POLYGRAPH: A PLUG-AND-PLAY FRAMEWORK TO QUANTIFY APPLICATION ANOMALIES

by

YAZEED ALABDULKARIM

A Dissertation Presented to the
FACULTY OF THE USC GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(COMPUTER SCIENCE)

DECEMBER 2018

Abstract

This thesis describes the functioning of Polygraph, a new plug-and-play tool created by the author for quantifying application anomalies attributed to system behavior that violates atomicity, isolation, linearizability properties of transactions. Example anomalies include erroneous results due to a software bug or an incorrect implementation, dirty reads when a database management system is configured with a weak consistency setting, and stale data produced by a cache. Polygraph is application-agnostic and scales to detect anomalies and compute freshness confidence in real time. It consists of authoring, monitoring, and validation components. An experimentalist uses the Polygraph authoring tool to generate code snippets to embed in application software. These code snippets generate log records capturing conceptual application transactions at the granularity of entities and their relationships. Polygraph validates application transactions in a scalable manner by dividing the task into sub-tasks that process fragments of log records concurrently. Polygraph's monitoring tool visualizes transactions to help reason about anomalies. Polygraph is extensible, enabling a developer to tailor one or more of its components to be application-specific. For example, this thesis describes an extension of Polygraph's validation component to validate range predicates and simple analytics, such as max and min. The thesis also demonstrates the use of

Polygraph with a variety of benchmarks representing diverse applications including TPC-C, BG, YCSB, SEATS and TATP.

Acknowledgments

First and foremost, I praise God for completing my Ph.D. Many people were part of my doctoral journey, and I would like to thank and recognize them.

I express my greatest gratitude to my advisor and mentor, Professor Shahram Ghandeharizadeh, for his guidance and dedication to his students. I have learned a lot from him personally and academically. I thank him for his time and continuous encouragement for me to improve and be a better person.

Besides my advisor, I thank the members of my thesis committee: Prof. Nenad Medvidovic, Prof. Chao Wang, Prof. Wyatt Lloyd, Prof. Christopher Gould, and Prof. Francois Bar for their guidance and constructive feedback.

I am grateful for my parents and siblings for their love and support in my whole life. Although they were overseas, they were always supportive and encouraging.

I express my deepest gratitude to the love of my life, my wife Hessa Albanyan, for sharing every moment of this journey. She supported and encouraged me every day in my Ph.D. study. She was the one whom I went to during my lowest moments. She sacrificed her personal and career life to be with me. I cannot thank her enough and I am forever indebted to her. I would also like to thank my daughter, Juju, for being part of my life.

Last but certainly not least, I want to thank my friends and labmates at the Database Laboratory: Marwan Almaymoni, Hieu Nguyen, Haoyu Huang, Abdulaziz Alhadlaq, Abdulmajeed Alameer, and Abdullah Alwabel.

Contents

Abstract ii

Acknowledgments iv

List of Tables viii

List of Figures x

1 Introduction 1
- 1.1 Definition of Anomalies . . . 3
- 1.2 A Case for Polygraph . . . 6
  - 1.2.1 Cache Augmented SQL Systems . . . 6
  - 1.2.2 DBAs Configuring SQL with Weaker Isolation Levels . . . 7
- 1.3 Atomicity and Durability Anomalies . . . 10
  - 1.3.1 Atomicity Anomalies . . . 10
  - 1.3.2 Durability Anomalies . . . 11
- 1.4 Thesis Contributions . . . 12

2 Related Work 14

3 Plug and Play 18
- 3.1 Overview . . . 18
- 3.2 Apache Kafka . . . 23
- 3.3 Polygraph Log Records . . . 24
- 3.4 Code Generation and Partitioning of Log Records . . . 25

4 Quantifying Anomalies 30
- 4.1 Detecting Anomalies . . . 30
  - 4.1.1 Delayed Log Records . . . 35
- 4.2 Computing Entity/Relationship Values . . . 37
- 4.3 Evaluated Benchmarks . . . 38

5 Scalability **41**
5.1 Tree Structured Applications 41
5.1.1 BG 42
5.1.2 TPC-C 43
5.1.3 Kafka Latency 45
5.2 Non-tree Structured Applications 46
5.2.1 Grouping of Log Records 46
5.2.2 Temporal Partitioning 47

6 Extensibility **54**
6.1 Authoring Component 54
6.2 Validation Component 55
6.3 Monitoring Component 56
6.4 Supporting Range Predicates 57
6.4.1 Main Memory Data Structure 57
6.4.2 Data Store 58
6.4.3 Evaluation 59
6.5 Reporting Freshness Confidence 61
6.6 Supporting Consistency 63

7 Overhead of Polygraph **64**
7.1 Overhead on the Application 64
7.2 Memory Overhead of Polygraph Servers 65
7.2.1 BG 66
7.2.2 TPC-C 67

8 Testing Database Applications **69**
8.1 E-Commerce Applications 69
8.1.1 A Financial Application 70
8.2 Cloud Applications 72
8.2.1 Session Store 75
8.2.2 Photo Tagging Application 75
8.2.3 Messaging Application 78

9 Future Work **80**
9.1 Explanation For Anomalies 80
9.2 Polygraph and Blockchain 81

Reference List **82**

List of Tables

1.1 Number of detected isolation anomalies with different database isolation levels of MySQL with TPC-C. 8

1.2 Atomicity anomalies with BG running a single thread and MongoDB. 11

1.3 Durability anomalies with YCSB and MongoDB. 12

2.1 Comparison of studies quantifying anomalies. 17

4.1 Details of studied benchmarks. 38

4.2 The type and size of log records generated by each transaction for the five studied benchmarks. 40

5.1 A mix of interactive social networking actions with BG. Reads constitute 90% of the mix. 43

5.2 Percentage of records in the largest group (τ) with different batch (ρ) and max group (g) sizes. 46

5.3 Validation time for the first and last thread to finish with different numbers of validation threads and time segments. 48

5.4 The load difference between the first and last thread to finish with eight time segments and eight threads. 49

5.5 Scale-up relative to one thread with different numbers of validation threads and time segments. 50
5.6 Approximate time of Java's garbage collector in seconds with 32 time segments and different numbers of threads. 50
5.7 Scale-out relative to one node with different numbers of nodes and time segments. 50
5.8 The number of detected false negative and false positive anomalies increases with a higher number of time segments. 51
5.9 The number of detected, false negative and false positive anomalies when terminating Polygraph as soon as one thread finishes with different numbers of time segments. 52

7.1 An example of computing network overhead of Polygraph in MB/second for BG and TPC-C benchmarks. 65
7.2 Memory overhead of Polygraph with BG and TPC-C. 66
7.3 Size of BG's write actions log record object in bytes. 67
7.4 Size of TPC-C entities and relationship sets in bytes. 67
7.5 Size of TPC-C's write transactions log record object in bytes. . . . 68

8.1 Conceptual representation of three benchmarks. 74

List of Figures

1.1 Isolation anomalies 4

1.2 Linearizability anomalies 5

1.3 Atomicity anomaly. 5

1.4 An anomalous TPC-C payment transaction due to configuring MySQL with a weak isolation level. 9

2.1 Techniques such as Conflict and View serializability are provided with an order of read and write of the data items. Polygraph is provided with the start and commit timestamps of transactions, their referenced entities/relationships, performed action, and each entity/relationship's property value. 15

3.1 Overview of Polygraph. 19

3.2 An example of TPC-C's payment transaction code snippet. 21

3.3 Authoring and deployment steps of Polygraph. 21

3.4 Example of Polygraph log record. 24

3.5 Example of constructed topics and partitions for an application. . . 29

4.1 To identify read transactions that observe anomalous values, a Polygraph thread must consider all write transactions that overlap the read transaction transitively. 33
4.2 Discarded serial schedules to validate one read transaction are discarded for all future read transactions. 33
4.3 An example schedule consisting of two write transactions and one read transaction. 34
4.4 Log records transmitted out of order. 37

5.1 Vertical scalability with BG uniform workload 44
5.2 Vertical scalability with BG skewed workload 44
5.3 Vertical scalability with TPC-C 45
5.4 Partitioning of log records into time segments. 47
5.5 Processing of read and write log records in a time segment. 48

6.1 Software Architecture of Polygraph's Authoring Component. 54
6.2 Software Architecture of Polygraph's Validation Component. 55
6.3 Software Architecture of Polygraph's Monitoring Component. . . . 56
6.4 Range partitioning of a scan log record. 60
6.5 Computed percentage of freshness confidence while varying the elapsed time after write value. 62

8.1 Two threads execute money withdrawal transactions concurrently which results in an anomaly. 71
8.2 Polygraph monitoring tool visualizes concurrent money withdrawal conceptual transactions. 72
8.3 Number of stale reads per second after recovery from a 10 and a 100 second failure. 76

9.1 A fork in the blockchain causes one of the branches to be discarded 81

Chapter 1

Introduction

Two systems that provide the same performance may be different in that one system may not execute transactions serially. Behavior that violates strict[1] serial execution are collectively termed *anomalies*. These anomalies may violate Atomicity, Consistency, Isolation, and/or Durability (ACID) properties of transactions [64]. Several benchmarks have argued the amount of anomalies should be a first class metric when evaluating systems and comparing them with one another [24, 37, 62, 59].

For example, Amazon's DynamoDB supports two kinds of reads: Strongly consistent and eventually consistent [1]. Eventually consistent reads are the default and 50% cheaper than strongly consistent reads. However, they may not reflect the result of recently acknowledged writes due to delayed propagation of the writes to different replicas of data. DynamoDB claims that consistency across all replicas is usually reached within a second. During this time, a read processed using a replica with a stale value is an anomaly.

Anomalies may impact the correctness of an application [48]. The amount of tolerated anomalies is both application- and workload-specific. Some applications are resilient to anomalies, while others can tolerate either no or infrequent anomalies [54, 33, 45, 46, 78]. For some applications, anomalies can be catastrophic. For example, on March 2014, Flexcoin Bitcoin exchange was a victim of an attack

[1]Strict serial execution means a transaction that starts after the commit point of another transaction cannot be reordered before it [58, 29].

exploiting these anomalies, which led to the company's bankruptcy [3, 14, 65, 2, 73]. The attacker was able to make transfers between Flexcoin users until the sending account was overdrawn before balances were updated.

Workload parameters such as inter-arrival time of requests and how they reference data items impact the number of observed anomalies (and whether they are observed at all). To illustrate, if inter-arrival time for read and write requests referencing a data item exceeds DynamoDB's one second threshold then the number of anomalies is zero.

The author developed Polygraph[2] [16], which is a plug-and-play framework to quantify the amount of anomalies produced by a system. It is external to the system and its data storage infrastructure. This conceptual tool is at the granularity of entities and their relationships. An experimentalist plugs Polygraph into an existing application or benchmark by identifying its entity sets, relationship sets, and transactions that manipulate them. In return, Polygraph generates software that extends each transaction to provide its start time, commit time, the set of entities and relationships read or written by that transaction, and the actions (read, update, insert, and delete) performed on an entity/relationship by each transaction.

Unlike techniques that check for Conflict and View Serializability [28, 58], Polygraph is external and is not provided with the precise order of actions by each transaction (see Chapter 2 for details). Hence, Polygraph computes a serial schedule to establish the ***value*** of an entity/relationship read by a transaction and builds a conceptual database to reflect the values of different entities/relationships. If the observed value is not produced by a serial schedule then Polygraph has detected an anomaly.

[2]Visit http://polygraph.usc.edu for a demonstration of the system.

Polygraph must examine the observed values because simply checking for different ways of serializing transactions is not sufficient. With the DynamoDB example, the transaction that reads a data item may start after the transaction that wrote the data item commits. However, DynamoDB may produce a stale read anomaly due to a delay in propagating the write to all replicas. While there is a valid serial schedule because the read and the write did not overlap in time, the read observes a stale value due to the use of DynamoDB's eventually consistent reads.

The author plugged Polygraph into several benchmarks including BG [24, 17] and OLTP-Bench implementation of TPC-C, YCSB, SEATS and TATP [38]. These confirm that an experimentalist may incorporate Polygraph with a benchmark to quantify both the number of anomalies as well as traditional performance metrics such as throughput.

1.1 Definition of Anomalies

Polygraph detects anomalies that violate strict serial execution of transactions [58, 29]. These include isolation [26], linearizability [47], and atomicity anomalies.

Definition 1.1.1. Isolation anomaly is a behavior that violates serial execution of transactions. Examples include dirty read, dirty write, read skew, write skew, lost update, and scoping anomalies [26, 73] (see Figure 1.1).

