i, tx_j: $\text{tx}_i \neq \text{tx}_j$ and
$(\exists \langle \text{key}, \text{val}_i \rangle \in \text{tx}_i\text{.updates}: \text{S.database}[\text{key}][\text{i}] = \langle \text{val}_i, \text{tx}_i\text{.commitSite}, \text{tx}_i\text{.seqNo} \rangle)$ and
$(\exists \langle \text{key}, \text{val}_j \rangle \in \text{tx}_j\text{.updates}: \text{S.database}[\text{key}][\text{j}] = \langle \text{val}_j, \text{tx}_j\text{.commitSite}, \text{tx}_j\text{.seqNo} \rangle)$ and
$\text{tx}_j\text{.startVTS}[\text{tx}_i\text{.commitSite}] < \text{tx}_i\text{.seqNo}$ and $i < j$

The following shorthand notation of a transaction's writeset will be used:
$\text{tx.writeset} = \{\text{key}: \langle \text{key}, ?, ? \rangle \in \text{tx.updates}\}$

Since key $\in \text{tx}_i\text{.writeset}$ and key $\in \text{tx}_j\text{.writeset} \Rightarrow \text{tx}_i\text{.writeset} \cap \text{tx}_j\text{.writeset} = \varnothing \Rightarrow$
$\xRightarrow{\text{from Theorem B.1}} \text{tx}_i\text{.startVTS}[\text{tx}_j\text{.startSite}] \geq \text{tx}_j\text{.seqno}$ **or** $\text{tx}_j\text{.startVTS}[\text{tx}_i\text{.startSite}] \geq \text{tx}_i\text{.seqno}$

The latter would contradict the assumption, so further assume that only the former is true:
$\text{tx}_i\text{.startVTS}[\text{tx}_j\text{.startSite}] \geq \text{tx}_j\text{.seqno} \xRightarrow{\text{from the converse proof}} j < i$, **a contradiction**.

Q.E.D.

Theorem B.3 claims that if a transaction is committed at a site, all the transactions it saw (i.e. is causally dependent on) are also committed at that site. Coupled with Lemma B.2, the PSI causality guarantee is also true: i.e. any transaction is committed at all sites *after* the transactions it causally depends on.

Theorem B.3 (Causality) *$\forall$ S, tx_1, tx_2: S.commitVTS[tx_2.commitSite] $\geq$ tx_2.seqNo **and** tx_2.startVTS[tx_1.commitSite] $\geq$ tx_1.seqNo $\Rightarrow$ S.commitVTS[tx_1.commitSite] $\geq$ tx_1.seqNo*

Proof by contradiction: Suppose $\exists$ S, tx_1, tx_2: S.commitVTS[tx_2.commitSite] $\geq$ tx_2.seqNo **and** tx_2.startVTS[tx_1.commitSite] $\geq$ tx_1.seqNo **and** S.commitVTS[tx_1.commitSite] $<$ tx_1.seqNo

Regardless of the method by which a transaction tx has come to be committed at site S (`singleSiteCommit`, `installTrans` or `twoPhaseCommit` as seen in Figure B.2), the following equivalent commands (commit sequence) would have been executed in order:

```
1  wait until S.commitVTS ≥ tx.startVTS
2  ...
3  S.commitVTS[tx.commitSite] ← tx.seqNo
```

Line 3 from the commit sequence (regardless of the committing function) is the only means by which `S.commitVTS` is updated in SpearDB's pseudocode. Because of this, if tx_2 is committed at site S it follows that tx_2 has passed its causality check on line 1. This means that any transaction that tx_2 has seen (which includes tx_1) is also committed at S:
S.commitVTS[tx_2.commitSite] $\geq$ tx_2.seqNo $\xrightarrow{\textit{line } 3 \rightarrow \textit{line } 1}$ S.commitVTS $\geq$ tx_2.startVTS $\Rightarrow$
$\Rightarrow$ S.commitVTS[tx_1.commitSite] $\geq$ tx_2.startVTS[tx_1.commitSite] $\geq$ tx_1.seqNo,
which is **a contradiction**.

Q.E.D.

B.iii Transaction-Consistent Database Snapshots

Lemma B.4 states that all the updates made by a transaction that's committed at a site S, pertaining to any locally replicated tuple at S are installed in site S's database.