Definition 1.1.2. Linearizability anomaly is a behavior caused when either a read observes a value that is not produced by the most recently completed write or operations violate the requirement of total order that is consistent with the real-time order (invocation and response times of operations). Each operation

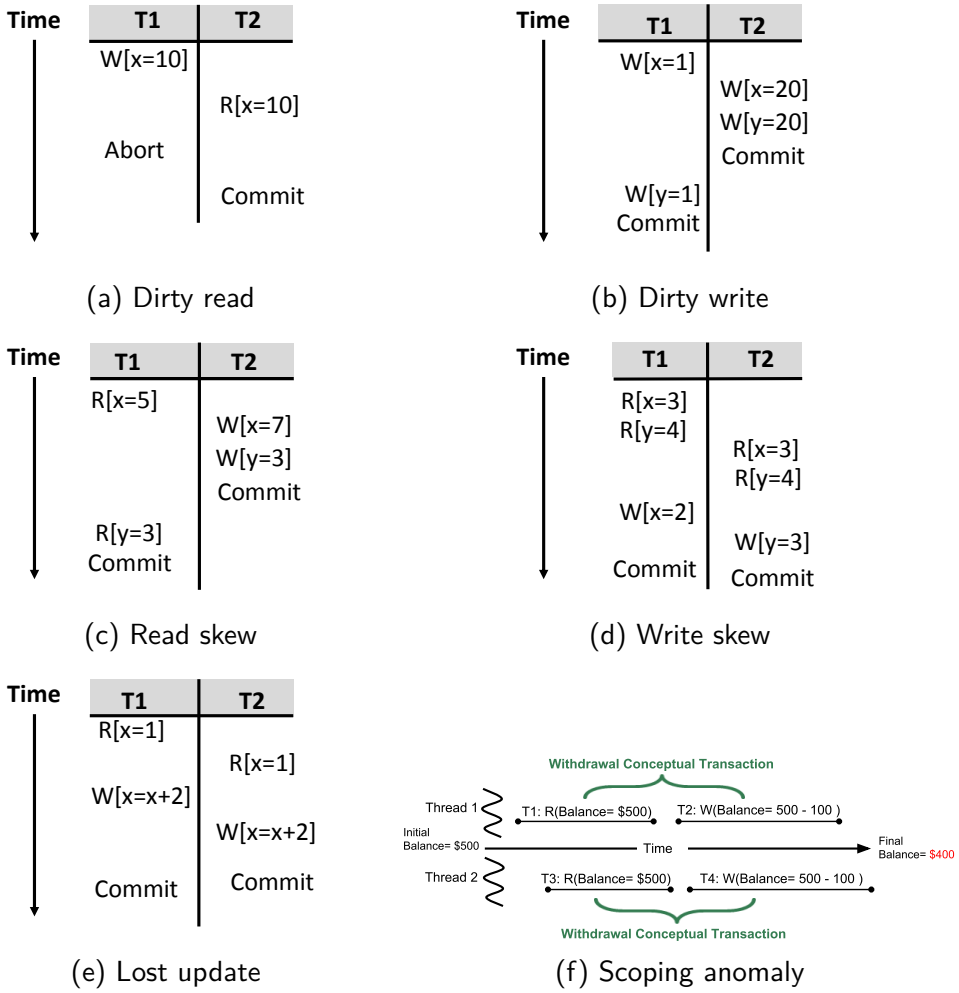
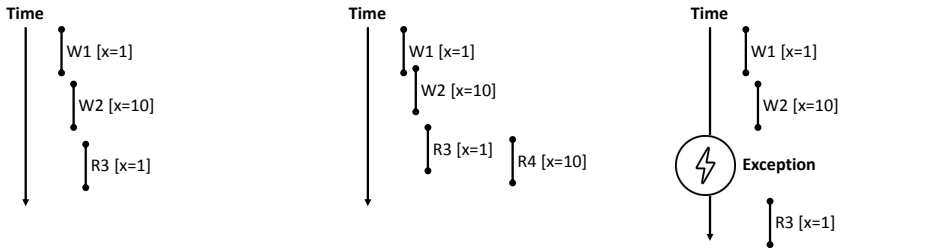


Figure 1.1: Isolation anomalies

is considered as one transaction. Examples include stale read, total order, and durability anomalies (see Figure 1.2).

Isolation anomalies are different than linearizability anomalies. Isolation focuses on serial order of transactions while linearizability focuses on real-time order of operations. For example, lost update anomaly shown in Figure 1.1.e is an isolation anomaly because the execution of transactions do not reflect a serial order. It is not a linearizability anomaly because reads observe the most recently completed writes and operations have a total order that is consistent with real-time.



(a) Stale read (b) Total order (c) Durability anomaly

Figure 1.2: Linearizability anomalies

On the other hand, stale read anomaly shown in Figure 1.2.a is a linearizability anomaly because the read did not observe the value of the latest completed write. It is not an isolation anomaly because operations can be reordered to reflect a serial order.

Definition 1.1.3. Atomicity anomaly occurs when a read observes the value of a transaction that executed partially due to a failure, for example (see Figure 1.3).

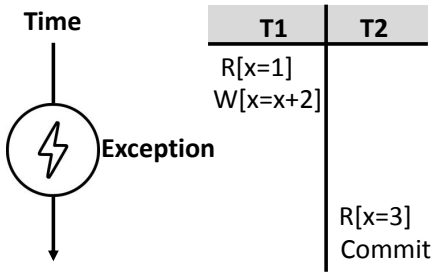


Figure 1.3: Atomicity anomaly.

Atomicity anomaly is different than isolation anomalies because it focuses on partially executed transactions. These are not considered by isolation anomalies because isolation focuses on transactions that either committed or aborted. Atomicity anomaly is different than linearizability anomalies because it focuses on partially executed multi-statement transactions.

1.2 A Case for Polygraph

This section reports on the use of Polygraph to quantify anomalies encountered in today's popular cache augmented architectures and use of weaker isolation levels by database administrators (DBAs).

1.2.1 Cache Augmented SQL Systems

Social networking applications, such as Facebook, extend a data store with a cache manager to enhance performance [56]. A read action looks up its result set in the cache prior to querying the data store. With a cache miss, the read action issues a query to the data store and inserts its result set in the cache for future reference. In the presence of updates, the application may use a write-back, write-through, or write-around policy [69, 57, 4].

Without special precautions, this architecture suffers from race conditions between read actions populating the cache and write actions updating the cache. For example, concurrent read and write actions may execute as follows. First, a write action may invalidate its impacted cached query result sets. Before starting to update the data store, a read action may query the data store and cache the query result set. Consequently, a subsequent read would observe a stale value in the cache.

One can use Polygraph to quantify the amount of anomalies produced by this architecture. To show this, the author plugged Polygraph into BG benchmark and compared MySQL versus MySQL extended with Twemcached [12] (MySQL+Twemcached). BG was run with 10,000 members, 100 threads, and a 90% read workload for 10 minutes (see Table 5.1). The throughput of MySQL increased from 4,781 to 12,795 actions/second with Twemcached, an

improvement of more than 250%. However, the number of Polygraph-detected linearizability anomalies increased from zero with MySQL to 1,014,623 with MySQL+Twemcached. In sum, with MySQL+Twemcached, the throughput increased by more than two folds while 15% of reads observed linearizability anomalies. Deciding whether observing anomalies is tolerable to enhance throughput depends on the application.

1.2.2 DBAs Configuring SQL with Weaker Isolation Levels

A recent survey [60] shows Database Administrators (DBAs) configure SQL systems with weaker isolation levels, such as Read-Committed and Read-Uncommitted, for enhanced performance. These isolation levels may produce a different number of isolation anomalies [26] with different SQL systems. Polygraph can quantify the number of isolation anomalies for an application. The present study shows this with the TPC-C benchmark [38] using MySQL.

MySQL supports Serializable, Repeatable-Read, Read-Committed and Read-Uncommitted isolation levels [10]. In the case of all four, a transaction releases its write locks at its commit/abort point. They differ in terms of how a transaction manages its read locks. With Serializable, a transaction releases its read locks at its commit/abort point. With Repeatable-Read and Read-Committed, MySQL employs a multi-versioning concurrency control mechanism [9]. With Repeatable-Read, all consistent reads within a transaction read the snapshot established by the first such read in that transaction. With Read-Committed, each consistent read within a transaction reads its own snapshot. Finally, with Read-Uncommitted, a read observes the latest value for a data item, which may result in dirty read anomalies [26].

This thesis augments TPC-C benchmark with Polygraph to quantify the number of anomalies when MySQL is configured with the alternative isolation levels. For each, the thesis shows the number of anomalies and new-order transactions per minute (see Table 1.1). These results show the DBA's assumption that lower isolation levels result in a higher throughput to be accurate. When comparing to Serializable, Read-Committed enhanced throughput by 33%. At the same time, Polygraph shows the amount of anomalies increases from 0 with Serializable to 151 with Read-Committed.

Table 1.1: Number of detected isolation anomalies with different database isolation levels of MySQL with TPC-C.

Isolation level	**# of isolation anomalies**	**New-Order transactions/minute**
Serializable	0	8,299
Repeatable-Read	172 (0.014% of reads)	10,021
Read-Committed	151 (0.011% of reads)	11,014
Read-Uncommitted	44 (0.0035% of reads)	10,277

These isolation anomalies represent lost updates [26] (see Figure 1.1.e) due to concurrent read-modify-write payment transactions. For example, two concurrent payment transactions may read the same value for a customer entity and update it, losing the update of one of the two transactions. In addition, Polygraph detected dirty read anomalies (see Figure 1.1.a) with Read-Uncommitted. These represent order-status transactions observing the values of aborted new-order transactions.

Figure 1.4 shows an example of a lost update anomaly with three payment transactions. Transactions 20 and 30 both read the values written by transaction 10. When validating transaction 30, Polygraph flags it as anomalous because there is no serial schedule for the three transactions reflecting their values. Polygraph shows the expected value(s) for the anomalous transaction 30, as it should have observed the values written by transaction 20.

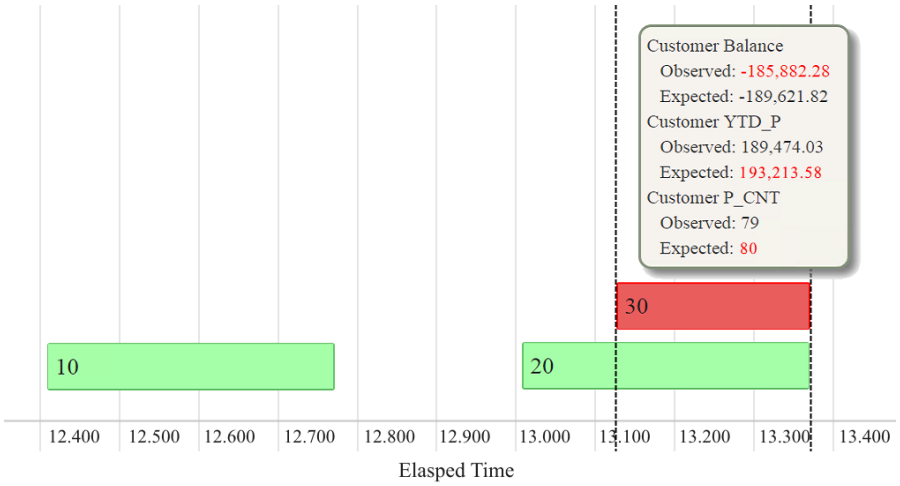


Figure 1.4: An anomalous TPC-C payment transaction due to configuring MySQL with a weak isolation level.

It is surprising that Read-Uncommitted produces significantly ($\sim 300\%$) fewer anomalies when compared with Read-Committed or Repeatable-Read. With Read-Uncommitted, concurrent payment transactions may observe the values written by other uncommitted transactions, reducing the likelihood of multiple transactions reading the same value and causing lost updates. This may result in dirty reads if one of these transactions aborts. On the other hand, with Read-Committed and Repeatable-Read, these concurrent transactions will more likely read the same value, resulting in lost updates. This is because every transaction establishes its own snapshot when reading a data item that does not include uncommitted transactions.

One can run the same experiments with different applications/benchmarks using different SQL systems. A SQL system that uses locking is different from one that uses multi-version concurrency control. Hence, the results may differ.

These results are interesting for two reasons. First, they show Polygraph is able to highlight counter-intuitive results: a weaker isolation level, Read-Uncommitted, produces fewer anomalies than a stronger isolation level, Read-Committed/Repeatable-Read. Second, they highlight the importance of measuring the anomalies observed by the application, rather than relying on the semantics of the isolation levels, as they may behave differently depending on the workload. These results highlight the timeliness and usefulness of Polygraph as a tool.

1.3 Atomicity and Durability Anomalies

This section demonstrates Polygraph's ability to detect atomicity (see Figure 1.3) and durability anomalies, categorized as linearizability anomalies (see Figure 1.2.c). The experiment uses MongoDB and injects failures to produce anomalies. These are fail-stop failures [63] that emulate power faults and system restarts. A failure that kills the mongod process is injected every five seconds. The following two sections demonstrate atomicity and durability anomalies using the BG and YCSB benchmarks, respectively.

1.3.1 Atomicity Anomalies

This experiment uses the BG benchmark with 1000 members and MongoDB configured with Journaling to provide durability [8]. Each BG member is represented as a document in MongoDB with the confirmed and pending friends as lists in the document. Each of Thaw Friendship and Accept Friend Request actions of BG is represented as one transaction in Polygraph. For each action, the manipulation of the confirmed friend lists of the two referenced documents is modeled as one atomic transaction. MongoDB version 3.4.2 was used which provides atomicity

for transactions at the granularity of a single document[3]. Consequently, anomalies occur because this version of MongoDB cannot provide atomicity for each of Thaw Friendship and Accept Friend Request actions which reference two documents. To demonstrate, this experiment runs BG with a single thread and different workloads for one hour (see Table 1.2). A higher percentage of writes increases the number of anomalous documents (third column of Table 1.2) as it increases the likelihood of a failure impacting Thaw Friendship and Accept Friend Request write actions of BG. The number of anomalies is increased with 99% and 99.9% read compared to 90% read because they issue more read actions which increases the frequency of observing anomalies. The number of anomalies decreases with 99.9% read compared to 99% read because it has significantly fewer anomalous documents (19 compared to 121) due to the small percentage of writes (0.01%) with 99.9% read.

Table 1.2: Atomicity anomalies with BG running a single thread and MongoDB.

Workload	**# of anomalies**	**# of anomalous entities (documents)**
90% read	65,652 (0.22% of reads)	153
99% read	364,709 (0.20% of reads)	121
99.9% read	168,502 (0.05% of reads)	19

1.3.2 Durability Anomalies

To demonstrate Polygraph's ability to detect durability anomalies, this experiment uses YCSB and MongoDB configured with the "ACKNOWLEDGE" setting [8]. With this setting, MongoDB acknowledges a write after updating the impacted document in its memory. It flushes dirty documents to disk every 60 seconds. If MongoDB fails after acknowledging a write and prior to flushing it to disk then it results in a durability anomaly by losing the acknowledged write.

[3]MongoDB version 4.0 provides atomicity for transactions that span multiple documents [8].

There are 10,000 YCSB records and each record is represented as a document in MongoDB. This experiment runs Workloads A (50% read) and B (95% read) with both a low (10 threads) and a high (100 threads) system load for 10 minutes.

Columns of Table 1.3 shows a higher number of anomalies with a high system load (100 threads) when compared with a low load (10 threads). A high load increases the number of writes that produce dirty documents lost due to a failure. For example, YCSB Workload B generates four times more write actions with 100 threads when compared with 10 threads, two Million compared with half a Million. Rows of Table 1.3 show the number of anomalies is higher with workload B than workload A. Workload B generates more reads. This increases the likelihood of an anomaly being observed before it is overwritten by a subsequent write.

Table 1.3: Durability anomalies with YCSB and MongoDB.

Workload	**# of anomalies with 10 threads**	**# of anomalies with 100 threads**
A (50% read)	5,547 (0.001% of reads)	7,301 (0.0005% of reads)
B (95% read)	51,642 (0.006% of reads)	79,138 (0.002% of reads)

1.4 Thesis Contributions

Polygraph is motivated by several benchmarks [24, 37, 62, 59] that elevated the number of anomalies produced by a system to a first-class metric. A limitation of these benchmarks is that they are applicable to one workload. For example, BG [24] quantifies anomalies for social networking applications. Polygraph, by contrast, is a plug-and-play framework for quantifying the number of anomalies for diverse applications. The contributions of this thesis include:

- A conceptual plug-and-play tool for quantifying anomalies.

- A scalable framework that partitions Polygraph produced log records for parallel processing.

- A tool for detecting anomalies that violate isolation [26], linearizability [47], and atomicity property of transactions. Anomalies attributed to durability violations are categorized as linearizability anomalies. See Section 1.1 for details.

The rest of this dissertation is organized as follows. Chapter 2 describes and compares related work. Chapter 3 provides an overview of Polygraph and details how its components realize a plug-and-play framework. Chapter 4 describes the mechanism to detect anomalies and the evaluated benchmarks. Chapter 5 evaluates the scalability of Polygraph with two classes of applications: tree-structured and non-tree-structured applications. Chapter 6 describes and demonstrates the extensibility of Polygraph. Chapter 7 assesses the overhead costs of Polygraph. Next, Chapter 8 shares experiences using Polygraph to test different e-commerce and cloud applications. Lastly, Chapter 9 describes possible future research directions for Polygraph.

Chapter 2

Related Work

Polygraph is novel because it is general-purpose and supports diverse applications. It operates externally to an application, and its data storage infrastructure at the conceptual granularity of entities and their relationships.

A number of studies model an application at a physical level and instrumentalize it to produce log records to detect anomalies [19, 53, 78, 79, 72, 62, 46, 59]. Several detect serializability violations by building a dependency graph [78, 79, 31, 73]. They focus on the order of actions performed by transactions inside the data store. By contrast, Polygraph is external to the application (see Figure 2.1). Moreover, it exists at a conceptual level (that of the entity and relationship) and is independent of the data model assumed by a data store.

As a comparison, ACIDRain [73] is an offline tool that has false positives, functions with SQL systems only, detects isolation anomalies and not linearizability or atomicity anomalies, and focuses on anomalies that can be exploited by concurrency-related attacks.

As another comparison, the work by [79] operates at a middle-tier and tags data items to obtain dependency information. It detects isolation anomalies and not linearizability or atomicity anomalies. Their approach relies on the middle-tier to provide some level of concurrency control and requires an isolation level that is equal to or higher than Read Committed. Additionally, they do not support blind writes. Polygraph does not have such requirements.

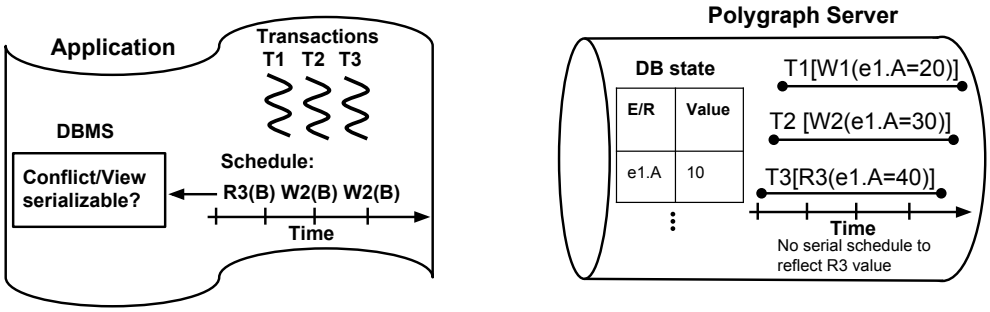


Figure 2.1: Techniques such as Conflict and View serializability are provided with an order of read and write of the data items. Polygraph is provided with the start and commit timestamps of transactions, their referenced entities/relationships, performed action, and each entity/relationship's property value.

Some studies use a metric different from the number of anomalies [59, 62]. Δ-atomicity [62] captures time-based staleness, defined as the amount of time between a stale read and the first unobserved write. In essence, a stale read must be extended Δ units of time to be considered atomic. It computes Δ for each key and takes the maximum to compute the global Δ for the system. YCSB++ [59] measures the time lag from one client completing an insert until a different client successfully observes the value. The probability of observing a fresh read at t units of time, which is the elapsed time from completing the last write until the start of the read, is the focus of [72]. The violation rate, defined as the number of transactions that violate integrity constraints divided by the number of committed transactions, is the focus of [40]. YCSB+t [37] employs a user-specified constraint to compute an anomaly score. Finally, ϕ-consistency [53] is a metric to assess how replicas may converge or diverge in a multi-replica system that encounters problems such as misconfigurations, network failures, etc.

Measurement of consistency at Facebook is the focus of [53]. The paper presents an offline framework that detects consistency anomalies for linearizability, per-object sequential and read-after-write consistency models. They do not consider multi-statement transactions.

Safety, atomicity, and regularity of a key value store are the focus of [19]. Its definition of a transaction is at the granularity of a single key-value pair. Polygraph supports transactions consisting of multiple entities and relationships that may be represented as either one or more key-value pairs in a key value store, multiple documents in a document store, or rows of different tables in a relational store. Table 2.1 shows a comparison of studies quantifying anomalies.

Other studies [41, 27] generate a synthetic workload and focus on session consistency models, such as Read Your Writes and Monotonic Reads. [41] also evaluates content and order divergence observed by different clients.

Table 2.1: Comparison of studies quantifying anomalies.

Study	**Consistency model(s)**	**Reported metric**
Polygraph	Strict Serializability [58, 29]	Amount of anomalies, Freshness confidence
Anderson et al. [19]	Safety, atomicity (equivalent to linearizability) and regularity [49]	Number of violations (cycles)
Bailis et al. [23]	Eventual consistency [71]	Probabilistic bounds on data staleness in versions and time
Fekete et al. [40]	Serializability [28]	Integrity constraints violation rate
Dey et al. (YCSB+t) [37]	Serializability [28]	Anomaly score using user specified constraints
Golab et al. [62]	Linearizability [47]	Δ-atomicity
Golab et al. [46]	Linearizability [47]	Γ (Gamma)
Liu et al. [52]	Read-your-write and linearizability [70]	Probability of satisfying a consistency model
Lu et al. [53]	Linearizability [47], per-object sequential [33, 50] and read-after-write (offline), Eventual consistency (online)	Reported the percentage of anomalies for the offline approach and ϕ-consistency metric for the online approach
Patil et al. (YCSB++) [59]	Eventual consistency [71]	Measures the time lag from one client completing an insert until a different client successfully observes the value
Wada et al. [72]	Read-your-write and monotonic read consistency	Freshness confidence [25]
Zellag et al. [79]	Serializability [28]	Number of violations (cycles)

Chapter 3

Plug and Play

This chapter details how the components of Polygraph realize a plug-and-play framework. It describes and motivates the use of a distributed stream processing framework such as Kafka. Subsequently, it describes the structure of Polygraph records streamed into Kafka. Next, it details how Polygraph's authoring tool computes the attributes to partition log records for parallel processing by Polygraph servers to quantify anomalies.

3.1 Overview

This thesis uses the following terminology. A read transaction consists of one or more read actions. A write transaction consists of one or more insert, update, or delete actions. A read/write transaction consists of one or more read actions in combination with one or more insert, update, and delete actions. To simplify discussion and without loss of generality, this dissertation does not discuss read/write transactions because their read actions are treated the same as those of read transactions and their write actions are treated the same as those of write transactions.

Polygraph represents a transaction as a log record. It computes the partitioning attributes of log records (transactions) and employs a messaging system called Apache Kafka [20] to partition these records for parallel processing by Polygraph servers. The choice of Kafka is arbitrary, and one can use an alternative such as RabbitMQ [61] instead.

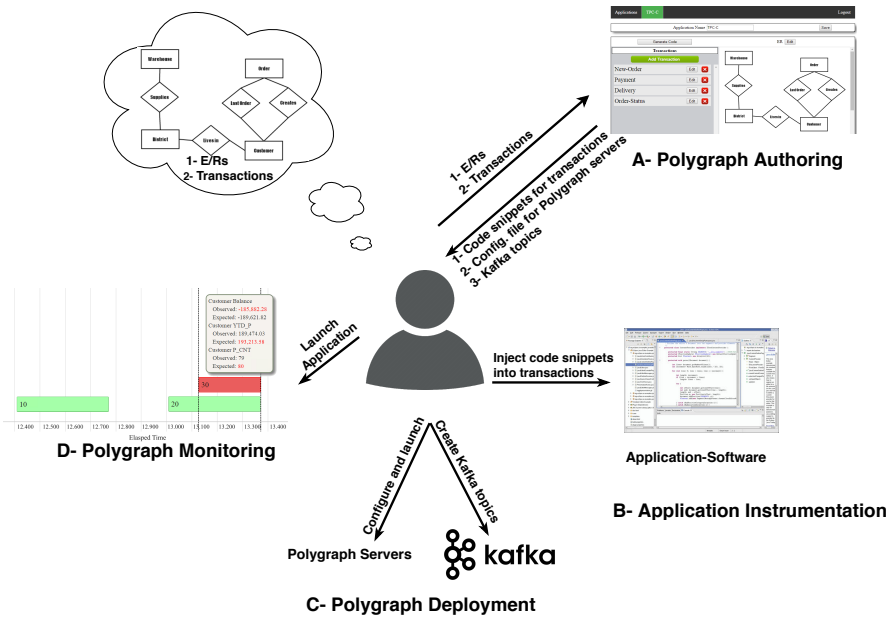


Figure 3.1: Overview of Polygraph.

Figure 3.1 shows how an experimentalist uses Polygraph in three distinct steps: Authoring (3.1.A and 3.1.B), Deployment (3.1.C), and Monitoring (3.1.D). During the authoring phase, an experimentalist employs Polygraph's visualization tool to identify entities and relationship sets of an application, and those actions that constitute each transaction and their referenced entity/relationship sets. An action may insert or delete one or more entities/relationships, and/or read or update one or more attribute value of an entity/relationship. For example, the payment transaction of the TPC-C benchmark reads and updates a customer entity.

Authoring produces the following three outputs. First, for each transaction, Polygraph generates a transaction-specific code snippet to be embedded in each transaction. These snippets generate a log record for each executed transaction. They push log records to a distributed framework for processing by a cluster of Polygraph servers. Second, it generates a configuration file for Polygraph servers, customizing it to process the log records. Third, it identifies how the log records

should be partitioned for parallel processing, including both Kafka topics and the partitioning attributes of the log records (see Figure 3.3).

Figure 3.2 shows an example of TPC-C's payment transaction code snippet generated by Polygraph. First, the experimentalist is asked to place line 27 before the start of the transaction to capture its start time. Lines 32-46 are placed after the commit of the transaction. They generate the transaction log record and stream it to Kafka. The log record contains the information of entities referenced by this transaction along with their properties and values. For example, lines 34-42 capture the customer entity referenced by this transaction and its properties. Line 36 represents the balance property of the customer entity. The third parameter ("R") specifies the action type which can be read ("R"), new value update ("N") or simple change ("I"), such as increment or decrement. The balance property is added twice in lines 36 and 39 representing the two actions performed by the payment transaction. It reads and updates a customer entity. Code variables, such as "oldBalance" and "newBalance" are provided by the experimentalist during Authoring to capture values of properties. Line 44 generates the log record. The first parameter ("Z") specifies the log record type. It can be "R" for read transactions, "W" for write transactions or "Z" for read-write transactions. The last line streams the generated log record with the transaction id to the "TPCC" topic and its assigned partition number.

During the deployment phase, an experimentalist deploys (1) the Kafka brokers and Zookeepers and creates the topics provided in Step A, (2) Polygraph servers using the configuration files provided in Step A, and (3) the application whose transactions are extended with the code snippets from Step A.

```
26        // ********************************************************************//
27        long startTime = System.nanoTime(); // Place this line before the start of the transaction.
28        // ********************************************************************//
29        /*
30         * Place the below code segment after the commit of the transaction. //
31         ********************************************************************/
32        long commitTime = System.nanoTime();
33        ArrayList<Entity> transEntities = new ArrayList<Entity>();
34        String customerKey = warehouseID + "-" + districtID + "-" + cutomerID;
35        Property[] propsCustomer = new Property[6];
36        propsCustomer[0] = new Property("balance", oldBalance, 'R');
37        propsCustomer[1] = new Property("payment_cnt", oldPaymentCount, 'R');
38        propsCustomer[2] = new Property("ytd_payment", oldYtd, 'R');
39        propsCustomer[3] = new Property("balance", newBalance, 'N');
40        propsCustomer[4] = new Property("payment_cnt", newPaymentCount, 'N');
41        propsCustomer[5] = new Property("ytd_payment", newYtd, 'N');
42        Entity eCustomer = new Entity(customerKey, "Customer", propsCustomer);
43        transEntities.add(eCustomer);
44        String logRecord = getLogRecordString('Z', "Payment", transactionID, startTime, commitTime, transEntities);
45        int partition = (warehouseID % TPCC_NUM_PARTITIONS);
46        kafkaProducer.send(new ProducerRecord<String, String>("TPCC", partition, transactionID, logRecord));
47        // ********************************************************************//
```

Figure 3.2: An example of TPC-C's payment transaction code snippet.

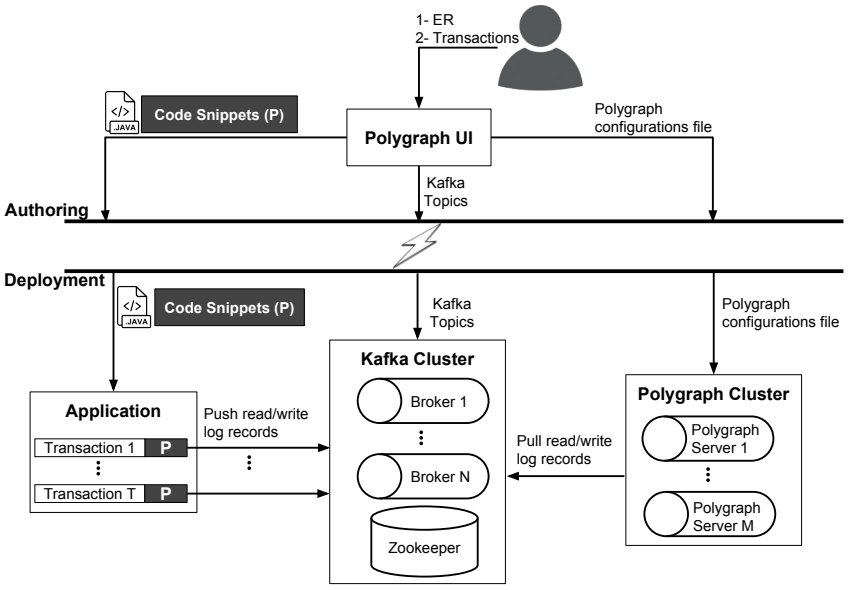


Figure 3.3: Authoring and deployment steps of Polygraph.