Lemma B.4 (No History Gaps) *$\forall$ key, val, tx, S: S.commitVTS[tx.startSite] $\geq$ tx.seqNo and key $\in$ S.lCopies and $\langle$key, val$\rangle$ $\in$ tx.updates $\Rightarrow$ $\langle$val, tx.commitSite, tx.seqNo$\rangle$ $\in$ S.database[key]*

Proof: Suppose $\exists$ key, val, tx, S: S.commitVTS[tx.startSite] $\geq$ tx.seqNo and key $\in$ S.lCopies and $\langle$key, val$\rangle$ $\in$ tx.updates and $\langle$val, tx.commitSite, tx.seqNo$\rangle$ $\notin$ S.database[key]

Every possible branch of the SpearDB code that results in a site's vector timestamp being updated is presented in Figure B.2. All vector timestamp updates happen by incrementing a single component of the vector by 1. This is achieved by first `wait`-ing for the vector component to reach a certain value (lines 4, 14 or 25 from Figure B.2) before incrementing it (lines 9, 19 or 30

```
function singleSiteCommit(tx)
  ...
  wait until S.commitVTS ≥ tx.startVTS
         and S.commitVTS[S] = tx.seqNo - 1
  WRITE(tx.updates, S, tx.seqNo) # store new versions of keys to database
  # the WRITE operation is equivalent to appending ⟨val, S, tx.seqNo⟩
  # to S.database[key], ∀ ⟨key, val⟩ ∈ tx.updates
  ...
  S.commitVTS[S] ← tx.seqNo
  ...

function installTrans(tx)
  wait until S.commitVTS ≥ tx.startVTS
         and S.commitVTS[tx.commitSite] = tx.seqNo - 1
  localUpdates ← {⟨key, val⟩ ∈ tx.updates: key ∈ S.lCopies}
  WRITE(localUpdates, tx.commitSite, tx.seqNo) # store to site database
  # the WRITE is equivalent to appending ⟨val, tx.commitSite, tx.seqNo⟩
  # to S.database[key], ∀ ⟨key, val⟩ ∈ localUpdates

  ...

function twoPhaseCommit(tx)
  ...
  wait until CORE.commitVTS ≥ tx.startVTS
         and CORE.commitVTS[CORE] = tx.seqNo - 1
  WRITE(tx.updates, CORE, tx.seqNo) # store to CORE's database
  # the WRITE operation is equivalent to appending ⟨val, CORE, tx.seqNo⟩
  # to CORE.database[key], ∀ ⟨key, val⟩ ∈ tx.updates
  ...
  CORE.commitVTS[CORE] ← tx.seqNo
  ...
```

Figure B.2: Pseudocode portions of all branches that can lead to a site S or the CORE's commitVTS to be updated - code fragments are extracted from Figures 4.6, 4.7 and 4.9.

from Figure B.2). As such, if S.commitVTS[tx.startSite] ≥ tx.seqNo, it must mean that one of lines 9, 19 or 30 from Figure B.2 has been executed - depending on the method by which the transaction tx came to be committed at site S. Which further implies that the preceding write operation (at lines 5, 16 or 26 from Figure B.2) has also occurred.

In the cases of tx being committed at site S by `singleSiteCommit` or `twoPhaseCommit` (for the latter, S = `CORE`), the write operation having executed, implies that all of tx's updates have been installed in site S's database. Furthermore, in both these cases, the commit site of tx is S. These two implications together contradict the original supposition as shown in (B.5).

$$\begin{aligned}&\langle \mathrm{val}, \mathrm{S}, \mathrm{tx}.\mathrm{seqNo}\rangle \in \mathrm{S}.\mathrm{database}[\mathrm{key}], \forall \langle \mathrm{key}, \mathrm{val}\rangle \in \mathrm{tx}.\mathrm{updates}\\ &\mathrm{tx}.\mathrm{commitSite} = \mathrm{S} \\ &\text{This contradicts the supposition that } \langle \mathrm{val}, \mathrm{tx}.\mathrm{commitSite}, \mathrm{tx}.\mathrm{seqNo}\rangle \notin \mathrm{S}.\mathrm{database}[\mathrm{key}].\end{aligned} \tag{B.5}$$

For the last case of tx being committed at site S by propagation (i.e. `installTrans`), the write operation having executed, implies that all of tx's local updates have been installed in site S's database. Local updates in this case means all updates to keys that have a secondary copy at site S as explicitly defined on line 15 of Figure B.2. This again contradicts the original supposition as shown in (B.6).

$$\begin{aligned}&\langle \mathrm{val}, \mathrm{tx}.\mathrm{commitSite}, \mathrm{tx}.\mathrm{seqNo}\rangle \in \mathrm{S}.\mathrm{database}[\mathrm{key}], \forall \langle \mathrm{key}, \mathrm{val}\rangle \in \mathit{localUpdates}\\ &\mathrm{localUpdates} \leftarrow \{\langle \mathrm{key}, \mathrm{val}\rangle \in \mathrm{tx}.\mathrm{updates} : \mathrm{key} \in \mathrm{S}.\mathrm{lCopies}\} \\ &\text{This contradicts the supposition that } \langle \mathrm{val}, \mathrm{tx}.\mathrm{commitSite}, \mathrm{tx}.\mathrm{seqNo}\rangle \notin \mathrm{S}.\mathrm{database}[\mathrm{key}].\end{aligned} \tag{B.6}$$

Q.E.D.