During the monitoring phase (see D in Figure 3.1), an experimentalist uses Polygraph's monitoring tool to view those read transactions that produce anomalies. For each such transaction, Polygraph shows the transactions that read/wrote the entities/relationships referenced by the violating transaction.

Polygraph code snippets embedded in a transaction stream log records into Apache Kafka, a distributed framework that stores streams in a fault-tolerant manner. Hence, an experimentalist may monitor an application in two possible modes: offline, by processing log records generated some time ago and buffered in

Kafka, or online, by processing log records as they are produced by the application and streamed into Kafka. When Polygraph is used in an online manner with an application, its quantified number of anomalies may vary from one run to another. This variance may be attributed to a changing workload, dynamic nature of thread execution with unpredictable context switches, and others. Given one run with its log records stored in Kafka, repeated execution of Polygraph produces the same results.

Polygraph stops being an online tool when many write transactions reference the same entity/relationship and overlap in time. For example, if 20 write transactions overlap one another and reference the same entity/relationship, then a thread of Polygraph must compute 20! serial schedules in the worst case scenario. This expensive operation renders Polygraph to be an offline tool. We can emulate this with YCSB Workload A (50% updates) and 100 threads generating requests in a very skewed manner.

Polygraph is a scalable framework that utilizes parallelism to quantify the amount of anomalies produced by an application. It partitions Polygraph-produced log records that reference different entities and relationships to be processed independently. This is achieved by computing one or more partitioning attributes for log records in its authoring step (see Section 3.4). The value of these attributes enables Kafka to construct partitions of log records. Log records in different partitions can be processed independently of one another because they reference entities and relationships that are mutually exclusive. A partition is assigned to at most one thread of a Polygraph server for processing. Polygraph servers pull log records from Kafka and validate them to detect anomalies. They build a conceptual database that reflects the values of different entities and relationships (see Section 4.1).

Polygraph is general-purpose and supports diverse applications with read-heavy and write-heavy workloads. The author has used Polygraph to evaluate alternative implementations of benchmarks including TPC-C [15], YCSB [34], BG [24], SEATS [67], and TATP [75] (see Section 4.3).

3.2 Apache Kafka

A Kafka cluster consists of one or more brokers that store streams of *records* in categories called *topics*. Each record consists of a key, a value, and a timestamp. Each topic consists of partitioned logs. Logs are partitioned and/or replicated across Kafka brokers. Kafka uses Zookeeper as a consensus service, electing partition leaders and detecting the addition and removal of Kafka brokers.

Kafka's topics with partitioned logs enable Polygraph to scale and facilitate the streaming of log records. With Polygraph, the key of a Kafka record is a transaction ID, its value is a structured log record (see Section 3.3), and its timestamp is the system clock of the server hosting the application. The Polygraph code snippets embedded in each transaction generate these records and and publish them to a Kafka topic. The code snippet assigns each record to a Kafka-partitioned log using an attribute of the structured log record (computed during the authoring phase; see Section 3.4). Kafka routes records to the appropriate broker with a Kafka-partitioned log. A broker appends records to its partitioned log in the order sent by the application. Polygraph validation threads pull records and process them to detect anomalous reads, observing log records in the same order produced by the Polygraph code snippet executed by the application.

Kafka's modularity simplifies software development and the deployment of Polygraph. The Kafka cluster retains all published log records—whether or not

they have been consumed—for a configurable period of time. Polygraph requires this feature to compute transactions that caused anomalous reads. Kafka frees Polygraph servers from almost all memory management issues, except for the buffering of some log records detailed in Section 4.1.1.

3.3 Polygraph Log Records

The value of a Kafka record is a structured log record that represents a transaction. It contains the start and commit time of a transaction, performed action (e.g., read, update, insert, delete), and a list of all entities/relationships manipulated by each action. For every read entity/relationship, the log record contains the observed value. For every updated entity/relationship, the log record contains the new value written or a simple change such as increment or decrement. For every inserted entity/relationship, the log record contains the attribute values of the new entity/relationship. For every deleted entity/relationship, the log record contains the primary attribute value of the deleted entity/relationship.

In the authoring step, Polygraph identifies a field of the log record as the partitioning attribute value. This value is used by the generated code snippet to assign the Kafka record to a partitioned log assigned to a broker. In Figure 3.4, the primary key of the Member entity (161) is the partitioning attribute value.

```
U,InviteFriend,1,450015459585869,450015554181298,MEMBER;161;PENDING_CNT:+1:I
```

Figure 3.4: Example of Polygraph log record.

Figure 3.4 shows a write log record pertaining to the Invite Friend transaction of the BG benchmark. It contains the transaction ID (1), the start and commit timestamps of the transaction, the entity set Member, the primary key of the entity

being updated (161), and the update impacting the pending_cnt attribute of the entity and changing its value by 1 using increment.

3.4 Code Generation and Partitioning of Log Records

In the authoring step, an experimentalist identifies each transaction and the collection of entities and relationships manipulated by that transaction (see Figure 3.3). In turn, Polygraph's user interface generates a code snippet to embed in a transaction. The code snippet constructs a log record for the transaction and publishes it to a Kafka topic. In addition, it identifies the partitioning attribute of the log record and applies a hash function to the value of this attribute to assign it to a partition for processing. This code snippet ensures log records of transactions referencing the same entity/relationship are directed to the same partition. Each partition is assigned to one validation thread. This thread builds a snapshot of the database state (e.g., values) pertaining to these entities and relationships.

A key challenge is computing the Kafka topic and partitioning attribute of the log records. As shown below, Algorithm 1 addressed this challenge. The input of Algorithm 1 is entity and relationship sets pertaining to an application and transactions that manipulate them. This is similar to the second and third columns of Table 4.2. The output is a set of Kafka topics along with their partitioning attributes.

Polygraph separates transactions that reference different entity and relationship sets as they cannot conflict with one another to produce anomalies. Each grouping identifies a Kafka topic.

Algorithm 1 Compute Kafka topics and partitioning attribute (ER, TXS)

1. Partition transactions in TXS that reference mutually exclusive entity and relationship sets into different groups.
2. Each grouping identifies a Kafka topic; let array K[ ] denote this list.
3. Initialize P[ ] = empty.
4. **for each** topic i **in** K[ ] {
   4.1. **for each** transaction T_i **in** i {
   4.1.1 Let PT_i = { p | p is a candidate partitioning attribute }, initialized to empty set.
   4.1.2. **if** (T_i references a M:N relationship set) {
   No partitioning attribute for the topic.
   break
   }
   4.1.3. Let CT = { t | t is a candidate tree }, initialized to empty set.
   4.1.4. Generate Tree T as follows:
   a. Each referenced entity set is represented as a node.
   b. Each referenced 1:M relationship set from e1 to e2 is represented as an edge from e1 to e2.
   4.1.5. **if** (T has a cycle) {
   No partitioning attribute for the topic.
   break
   }
   4.1.6. Add T to CT.
   4.1.7. **for each** referenced 1:1 relationship set from
   e1 to e2 {
   for each tree CT_i in CT {
   Make two copies of CT_i, CT_1 and CT_2.
   Add an edge from e1 to e2 in CT_1.
   Add an edge from e2 to e1 in CT_2.
   Remove CT_i from CT.
   if (CT_1 has no cycle) {
   Add CT_1 to CT.
   }
   if (CT_2 has no cycle) {
   Add CT_2 to CT.
   }
   }
   }
   4.1.8. **for each** CT_i in CT {
   if (a node of CT_i is either not reachable from its root or has more than one
   incoming edge from more than one node) {
   continue
   }
   if (T_i references only one entity of the root of CT_i) {
   Add the primary key of the root entity set to PT_i.
   }
   }
   }
   4.2. S= intersection set of all PT_i in a topic.
   4.3. **If** (S is empty) {
   Assign a constant value, C_i, for the partitioning attribute; let P[i]= C_i.
   } **else** {
   Let P[i]= a composite key of all keys in S as the partitioning attribute.
   }
   }
5. Return arrays K[ ] and P[ ].

To identify the partitioning attributes for a topic, Polygraph checks whether the application has tree-structured schema and computes the partitioning attributes by using this structure [66]. Log records of an application with tree-structured schema can be partitioned using the primary key of the root table. This class of application applies to many real-world OLTP workloads [66].

The details are as follows. First, Polygraph computes a set of candidate partitioning attributes for every transaction in the topic. Next, Polygraph computes the intersection set of all sets of candidate partitioning attributes for transactions in the topic. If the intersection set is empty, then there is only one partition: Polygraph assigns a constant as the partitioning attribute for that topic to route all log records to one partition. Otherwise, Polygraph assigns a composite key of the intersection set as the partitioning attribute. Selecting one key from the intersection set may result in fewer partitions, compared to a composite key of the intersection set.

Algorithm 1 computes a set of candidate partitioning attributes for a transaction as follows. If a transaction references a many-to-many relationship, it has no partitioning attribute because it does not form a tree. Otherwise, Polygraph generates a tree for the transaction (Step 4.1.4) by representing each referenced entity set as a node. Each referenced one-to-many relationship appears as an edge from one to many. The algorithm adds the tree to a set of candidate trees called CT in Step 4.1.6. Each one-to-one relationship from e1 to e2 can appear as an edge from e1 to e2 or as an edge from e2 to e1 (Step 4.1.7). To compute all candidate partitioning attributes, the algorithm duplicates each tree in CT to represent each possibility. Next, it removes all invalid trees from CT. In Step 4.1.8, it loops each tree in CT. The primary key of its root entity set is a candidate partitioning attribute if the transaction references only one entity of that root entity set. If the

transaction references more than one entity of the root entity set, then it has no partitioning attribute.

The complexity of the algorithm to compute Kafka topics and their partitioning attributes depends on the number of transactions, T, and number of one-to-one relationship sets, N. It builds a tree for every transaction and duplicates the set of candidate trees for every one-to-one relationship set. Additionally, for each candidate tree, it performs a Depth First Search to check for cycles. The complexity of the algorithm is: $O(T * (N^2 + 1) * (V + E))$ for an application with V entity sets and E relationship sets that include one-to-one and one-to-many relationship sets.

Example

Figure 3.5 shows an example of an application consisting of four entity sets: Employee, Project, Department and Resource. It has three relationship sets: Own, Has and Work. The application consists of four transactions: AddResource, UpdateDepartment, AddProject and AssignEmpToProj. Polygraph constructs two topics for the application because the transactions reference two mutually exclusive groups of entity and relationship sets. The first topic consists of AddResource and UpdateDepartment transactions, which reference Department, Resource and Own entity/relationship sets. The second topic consists of AddResource and UpdateDepartment transactions, which reference Project, Employee and Work entity/relationship sets. Algorithm 1 constructs multiple partitions for the first topic, assigning the primary key of the department entity as the partitioning attribute. Algorithm 1 also constructs a single partition for the second topic

because of the many-to-many Work relationship between Project and Employee entity sets, however.

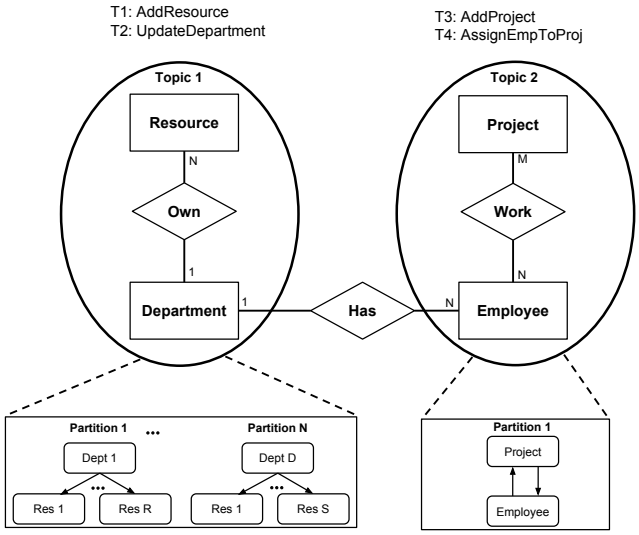


Figure 3.5: Example of constructed topics and partitions for an application.

Chapter 4

Quantifying Anomalies

This chapter describes Polygraph's algorithm for detecting anomalies and establishing the values of entities/relationships. Following this is an evaluation with different benchmarks.

4.1 Detecting Anomalies

A cluster of multi-threaded Polygraph servers (see Figure 3.3) pulls log records from Kafka and processes them to detect anomalies. Each thread receives one or more partition logs for processing. This thread, a *validation* thread, builds a snapshot of the database (at a conceptual level) as it processes log records. Every time when a read transaction observes a value that is not consistent with its snapshot it has detected an anomaly.

At launch, a Polygraph server's input specifies the number of Kafka-partitioned logs and how many threads it must launch. Polygraph assigns Kafka partitions to threads using a hash function. Polygraph assigns a Kafka partition to, at most, one thread.

The validation thread uses a hash table to maintain the value of its assigned entities/relationships. The primary key of an entity/relationship identifies its value. This value contains the candidate database values of that entity/relationship

(if any) and a set of candidate serial schedules for the write transactions referencing it. The validation thread computes these using the structured logs fetched (see Section 3.3) from its assigned Kafka-partitioned logs.

The validation thread computes candidate serial schedules that satisfy the isolation property of processing concurrent transactions and follows a *strict serial order* [58]. Given T overlapping transactions, there are T! serial schedules. Once a write transaction commits, all subsequent read transactions that reference the same entity/relationship must observe the value produced by the write transaction. Otherwise, the read transaction has observed an anomaly. Moreover, at most one serial schedule may validate all read transactions. Read transactions prune down candidate serial schedules incrementally as Polygraph validates them, eliminating those schedules that do not match their observed values. Once there is no candidate serial schedule to validate a value observed by a read transaction, Polygraph marks the value as anomalous.

Computing strict serial schedules enables Polygraph to detect both transaction isolation anomalies [26] and linearizability anomalies [47]. This is because strict serializability is a stronger consistency model than both serializability [58, 29] and linearizability [47, 53]. Strict serializability prohibits all executions unacceptable by serializability or linearizability, but not vice-versa. For non-transactional systems, Polygraph detects linearizability anomalies by representing each action as one transaction [47].

A disadvantage of the current approach Polygraph uses to compute candidate serial schedules is that the computations may be intractable for certain workloads. These are heavily skewed workloads with high updates, such as YCSB's 50% updates workload with Zipfian distribution.

Algorithm 2 shows how each validation thread processes log records that identify different transactions. The input of Algorithm 2 is a read log record representing a read transaction. For every read transaction, Algorithm 2 establishes the value of each entity/relationship observed by the read transaction as either valid or anomalous. It does so by enumerating the possible candidate serial schedules that include those write transactions that reference the same entity/relationship with a start and commit time that overlaps the read transaction.

Algorithm 2 satisfies the following two properties. First, to validate a read transaction, it considers all write transactions that overlap it transitively, excluding writes starting after the commit time of the read. Second, once it discards a set of serial schedules to validate a read transaction, it discards that set for all future reads that reference the same entity/relationship. What follows describes each property in turn. Subsequently, an example will illustrate the algorithm.

The read and write transactions shown in Figure 4.1 illustrate the first property. All transactions reference the same entity $Cust1$. Transaction W2 overlaps R1 transitively. When computing serial schedules to validate the value 20 observed by R1, Algorithm 2 (see Step 4) includes W2 when computing the set of candidate serial schedules for R1. Consequently, R1 may observe the value written by W2 since one possible serial schedule is W1:W2:R1:R2. A naive algorithm that considers only W1 when enumerating serial schedules would incorrectly mark R1 as having observed an anomalous value.

To illustrate the second property (see Figure 4.2), the ordering W1:W2 used to validate R1 must also be used to validate R2. If R2 had observed the value 10 for $Cust1.bal$, the algorithm would have marked it as invalid and detected an anomalous read (because the schedule W2:W1 was discarded when validating R1).

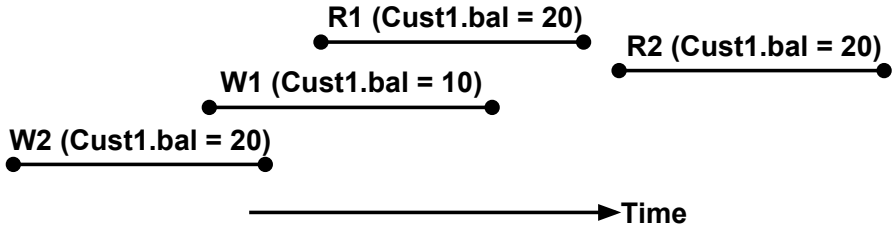


Figure 4.1: To identify read transactions that observe anomalous values, a Polygraph thread must consider all write transactions that overlap the read transaction transitively.

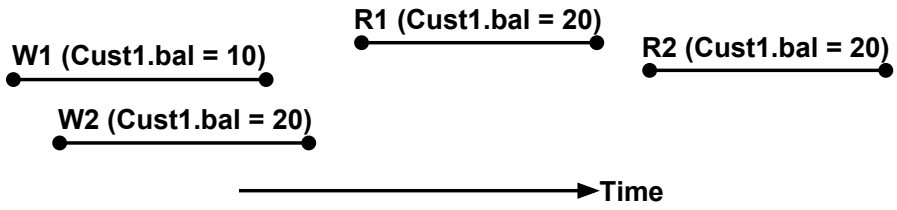


Figure 4.2: Discarded serial schedules to validate one read transaction are discarded for all future read transactions.

Algorithm 2 is general-purpose and works for transactions that manipulate a set of entities/relationships Q. In its pseudo-code, SSL refers to the maintained set of serial schedules for an entity/relationship. The algorithm assumes transaction sorting based on start time. Figure 4.3 is used to illustrate the validation process. This example shows three transactions referencing the balance (bal) property of the same entity $Cust1$, $Q = \{Cust1\}$, assuming the initial value of $Cust1.bal$ is 10.

The algorithm selects the first read, $R1$, for validation. It places all unprocessed write transactions with a start time earlier than $R1$'s commit time into a set denoted as $Writes$, $Writes$=$\{W1,W2\}$. *W-Before-R1* is a subset of $Writes$ that includes writes with start times earlier than $R1$'s start time and that reference an element of Q, *W-Before-R1*=$\{W1,W2\}$. Since the initial state of SSL is empty, the algorithm removes nothing in Step 5. Next, it computes all valid serial schedules using *W-Before-R1*, *SL-Before-R1*=$\{W1{:}W2, W2{:}W1\}$.

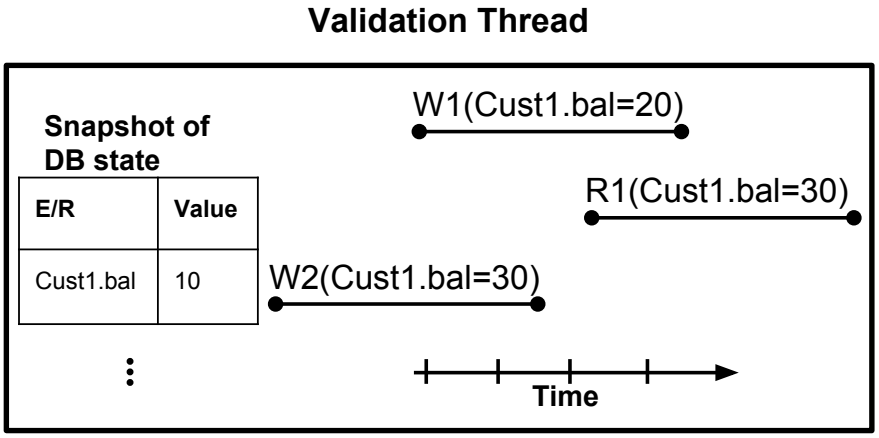


Figure 4.3: An example schedule consisting of two write transactions and one read transaction.

In Step 7, Polygraph combines the serial schedules by computing the cartesian product between SSL, which is empty, and *SL-Before-R1* and stores the result in SSL, SSL=$\{W1{:}W2, W2{:}W1\}$. It removes the second occurrence of all duplicate writes in a serial schedule in SSL, if any. Then, starting from the end of a serial schedule, it removes any write that overlaps with $R1$ until it encounters a non-overlapping write, SSL=$\{W1{:}W2, W2\}$.

It lets *T-Overlap-R1* be a set of $R1$ and any write transaction in $Writes$ that overlaps $R1$ in time and references an element of Q, *T-Overlap-R1* = $\{R1, W1\}$. It computes all valid serial schedules for that set, *SL-Overlap-R1*=$\{W1{:}R1, R1{:}W1\}$.

Then, it computes *Validate-R1*, which is a set of candidate serial schedules to validate $R1$. The set is the result of the cartesian product between SSL and *SL-Overlap-R1*, *Validate-R1*=$\{W1{:}W2{:}W1{:}R1, W1{:}W2{:}R1{:}W1, W2{:}W1{:}R1, W2{:}R1{:}W1\}$. The algorithm removes the second occurrence of all duplicate writes in a serial schedule in *Validate-R1*, *Validate-R1*=$\{W1{:}W2{:}R1, W1{:}W2{:}R1, W2{:}W1{:}R1, W2{:}R1{:}W1\}$. Next, the algorithm removes duplicate serial schedules in *Validate-R1*, *Validate-R1* =$\{W1{:}W2{:}R1, W2{:}W1{:}R1, W2{:}R1{:}W1\}$. Now,

Validate- $R1$ contains a set of candidate serial schedules to validate $R1$. Polygraph compares the values $R1$ observes with the expected values produced by each candidate serial schedule.

In Figure 4.3, the computed candidate values for $W1$:$W2$:$R1$ and $W2$:$R1$:$W1$ serial schedules in *Validate-R1* match $R1$'s observed value, while the serial schedule $W2$:$W1$:$R1$ does not match $R1$'s observed value. Therefore, Polygraph marks $R1$ as a valid read transaction and the first and last serial schedules persist for $Cust1$.

4.1.1 Delayed Log Records

A multi-threaded application may create log records in a manner that does not represent the order of transactions the application executes. A context switch may suspend the execution of a thread after it has executed a transaction and created its log and prior to transmitting this log record to Kafka (see Thread 1 in Figure 4.4). During this time, a different thread (Thread 2) executes its transaction, generates the corresponding log record, and transmits this log record to Kafka. A Kafka broker appends log records to its partitioned log in the order the application produces. If the transaction processed by Thread 1 writes a value that is read by Thread 2, then their reverse order of processing causes Polygraph to detect an incorrect anomaly observed by Thread 2. Even when both threads execute read transactions, their orderly processing is important because other threads may exist that process write transactions referencing the same entity/relationship and overlapping them in time.

Polygraph detects delayed log records, re-orders them in the right sequence, and reverts back to a previous database snapshot to process this corrected sequence. Polygraph snapshots its hash table that maintains the value of entities/relationships after processing a fixed number of log records, say 1000. The

Algorithm 2 Detect Anomalies (Read Transaction $R1$)

1. $Q = \{\ e \mid e$ is an entity/relationship referenced by $R1\ \}$.
2. $SSL = \{\ (s) \mid s$ is a candidate serial schedule for write transactions referencing an element of $Q\ \}$, initialized to empty set.
3. $Writes = \{\ w \mid w$ is an unprocessed write transaction, $w.Start < R1.End\ \}$.
4. *W-Before-R1* $= \{\ w \mid w \in Writes$, references an element of Q, $w.Start < R1.Start\ \}$.
5. **for each** $SSL_i \in SSL$ {
   loop from the end to the start, **for each** $W_i \in SSL_i$ {
   if ($W_i \in$ *W-Before-R1*) {
   remove W_i from SSL_i
   } **else** {
   break
   }
   }
   }
6. *SL-Before-R1* $= \{\ (s) \mid s$ is a valid serial schedule using $t_i \in$ *W-Before-R1*$\}$.
7. $SSL =$ Compute the cartesian product of SSL and *SL-Before-R1*.
8. **for each** $SSL_i \in SSL$ {
   Remove second occurrences of a write transaction.
   loop from the end to the start, **for each** $W_i \in SSL_i$ {
   if (overlap($W_i, R1$)) {
   remove W_i from SSL_i
   } **else** {
   break
   }
   }
   }
9. *T-Overlap-R1* $= \{\ t \mid t$ is $R1$ or $t \in Writes$ that references an element of Q and overlaps in time with $R1\}$.
10. *SL-Overlap-R1* $= \{\ (s) \mid s$ is a valid serial schedule using $t_i \in$ *T-Overlap-R1*$\}$.
11. *Validate-R1* $=$ Compute the cartesian product of SSL and *SL-Overlap-R1*.
12. **for each** $V_i \in$ *Validate-R1* {
    Remove second occurrences of a write transaction.
    }
13. Remove duplicate schedules from *Validate-R1*.
14. $CandidateValues = \{\ v \mid v$ is a candidate value for an entity/relationship $\in Q$ using a schedule of *Validate-R1*$\}$.
15. **if** (match($R1$, $CandidateValues$)) {
    Mark $R1$ as observing a valid value.
    Discard schedules that compute a different value from SSL.
    } **else** {
    Mark $R1$ as observing an anomalous value.
    }

timestamp of the last read transaction that Polygraph validated using this snapshot is the timestamp of this snapshot. Moreover, Polygraph maintains the timestamp

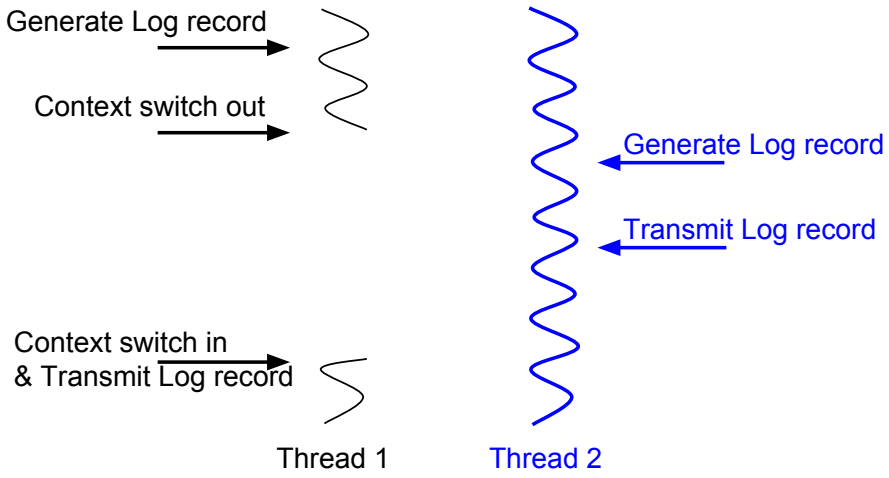


Figure 4.4: Log records transmitted out of order.

of its last processed log record. Should the next log record have a start timestamp before the last, then it has detected a delayed log record $T_{delayed}$. Polygraph re-arranges this log record relative to previous ones and restores the state of its database to a snapshot that is before the start time of $T_{delayed}$. Next, it processes the re-arranged log records that correctly reflects the ordering of the delayed log record. To minimize the frequency of encountering delayed log records, Polygraph maintains a batch of log records and sorts them based on their start timestamps prior to processing them.

4.2 Computing Entity/Relationship Values

Polygraph establishes the value of an entity/relationship based on the processed transactions and their actions. An insert action specifies a value for all properties of an entity/relationship. A validated read or an isolated write transaction sets the value of one or more properties of an entity/relationship. In case of overlapping write transactions referencing the same entity/relationship, Polygraph maintains

multiple candidate values for that entity/relationship. Polygraph prunes this list as it processes additional read and write transactions.

A limitation of this approach is that, should the first value observed for an entity/relationship be anomalous, Polygraph flags all subsequent reads as invalid even though they are correct. One way to prevent the erroneous states from carrying over is to periodically reset the database value of entities/relationships marked as anomalous.

4.3 Evaluated Benchmarks

Table 4.1: Details of studied benchmarks.

Benchmark	**Target Application**	**Max Number of Partitions**
BG	Interactive social networking actions	Number of members
TPC-C	E-Commerce, wholesale supplier	Number of warehouses
SEATS	Airline ticketing systems	1
TATP	Telecommunication	Number of subscribers
YCSB	User profile cache, threaded conversations, etc.	Number of users

The author evaluated the feasibility of Polygraph using multiple benchmarks including TPC-C [15], BG [24, 17], SEATS [67], TATP [75], and YCSB [34]. All nodes used to conduct experiments have 8 hyperthreaded i7-4770 (4 core) 3.4 GHz CPU, 16 GB of memory, and a 1 Gbps network card, unless specified otherwise.

The second column of Table 4.1 provides a brief description of each benchmark. The third column shows the maximum number of partitioned logs supported by each benchmark. Except for SEATS, all other benchmarks provide for the use of

concurrent processing of log records.[1] Additional discussion of this appears below in Chapter 5, where the focus is on the scalability of Polygraph.

Polygraph cannot perform query processing, which is a limitation. Instead, it looks up entities and relationships using their primary keys. For example, Stock-Level transactions in TPC-C require special handling to identify the items of the last 20 orders in a district. One can extend and tailor Polygraph to support Stock-Level, but doing so renders Polygraph workload-specific. Chapter 6 describes and demonstrates the extensibility of Polygraph.