Theorem B.7 (Transaction-Consistent Database Snapshots) *Given a site S where a tuple key is locally replicated, a* READ *request directed at site S' database with a vector timestamp VTS that's older than or equal to site S' vector timestamp S.commitVTS must return the* latest *value val of key that has been installed at S with a version number visible to VTS.*

Proof: Any transaction tx that is visible at vector timestamp VTS is by extension also committed at site S since S.commitVTS ≥ VTS. Furthermore, since key is locally replicated, Lemma B.4 guarantees that all of tx's updates (pertaining to key) are installed in S's database. Theorem B.3 then guarantees that all of these transactions are committed in causal ordering at site S and Lemma B.2 further guarantees that these transaction's updates are installed in the same order in S.database.

However, S.database[key] might still contain entries of the form: $\langle$val, site, seqNo$\rangle$, where S.commitVTS[site] $\geq$ seqNo $>$ VTS[site]; that are visible to S.commitVTS, but not visible to VTS. The purpose of the READ method described in Appendix A is specifically to use a commit-ordered list and filter out the latest entry visible at timestamp VTS.

Q.E.D.

```
1 function GET(tx, key)
2   if S.lCopies[key] = PRIMARY or S.lCopies[key] = SECONDARY
3     # S has a copy of key that isn't PARTIAL
4     val ← READ(key, tx.startVTS)
5   else
6     val ← remote READ(key, tx.startVTS) @ CORE
7   apply updates from tx.updates to val
8   return val
```

Figure B.3: GET at site S - reproduced from Figure 4.3

The GET operation that is shown again in Figure B.3 always calls READ at either the transaction's start site or the core. If the transaction's start site is used, the check on line 2 ensures the key is locally replicated. Furthermore, tx.startVTS has been assigned as at the start of the transaction to be S.commitVTS, where S is tx's start site. As such, the READ request timestamp is older than the site's timestamp: i.e. S.commitVTS $\geq$ tx.startVTS. Thus all input requirements of Theorem B.7 are met for this call of READ.

If the core is used for reading, all tuples (including key) are locally replicated as a the core maintains a full database replica. Furthermore, there can't exist any updates to key that are installed at the transaction's start site S, but not at the core at the time core reading is invoked. If that were the case, because of SpearDB's network topology, the transaction having installed such an update would have had to commit using `singleSiteCommit` at site S. That can only happen if key's primary copy is at site S and thus is locally replicated and wouldn't call the core for reading. So, all input requirements of Theorem B.7 are met for this case as well.

Finally, snapshot reads are extended to include any updates that have been made to the running transaction (i.e. before it is committed). This is explicitly implemented on line 7 of Figure B.3 and functions independent of the site databases.

Appendix C: Optimized Greedy Primary Placement

This appendix describes an optimized version of the greedy primary copy placement algorithm introduced in Section 6.3. The original version of the algorithm, described in Figure 6.4, did not include any optimizations in order to remain clearer and easier to understand. The worst case complexity of one iteration step of the original algorithm is $O(n^2)$, where n is the size of the workload.

Figure C.1 describes the optimized version of the algorithm. Several data structures are used to store commonly computed values in the algorithm. First, a reverse lookup (line 2) is built that allows the retrieval of the set of all transactions that have a given tuple in their writeset. This lookup is then used to build another lookup (line 4) that allows $O(1)$ average time retrieval of all transactions whose writesets overlap with a given transaction (including itself). The transaction costs of all transactions under the current placement are also stored in a map (line 6). The cost function here is that defined in equation (6.2). Initializing these data structures only requires two passes over the workload (lines 7 and 11).

The main optimization of the algorithm in Figure C.1 consists of using a min-heap data structure (line 14) to memoize the cost of different placements. Whenever a refinement is tried, consisting of either the coring or edging of one transaction, the resulting placement, its *cost delta*, and the original transaction whose coring/edging has produced the new placement can be stored in the heap, sorted on the cost delta value. The cost delta value refers to the cost difference between the placement after the refinement and the one before it. This may be cheaper to calculate than an entire placement's cost value as it only requires the cost of the transactions affected by the placement modifications. The `getCostDelta` function on line 34 is responsible for computing this cost delta given the placement after the refinement and the transaction that produced it. The cost delta is then the sum of the deltas for all affected transactions which is computed on line 35 with the aid of the transaction cost map data structure which contains all the costs of transactions before the refinement.