Table 4.2 shows the different transactions that constitute each benchmark. The actions (read, insert, update, and/or delete) performed on the entities/relationships accompany each transaction. The table additionally shows the minimum, maximum, and average size of log records produced by the Polygraph-generated code snippet embedded in each transaction. The size of log records for a transaction is not a constant for several reasons. First, different instances of a transaction may reference a different number of entities/relationships, resulting in a larger log record for those instances that reference a higher number of entities/relationships. An example is the Delivery transaction of TPC-C. Second, a transaction may read an entity that does not exist. This happens with YCSB, where a read transaction retrieves a deleted user record which results in a compact log record. Third, Polygraph uses a string representation of values for its structured record that results in variance – e.g., the string representation of value 1245 is 4 bytes long, while that of value 9 is one byte long.

---

[1]With Polygraph, we represent SEATS with three entity sets: Customer, Flight, and Airline. Airline and Flight participate in a one-to-many Operates relationship. Customer and Flight participate in a many-to-many Reservation relationship. Customer and Airline participate in a many-to-many Frequent-Flyer relationship. The many-to-many relationships causes Polygraph to assign a constant value for the partitioning attribute. Hence, Polygraph sends all log records to one partitioned log for processing.

Table 4.2: The type and size of log records generated by each transaction for the five studied benchmarks.

Bench-mark	**Transaction name**	**Type**	**Min**	**Avg**	**Max**
BG	View Friend Requests	Read member entity	80	87	89
	Thaw Friendship	Update two member entities	78	84	86
	Reject Friend Request	Update member entity	77	83	85
	Accept Friend Request	Update two member entities	77	91	101
	Invite Friend	Update member entity	78	84	86
	List Friends	Read member entity	74	81	83
	View Profile	Read member entity	74	81	99
TPC-C	New-Order	Insert order entity, read and update district entity, update the last order relationship	170	187	197
	Payment	Read and Update customer entity	133	149	163
	Delivery	Update order and customer entities	840	880	927
	Order-Status	Read the last order relationship, read customer and order entities	194	224	246
SEATS	New Reservation	Insert reservation relationship, read and update flight and customer entities, update frequent-flyer relationship	282	317	322
	Find Open Seat	Read a flight entity	75	82	83
	Update Customer	Read and update a customer and frequent-flyer	149	430	22067
	Delete Reservation	Delete reservation relationship, update frequent-flyer, update a flight and customer entities	201	221	266
	Update Reservation	Update reservation relationship	100	117	120
TATP	Update_Location	Update subscriber entity	66	71	73
	Get_Subscriber_Data	Read subscriber entity	74	79	81
	Update_Subscriber_Data	Update subscriber and special_facility entities	59	63	64
	Insert_Call_Forwarding	Insert call forwarding entity	64	67	69
	Delete_Call_Forwarding	Delete call forwarding entity	66	70	72
	Get_Access_Data	Read access_info entity	64	69	71
YCSB	Read	Read user entity	58	77	290
	Delete	Delete user entity	60	66	68
	Modify	Read and Update user entity	322	392	554
	Update	Update user entity	311	317	319
	Insert	Insert user entity	315	319	321

Chapter 5

Scalability

The number of unique values for the partitioning attribute of the records dictates the scalability of Polygraph. The third column of Table 4.1 shows this for the different benchmarks. Polygraph constructs multiple partitioned logs for tree-structured applications, such as BG and TPC-C. By contrast, it constructs a single partitioned log for non-tree-structured applications, such as the SEATS benchmark.

This chapter focuses on the scalability of Polygraph with these two classes of applications: tree-structured and non-tree-structured. The first section contains an evaluation of the vertical scalability of two tree-structured applications: BG and TPC-C. The second presents and evaluates two designs to scale with non-tree-structured applications, such as the SEATS benchmark.

5.1 Tree Structured Applications

Polygraph constructs multiple partitions for tree-structured applications, such as the TPC-C and BG benchmarks. The number of unique values for an entity set dictates the number of partitioned logs. For BG, this comes from the Member entity set, while for TPC-C from the Warehouse entity set.

The present section discusses the vertical scalability of Polygraph with these two benchmarks. Analysis relied on Polygraph processing log records in an offline mode where a benchmark had generated all its log records and inserted them in

Kafka. The study does not include discussion of an online version of Polygraph because its scalability is dictated by the rate at which a benchmark produces log records.

The experimental design consisted of a Kafka cluster with five brokers and one Zookeeper instance, and no replication of log records for data availability. The Polygraph cluster consisted of one server. The author repeated each experiment five times showing the lower and upper bounds for the scalability results. The use of two servers enabled comparison of Polygraph scalability. One had an i7-4770 (four-core) 3.4 GHz CPU, 16 GB of memory, and a 1-Gbps network card. The other was an Emulab [74] d820 server that was Dell Poweredge R820 with 128 GB of memory and four 2.2 GHz, 64-bit, eight-Core Xeon E5-4620 processors (32 cores). Hyperthreading was disabled on both servers. For both BG and TPC-C, 32 cores offer improved Polygraph scalability compared to four cores.

5.1.1 BG

For BG, two different workloads offered a contrast between a skewed or uniform access pattern (see Figures 5.1 and 5.2). With a skewed access pattern, roughly 62% of requests referenced 20% of data items. Both workloads pertain to a mix of read and write actions shown in Table 5.1, generating[1] approximately 17.1 million read log records and 2.6 million write log records.

With the uniform workload and four cores, Polygraph scales linearly up to four threads (the number of cores) (see Figure 5.1). With four or more threads, the CPU cores are 90-100% utilized on the average, capping vertical scalability of Polygraph. Similarly, with 32 cores and the uniform workload, Polygraph scales

[1]The social graph consisted of 10,000 members and 100 friends per member. BG produced a load using 100 concurrent threads for one hour.

linearly up to 32 threads (the number of cores). With 32 or more threads, the CPU cores are 90-100% utilized on the average, capping vertical scalability of Polygraph.

Figure 5.1 shows Polygraph scales better with the uniform access pattern as compared to the skewed access pattern (see Figure 5.2). This is due to log records being partitioned across validation threads more evenly. For example, with four cores, 32 threads and the skewed access pattern, the fastest thread finishes 39% sooner than the slowest thread. This difference is only 8% with the uniform access pattern, which explains its superior scalability characteristics[2].

Table 5.1: A mix of interactive social networking actions with BG. Reads constitute 90% of the mix.

BG Social Actions	Action ratio
View Profile	30%
List Friends	30%
View Friend Req	30%
Invite Friend	4%
Accept Friend Req	2%
Reject Friend Req	2%
Thaw Friendship	2%

5.1.2 TPC-C

The OLTP-Bench [38] generated the TPC-C traces. TPC-C operated with 100 warehouses, 300 threads, and the default workload for three hours against MySQL. The total number of transactions was 2,008,619. With four cores, Polygraph validated these transactions in 66 minutes with 1 thread. By contrast, this time was 3.6 minutes with 32 threads.

[2]With 32 threads and four cores, Polygraph processes uniform in 5.73 minutes versus 7.9 minutes with skewed.

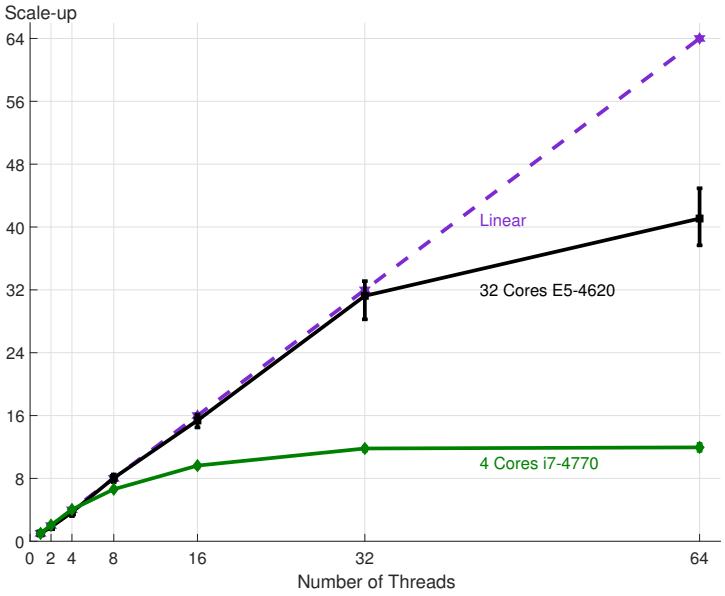


Figure 5.1: Vertical scalability with BG uniform workload

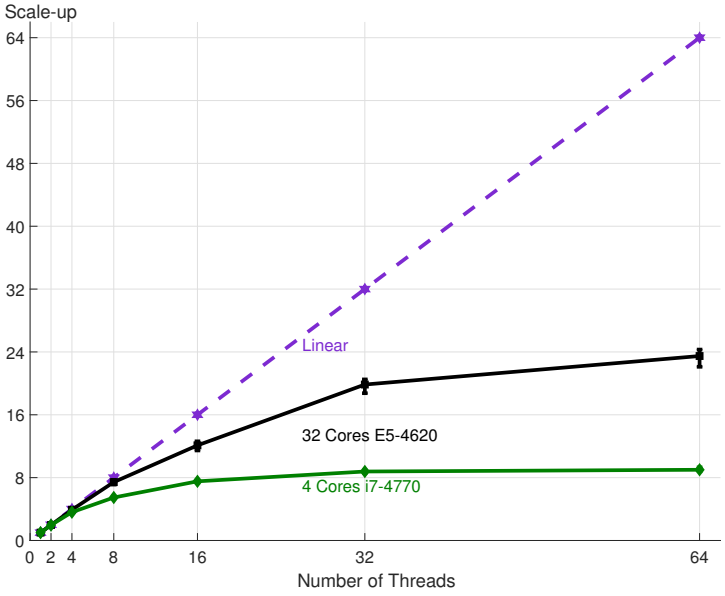


Figure 5.2: Vertical scalability with BG skewed workload

With four cores, Polygraph scales linearly up to four threads (the number of cores) (see Figure 5.3). With four or more threads, the CPU cores are 80-100% utilized on the average, capping vertical scalability of Polygraph. With 32 cores,

Polygraph scales linearly up to 16 threads. With 32 threads, there is a variance between the fastest and slowest thread to finish preventing linear scalability. The fastest thread finishes 13% sooner than the slowest thread. With 32 or more threads, the CPU cores are 80-100% utilized on the average, capping vertical scalability of Polygraph.

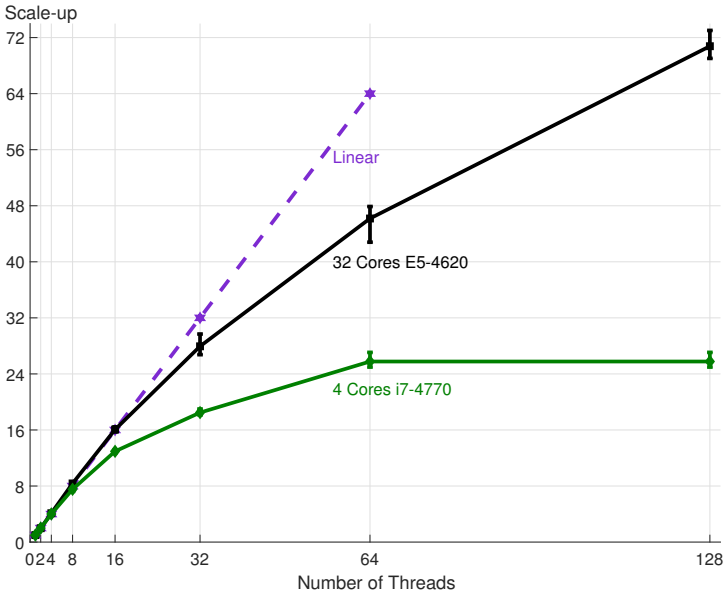


Figure 5.3: Vertical scalability with TPC-C

5.1.3 Kafka Latency

The time spent polling log records from Kafka may dominate validation time to limit the scalability of Polygraph. This is more evident with a read-heavy workload such as BG because it computes fewer serial schedules compared to a write-heavy workload such as TPC-C. With BG, the time spent polling log records from Kafka is 69% of the validation time. With TPC-C, it constitutes 40% of the validation time. Hence, the scalability of Kafka dictates Polygraph's scalability characteristics.

5.2 Non-tree Structured Applications

Polygraph constructs a single partition for log records with non-tree-structured applications, such as the SEATS benchmark. This reflects the lack of a partitioning attribute to divide the problem into mutually exclusive sub-problems that can be solved independently. The following sections present and evaluate two techniques to scale Polygraph with these applications. The first processes log records into groups which did not scale. The second is a temporal partitioning technique which enhances scalability of Polygraph by reducing its accuracy.

5.2.1 Grouping of Log Records

The method involved considering processing ρ log records into g mutually exclusive groups based on their referenced entities/relationships, using t threads to process different groups concurrently. An evaluation of this approach with the SEATS benchmark did not scale because the number of log records assigned to g groups was heavily skewed, with at least 95% of log records assigned to one group. This highlights the interdependence between the entities/relationships referenced by different SEATS transactions. The largest group dictates the overall processing time with one thread, limiting vertical scalability. This held true with ρ values of 10K, 50K, 100K records, g groups of 10, 100, and t threads of 1, 2, 4, and 8 (see Table 5.2).

g	τ
10	99.7%
100	99%

(a) ρ = 100,000

g	τ
10	99.7%
100	98.7%

(b) ρ = 50,000

g	τ
10	97.5%
100	95.1%

(c) ρ = 10,000

Table 5.2: Percentage of records in the largest group (τ) with different batch (ρ) and max group (g) sizes.

5.2.2 Temporal Partitioning

This technique partitions log records based on time, constructing temporal segments. Each segment has a duration of t time units (see Figure 5.4). These time segments are assigned to different validation threads (using a hash partitioning strategy) to be processed concurrently. A validation thread discards its maintained database state of entities/relationships after validating a time segment and starts with an empty database state with the next segment. This may impact the correctness of Polygraph, as discussed in more detail in the Correctness Section.

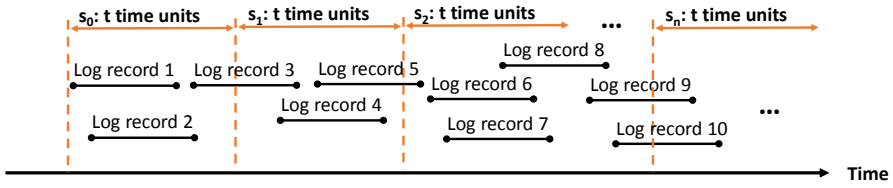


Figure 5.4: Partitioning of log records into time segments.

A validation thread processes read and write log records as follows. One validates read log records with a start time within the current time segment (see Figure 5.5). One discards read log records because one evaluates them in their corresponding time segment according to their start time. For write log records, a validation thread includes log records overlapping the current time segment or one of its read log records and discards others (see Figure 5.5). The following sections evaluate the scalability and correctness of this approach using the SEATS benchmark.

Scalability

The author used the SEATS benchmark to evaluate the scalability of the time partitioning approach. The process generated a trace running SEATS with 100

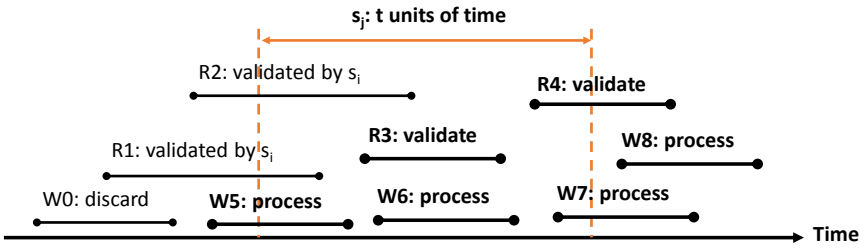


Figure 5.5: Processing of read and write log records in a time segment.

threads and 10 scale factor for one hour, generating approximately 7.1 million transactions. The author used one Emulab [74] d820 node to validate log records. This node was a Dell Poweredge R820 with 128 GB of memory and four 2.2 GHz, 64-bit, eight-core Xeon processors.

Table 5.3 shows the validation time in seconds with different numbers of validation threads and time segments. With 8, 16, and 32 time segments, the duration of a time segment is 450, 225 and 112.5 seconds respectively. The researcher staged log records into main memory in advance before measuring the validation time.

Table 5.3: Validation time for the first and last thread to finish with different numbers of validation threads and time segments.

Number of threads	**Validation time (sec)with 8 segments**		**Validation time (sec) with 16 segments**		**Validation time (sec) with 32 segments**	
	First	Last	First	Last	First	Last
1	1,837	1,837	1,815	1,815	1,596	1,569
2	858	1,023	892	924	785	807
4	416	665	437	473	416	448
8	249	572	257	408	241	281

Depending on the number of overlaps between reads and writes, the time to process each time segment differed. Table 5.3 shows the validation time for the first and last thread to finish. For example, with eight time segments and eight threads, the first thread finished 56% sooner than the last thread. This difference was reduced with a larger number of segments.

The load of a time segment is impacted by multiple factors, including the number of log records, their referenced entities/relationships, and how these log records overlap, determining the number of computed candidate serial schedules. Table 5.4 compares the load of the first and last thread to finish with eight time segments and eight threads. The author quantified the load of a time segment by measuring the number of processed log records, computed serial schedules, merged serial schedules and created candidate database states for referenced entities/relationships. Merging serial schedules occurs when processing a transaction that references multiple entities/relationships that have their own lists of candidate serial schedules. Consequently, the lists of candidate serial schedules for these entities/relationships are merged to validate that transaction.

Table 5.4: The load difference between the first and last thread to finish with eight time segments and eight threads.

Factor	**First thread**	**Last thread**
Number of processed log records	1,026,750	1,066,334
Number of computed serial schedules	1,105,858	1,479,454
Number of merged serial schedules	11,745	50,595
Number of created candidate database states	1,149,412	1,771,707

Table 5.5 shows the scale-up with different numbers of threads relative to one thread. Polygraph scales sublinearly with increasing numbers of threads. The overhead of Java's garbage collector limits the scalability of Polygraph. This study used Java's GarbageCollectorMXBean interface [6] to measure the approximate garbage collection time in seconds with 32 time segments (see Table 5.6). The garbage collection time increases with an increase in the number of threads.

When running the threads as processes (eight processes instead of eight threads) on different nodes, the scalability of Polygraph is improved (see Table 5.7). For example, with 32 time segments and eight threads, running the threads on eight

Table 5.5: Scale-up relative to one thread with different numbers of validation threads and time segments.

Number of threads	**Scale-up with 8 segments**		**Scale-up with 16 segments**		**Scale-up with 32 segments**	
	First	Last	First	Last	First	Last
1	1	1	1	1	1	1
2	2.14	1.80	2.03	1.96	2.03	1.98
4	4.41	2.76	4.15	3.83	3.83	3.56
8	7.38	3.21	7.06	4.45	6.62	5.68

Table 5.6: Approximate time of Java's garbage collector in seconds with 32 time segments and different numbers of threads.

Number of threads	**Approximate garbage collection time (sec)**
1	46
2	50
4	69
8	101

nodes improved the scalability from 5.68 to 7.1. This reflects the fact that the complexity of memory management and garbage collection is reduced as the program is divided into eight java processes instead of one. Moreover, having more nodes increases the CPU resources to the JVM garbage collector.

Table 5.7: Scale-out relative to one node with different numbers of nodes and time segments.

Number of nodes	**Scale-out with 8 segments**		**Scale-out with 16 segments**		**Scale-out with 32 segments**	
	First	Last	First	Last	First	Last
1	1	1	1	1	1	1
2	1.99	1.70	2.18	2.04	2.03	1.96
4	4.65	3.02	4.55	4.27	4.06	3.84
8	9.77	4.55	9.26	7.79	8.1	7.1

Correctness

This temporal partitioning of log records enables Polygraph to scale. It compromises the correctness of Polygraph by resulting in both false positives and false negatives, however. When a thread starts processing a time segment, an anomalous read may establish the value of an entity/relationship. This is a false negative. It may cause Polygraph to mark a subsequent (correct) read as anomalous, resulting in a false positive.

This experiment evaluated the correctness of Polygraph with a trace of the SEATS benchmark that has 832 anomalies. The experiment ran Polygraph constructing different numbers of time segments (see Table 5.8). Increasing the number of time segments increases the number of false positives and negatives. This is because establishing the values of entities/relationships is increased with more time segments.

Table 5.8: The number of detected false negative and false positive anomalies increases with a higher number of time segments.

Num segments	Num detected anomalies	Num false negatives	Num false positives	Total error
2	824	9	1	10
4	818	16	2	18
8	826	28	22	50
16	792	51	11	62
32	752	91	11	102

Next, the author repeated the above experiments but terminated Polygraph as soon as one thread completed validation. Figure 5.9 shows the numbers with 8, 16 and 32 time segments. The number of false negatives has increased due to the early termination of Polygraph. The total number of false negatives and positives has not decreased because they occur at the beginning of a time segment when the values of entities/relationships are established. Consequently, they are not

minimized when terminating Polygraph as soon as one thread completes validating its assigned segments.

Num threads	Num detected anomalies	Num false negatives	Num false positives	Total error
2	817	37	22	59
4	803	50	22	72
8	810	44	22	66

(a) 8 time segments

Num threads	Num detected anomalies	Num false negatives	Num false positives	Total error
2	729	113	11	124
4	734	108	11	119
8	838	104	11	115

(b) 16 time segments

Num threads	Num detected anomalies	Num false negatives	Num false positives	Total error
2	747	96	11	107
4	652	190	11	201
8	631	211	11	222

(c) 32 time segments

Table 5.9: The number of detected, false negative and false positive anomalies when terminating Polygraph as soon as one thread finishes with different numbers of time segments.

In sum, temporal partitioning enhances scalability of Polygraph with a higher number of time segments by reducing its accuracy.

Detecting False Positives and Negatives: The author investigated a verification technique to analyze detected false positives and negatives in temporal segments other than the first, i.e., S_i where $i \geq 1$. The objective was to stitch together the results obtained from different segments to eliminate false positives

and negatives, providing 100% accuracy. With the SEATS benchmark, the verification technique proved too slow, eliminating the scalability benefits of temporal segmentation. The verification technique used must process a large number (73%) of log records and re-validate them. The main insight was that a significant amount of overlap exists between different transactions referencing different entities/relationships. This renders the verification process complex, requiring a dependency graph to identify the entities and relationships that must be re-validated. Inherently, this many-to-many relationship limits scalability.

Chapter 6

Extensibility

Polygraph is extensible to support custom-application requirements. This chapter describes the software architecture of Polygraph's Authoring, Validation and Monitoring components (see Figure 3.1), the goal being to illustrate how a developer may extend these components and to demonstrate the extension of Polygraph to support range predicates and report the freshness confidence metric along with the number of anomalies.

6.1 Authoring Component

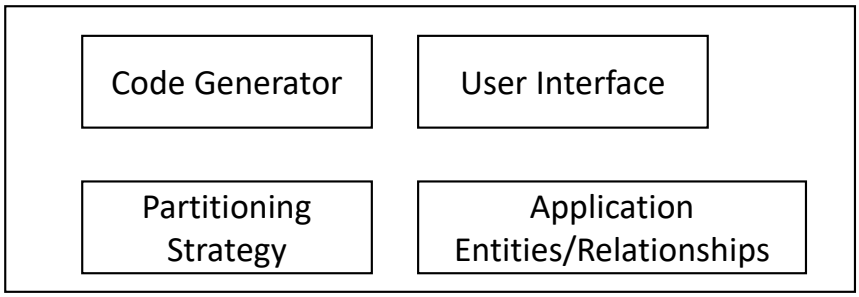


Figure 6.1: Software Architecture of Polygraph's Authoring Component.

Polygraph Authoring has four software components, as shown in Figure 6.1. The following will describe each in turn.

- Code Generator: generates code snippets that create and publish log records for transactions; it currently supports Java. A developer may extend it to support other programming languages, such as Python or C++.

- Partitioning Strategy: computes partitioning attributes and strategy for log records.
- User Interface: enables users to provide the ER diagram of the application and its transactions.
- Application Entities/Relationships: maintains entities/relationships of the application and how transactions manipulate them.

6.2 Validation Component

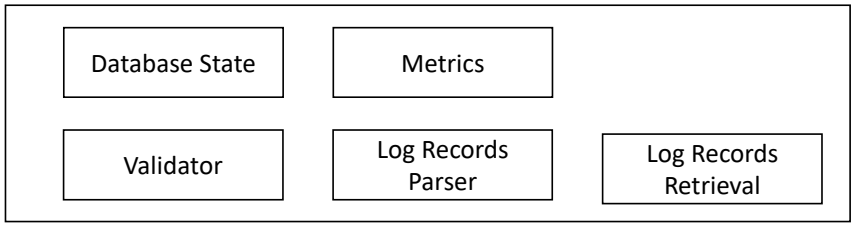


Figure 6.2: Software Architecture of Polygraph's Validation Component.

Polygraph Validation has the following five software components (see also Figure 6.2):

- Database State: maintains candidate values and serial schedules for entities and relationships.
- Validator: validates transactions and computes serial schedules, currently computes strict serial schedules. A developer may extend it to support alternative consistency models, such as serializability [58].
- Metrics: compute reported metrics as Polygraph processes log records.

- Log Records Parser: parses retrieved log records and currently supports string as format for log records. A developer may extend it to support JSON or XML formats.

- Log Records Retrieval: retrieves log records from the streaming platform and currently supports Kafka. A developer may extend it to support other streaming platforms, such as RabbitMQ [61].

6.3 Monitoring Component

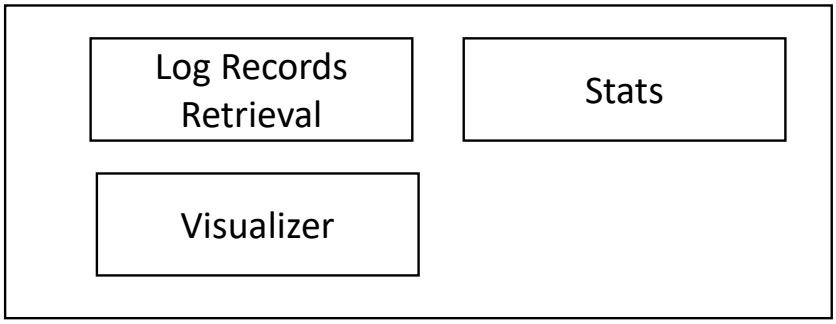


Figure 6.3: Software Architecture of Polygraph's Monitoring Component.

Polygraph Monitoring has the following three software components (see also Figure 6.3):

- Log Records Retrieval: retrieves log records from the streaming platform to visualize transactions and currently supports Kafka. A developer may extend it to support other streaming platforms, such as RabbitMQ [61].

- Stats: gathers stats and reported metrics to view them.

- Visualizer: visualizes stats, reported metrics, and anomalous transactions.

The following two sections demonstrate Polygraph's extensibility by extending it to support range predicates and report freshness confidence.

6.4 Supporting Range Predicates

A range predicate references an attribute of a data item and specifies either (a) a lower bound and/or an upper bound for its values using mathematical comparison operators, e.g., 10 < temperature < 12, or (b) a lower or an upper bound with a limit on the number of records to retrieve (e.g., temperature > 10 limit 3, temperature < 11 limit 5, etc.).

This study presents two approaches to support range predicates, both of which extend Polygraph's Database State component (see Figure 6.2). The first approach extends Polygraph to use an order-preserving main-memory data structure, while the second uses a data store to process queries. The first is faster and has less overhead because it uses a main-memory data structure, while the second is more general and extendable to support additional query types.

6.4.1 Main Memory Data Structure

This approach uses an order-preserving data structure. Specifically, Java's TreeMap (sorted hashmap) served as a main-memory indexing data structure. It was implemented as a Red-Black Tree, which is a self-balancing Binary Search Tree. It is based on NavigableMap implementation [7]. This tree supports retrieving nodes greater than a lower bound and/or smaller than an upper bound. The time complexity of this operation is O(log n+k) where n is the number of nodes in the tree and k is the number of nodes satisfying the range predict. The tree provides

a guaranteed log(n) time cost for insert, delete, and get operations [7]. This tree map is queried to validate range predicates and simple analytics, such max or min.

6.4.2 Data Store

This approach uses a document data store, namely MongoDB. The author opted for a document, instead of a relational, data store to efficiently represent multi-valued properties, such as a friends list. To represent an entity/relationship with a multi-valued property, a relational model requires an additional table that repeats the entity/relationship key with each value as a separate row. This requires one insert for each value and increases the number of rows to be joined to retrieve the values of the property of an entity/relationship. With a document store, a multi-valued property can be represented as an array.