The worst case complexity of the `getCostDelta` function is $O(n)$, where n is the size of the workload. This worst case occurs when a transaction being cored or edged has a writeset that overlaps with every other transaction in the workload.

The algorithm treats the first iteration step separately (line 16). This first step consists of trying all possible refinements for the entire workload. The `addRefinements` function on line 28 is responsible for building the edged and cored placements of one transaction and adding them to the heap along with their cost deltas and the transaction used to build them. The `addRefinements` function, which has a worst case complexity of $O(n)$, is called twice for every transaction. Furthermore, two heap insertions, with an $O(\log n)$ complexity are required for every transaction as well. Thus, the worst case complexity of the first step of the algorithm is $O(n^2)$.

At the end of the first step, the top of the heap will represent the best refined placement found.

Algorithm Greedy for solving primary copy placement problem
Input: workload W, network N and an initial placement p
Output: primary copy placement p (after being refined)

```
# Build a lookup of transactions that have a tuple in their writeset
txsAffectedByTuple ← new Map⟨tuple, Set⟨transaction⟩⟩
# And a lookup of transactions with overlapping writesets
txsAffectedByTx ← new Map⟨transaction, Set⟨transaction⟩⟩
# And memoize individual transaction costs in current placement
costOf ← new Map⟨transaction, cost⟩
for each transaction tx in workload W
  costOf[tx] ← cost(tx, N, p)
  for each tuple t in writeset tx.ws
    txsAffectedByTuple[t].add(tx)
for each transaction tx in workload W
  txsAffectedByTx[tx] ← ∪ txsAffectedByTuple[t], ∀ t : tx.ws
# Cost-delta sorted heap of placements and transactions that produced them
refinements ← new MinHeap⟨Tuple⟨cost_delta, placement, transaction⟩⟩

for each transaction tx in workload W # treat first iteration separately
  addRefinements(p, tx)
while refinements.peek()[cost_delta] < 0 # there is a better placement
  newP ← refinements.peek()[placement] # best placement
  for each transaction tx in txsAffectedByTx[refinements.peek()[transaction]]
    refinements.delete(⟨?, ?, tx⟩) # remove all affected items
    costOf[tx] ← cost(tx, N, newP) # recalculate affected tx costs
  for each transaction tx in txsAffectedByTx[refinements.peek()[transaction]]
    addRefinements(newP, tx)
  p ← newP
return p

function addRefinements(p, tx) # p is a placement and tx is a transaction
  e ← p modified so that ∀ t ∈ tx.ws, e[t] ← tx.s  # edged tx placement
  refinements.insert(⟨getCostDelta(e, tx), e, tx⟩)
  c ← p modified so that ∀ t ∈ tx.ws, c[t] ← CORE  # cored tx placement
  refinements.insert(⟨getCostDelta(c, tx), c, tx⟩)

function getCostDelta(newP, txA)
  return ∑(cost(tx, N, newP) - costOf[tx]), ∀ tx ∈ txsAffectedByTx[txA]
```

Figure C.1: Optimized Greedy Primary Placement Algorithm

If its cost delta is negative it means that the placement improves upon the initial placement given. Otherwise, the `while` loop on line 18 is never entered and the initial placement is returned as the result.

The following iterations of the algorithm can then occur while new placements with negative cost deltas keep being found. However, since the heap already contains all refined placements after the first iteration and they are already sorted, it is possible to only update the heap items that were affected by the previous placement change. In each iteration following the first, the top of the heap contains the best placement along with the set of transactions affected. All the placements resulting from these affected transactions need to first be removed from the heap (line 21) and then recalculated and re-added (line 24). Furthermore, the transactions costs of these transactions would have changed and need to be updated in the transactions cost map (line 22).

The heap removal operation on line 21 has a worst case complexity of $O(\log n)$. In the worst case, every transaction in the workload is affected by the placement change. This would result in the same worst case complexity $O(n^2)$ as the first iteration.

The worst case complexity of the optimized algorithm doesn't improve on the original algorithm's worst case complexity. However, the optimization techniques described are expected to lower the runtime of this algorithm in practice when compared to the unoptimized version, especially for workloads with few overlapping writesets. The drawback of these optimizations is the additional space (memory) required to store all these data structures. The largest of these data structures is the heap which materializes two additional placements for every transaction in the workload. However most of these placements are very similar to one another. They only differ from a base placement in the location of the tuples in one transaction writeset. Thus, more compressed data structures may be used to save space if required.