With MongoDB, each entity/relationship appears as a document along with its properties as fields of the document. This approach requires maintaining the documents stored in MongoDB in sync with the main memory state of entities/relationships. This process imposes a significant overhead on Polygraph that impacts its performance.

An advantage of this approach is that it is general and can be used to support a wide variety of queries. It enables Polygraph to validate queries that can be modeled in MongoDB query language. This includes range predicates and simple analytics, such as max, min, average or sum. Additionally, it supports boolean combination of exact match and range predicates over multiple properties of entities/relationships.

6.4.3 Evaluation

The present implementation and evaluation focused on queries with range predicates. The author evaluated the first approach, which uses an order-preserving data structure with the YCSB benchmark. The workload consists of 95% scan and 5% update actions with uniform distribution and 10 Million users. The scan action retrieves users greater than or equal a lower bound with limit on the number of records to be retrieved. Scan/update actions are configured to read/update all fields of a user. The author fixed the number of records to be retrieved with a scan action to 5 and modified the Partitioning Strategy component of Polygraph Authoring to range partition users across 1000 Kafka partitioned logs. An update action generates one log record that is published to the partition of its referenced user. A scan action generates one log record that may reference up to five users. If a scan log record references users spanning across more than one partition, the scan log record is replicated across these different partitions. A scan log record published to a Kafka partition contains only the set of users belonging to that partition. For example, with two partitions P1 and P2, P1 contains users from 1 to 3 and P2 contains users from 4 to 5 (see Figure 6.4). A scan log record referencing users from 1 to 5 is replicated to partitions P1 and P2 as follows. The scan log record published to partition P1 contains only users 1, 2 and 3. The scan log record published to partition P2 contains only users 4 and 5.

The study used MySQL v5.6 configured with a 48-GB buffer pool and extended with one IQ-Twemcached server [5]. The cache server ran 16 IQ-Twemcached instances, each with 3 GB of memory. All nodes were Emulab [74] d430. Each node was a Dell Poweredge R430 with 64 GB of memory, a 10-Gbps networking card and two 2.4 GHz 64-bit, eight-core Xeon E5-2630v3 processors. The YCSB client described in [42] served to cache scan queries, providing strong consistency.

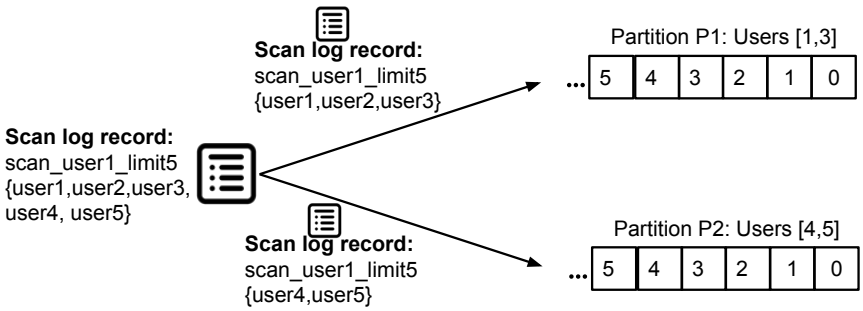


Figure 6.4: Range partitioning of a scan log record.

This resulted from employing an interval tree as a lock manager to synchronize concurrent scan and update actions. The experimental design called for write-back as the caching consistency technique to generate higher throughput [42]. Fifteen YCSB clients with 1000 threads issued actions for five minutes, producing a total of 24.4 and 1.28 Million scan and update actions, respectively. NTP served to synchronize clients' clocks [55]. Furthermore, expanding the start and commit times by 2 ms adjusted for clock skew. This approximates that all detected anomalies are true anomalies. A Kafka cluster consisted of five nodes hosting five brokers and one Zookeeper instance. The running of 17 Polygraph servers with 1000 validation threads resulted in no detected anomalies. The validation time was 2 minutes and 46 seconds.

To produce anomalies, the author repeated the above experiment disabling the lock manager and without checking for buffered writes to be applied to the database in case of a cache miss. Polygraph validated a total of 26.97 and 1.42 Million scan and update actions, respectively, in 3 minutes and 5 seconds. It detected 214,523 anomalies, which is about 0.8% of scan actions.

Next, generating a workload mix consisting of 90% scan, 5% insert and 5% delete actions against MySQL ensured that Polygraph validated ranges with holes correctly. Additionally, repeating the same experiment representing the user ID as

string tested ranges with a different data type. In both experiments, Polygraph detecting no anomalies as expected.

6.5 Reporting Freshness Confidence

Freshness confidence [24] is the probability of observing the freshest value for a data item up to t units of time after the last write referencing it. Freshness confidence is useful for applications that tolerate anomalous values for some amount of time. For example, threaded conversation applications may tolerate a delay on receiving the latest replies for at most one second. For these applications computing freshness confidence is more useful than quantifying the number of anomalies.

To compute freshness confidence, the author extended Polygraph's Database State component to maintain the commit time of the latest write for every entity/relationship and a hashmap of buckets, each with a duration of d time units that is configured by the user. Each bucket has an index (i) and stores the number of valid reads ($V_b i$) and the total number of reads ($R_b i$) corresponding to its interval of time from $i * d$ to $(i + 1) * d$ time units. When validating a read transaction, Polygraph identifies its corresponding bucket by subtracting the commit time of the latest write before the read from the start time of the read. Next, it divides that value by d to get the bucket index that represents its key in the hashmap. The probability of observing the freshest value for a read at most t units (P_t) after the write is computed by finding $i = Ceil(t/d - 1)$ and computing the following:

$$P_t = \frac{\sum_{j=i}^{j=numBuckets-1} V_{bi}}{\sum_{j=i}^{j=numBuckets-1} R_{bi}}$$

When launching Polygraph servers, the user provides the desired bucket duration (d). There is no change to Polygraph code snippets. Polygraph computes the freshness confidence for all entities/relationships as it processes read and write log records. Moreover, the author extended the Visualizer and Stats components of Polygraph monitoring tool to collect stats and provide visualization for freshness confidence, as seen in Figure 6.5.

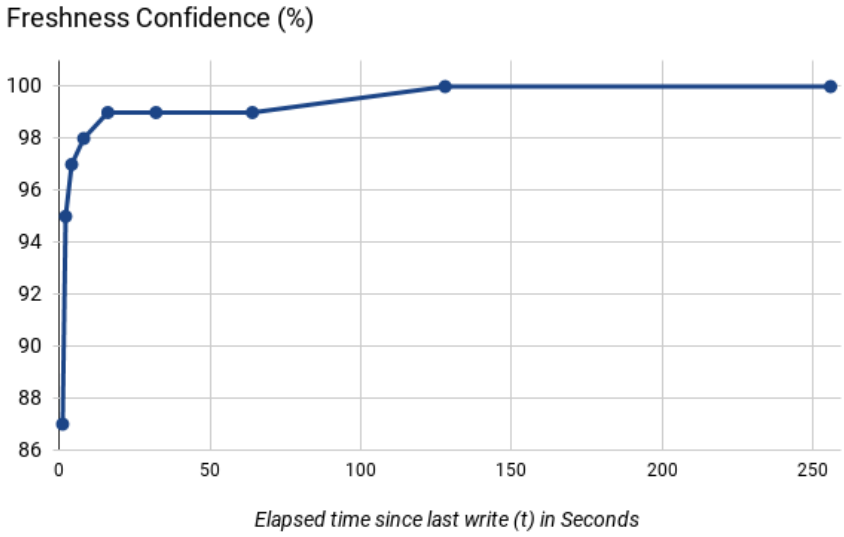


Figure 6.5: Computed percentage of freshness confidence while varying the elapsed time after write value.

Figure 6.5 shows the freshness confidence for MySQL extended with Twemcached [12]. The author ran BG with 10,000 members, 100 threads, and a 90% read workload for 10 minutes. Write actions are extended to invalidate their impacted key-value pairs in the cache. This configuration produces anomalies due to undesirable cache race conditions between concurrent read and write actions [44]. 72% of anomalies occur within one second since the last write. This is due to the Zipfian distribution where popular members are referenced more frequently by concurrent read and write actions. Consequently, anomalies occur and these frequent reads and writes reduce the duration since the last write. The freshness confidence of

reads is improved, as the duration since the last write (t) increases. This is because more anomalies are not counted as they become tolerable by the computed freshness confidence for the application.

6.6 Supporting Consistency

Consistency constraints, or application invariants, such as balance ≥ 0 are application specific. A developer may extend the Authoring tool that customizes Polygraph servers with application-specific code snippets to verify consistency constraints, such as balance ≥ 0, or salary $> 50{,}000$. Other consistency constraints may require the developer to extend the Validator component to verify them, such as sales of a warehouse = sum sales of its districts.

Chapter 7

Overhead of Polygraph

This chapter reports on the overhead imposed by Polygraph code snippets on the application and the memory overhead of Polygraph servers when validating log records.

7.1 Overhead on the Application

Polygraph code snippets embedded in the application generate log records for transactions and publish them to a Kafka cluster. This network overhead is a function of concurrent transactions executed per unit of time (T), mix of transactions, and the size of log records generated by each transaction. The author quantified the network overhead in bytes/second as follows:

$T * \sum_{t \in transactions} frequency(t) \times logSize(t)$.

Table 7.1 shows an example of computing the network overhead of Polygraph with TPC-C and BG. For example, with TPC-C and a throughput of 10,000 transactions/sec, the minimum, average, and maximum network overhead is 5.2, 5.6, and 5.99 MB/second, respectively. This is insignificant with today's high speed (10 Gbps) networking cards.

To experimentally evaluate the overhead of Polygraph on the application, the author ran BG with one client, 100 threads, 10,000 members and the workload in Table 5.1 using MySQL for one hour with and without generating and publishing

log records to one Kafka server [1]. BG's throughput with and without generating log records are 4,307 and 4,300 actions per second, respectively. This is because the network is not the bottleneck.

Table 7.1: An example of computing network overhead of Polygraph in MB/second for BG and TPC-C benchmarks.

Benchmark	**Throughput (transactions/sec)**	**Min**	**Avg**	**Max**
BG	10,000	0.76	0.83	0.9
TPC-C	10,000	5.2	5.6	5.99

* With BG, the author used the mix of actions shown in Table 5.1. Accept Friend and Thaw Friendship actions impact two members, manipulation of each member is represented as a different log record.

7.2 Memory Overhead of Polygraph Servers

Polygraph builds a conceptual database for the application to validate transactions by processing log records. Polygraph processes log records in a read-driven manner. It retrieves the read transaction with the earliest start time. Next, it retrieves all write transactions with a start time less than or equal to the commit time of the read. Write transactions manipulating an entity/relationship referenced by the read are processed. Write transactions referencing other entities/relationships are kept in memory until they are referenced by a read transaction. Those write transactions are termed *unprocessed writes*. This approach prevents the unnecessary creation of entities/relationships and serial schedules, as those entities/relationships may never be referenced by a read transaction. The memory overhead of unprocessed writes is a function of write transactions.

[1]The same experiment was repeated with a single thread of BG producing a throughput of 790 and 815 actions per second with and without generating log records, respectively.

The next section reports on the memory overhead of Polygraph using BG and TPC-C benchmarks. Table 7.2 shows a summary of the memory overhead of Polygraph with these benchmarks. The size of the conceptual database that Polygraph maintains is far smaller than the physical database size on file.

Table 7.2: Memory overhead of Polygraph with BG and TPC-C.

Benchmark	Entities/ relationships (MB)	Serial Schedules (MB)	Unprocessed Writes (MB)	Initial Database Size on File
BG	2.5	7.8	1.12	134 MB
TPC-C	459	348	646	11 GB

7.2.1 BG

BG was run with 10,000 members, 100 threads, Zipfian mean 0.27 and 90% read workload for one hour. BG is represented with one entity set that has two properties: friends and pending requests counts. The number of entities is 10,000. The average memory overhead for entities is 2.5 MB. The average size for one entity is 270 bytes. The maximum and minimum sizes for one entity are 280 and 208 bytes, respectively. A lower bound for entities memory overhead is 1.98 MB and an upper bound is 2.67 MB. The number of candidate values for entities depends on the number of overlapping write transactions and the number of read transactions.

The average memory overhead of candidate serial schedules is 7.8 MB for 9,881 schedules. The average memory overhead of unprocessed writes is 1.12 MB for 1,682 write log records. Table 7.3 shows the size of log record objects for BG's write actions. The number of properties varies for an Accept Friend Request because it generates one log record for each participating member. The number of properties for the log record for the inviter is one and the number of properties for the invitee is two.

Table 7.3: Size of BG's write actions log record object in bytes.

Write Action	Min	Avg	Max	Number of properties
Invite Friend	496	509	512	1
Accept Friend Request	488	569	640	min=1, avg=1.5, max=2
Reject Friend Request	488	501	504	1
Thaw Friendship	496	509	512	1

7.2.2 TPC-C

This study ran TPC-C with 100 warehouses and 300 threads for three hours with the default workload. TPC-C is represented with three entity sets and one relationship set. The entity sets are Customer, Order and District. The relationship set, Last order, represents a relationship between a customer and its last order. Table 7.4 shows the size of TPC-C's entities and relationship in bytes.

Table 7.4: Size of TPC-C entities and relationship sets in bytes.

Entity/Relationship set	Min	Avg	Max	Number of properties
Customer entity set	216	336	352	3
Order entity set	232	325	352	3
District entity set	208	223	224	1
Last order relationship set	240	267	336	1

The number of customer entities is the number of warehouses multiplied by 30,000. The number of threads, duration of running the benchmark, and the speed of the underlying system dictate the number of order entities. The number of district entities is the number of warehouses multiplied by 10. The number of last order relationships equals the number of customer entities. The average number of TPC-C entities and relationships is 1,359,360. It is difficult to compute lower and upper bounds on the memory size of entities/relationships because the max number of order entities depends on each experiment. The number of candidate values for

entities/relationships depends on the number of overlapping write transactions and the number of read transactions.

The average memory overhead of candidate serial schedules is 348 MB for 286,406 schedules. The average memory overhead of unprocessed writes is 646 MB for 304,141 write log records. Table 7.5 shows the size of log record objects for TPC-C's write transactions.

Table 7.5: Size of TPC-C's write transactions log record object in bytes.

Write transaction	**Min**	**Avg**	**Max**	**Number of properties**
Payment	1136	1160	1184	6 (1 entity)
New-Order	1464	1492	1504	6 (2 entities and 1 relationship)
Delivery	7480	7591	7632	30 (20 entities)

Chapter 8

Testing Database Applications

This chapter relates experiences using Polygraph to test different application classes, the purpose being to demonstrate its usefulness to verify data consistency of an implementation to provide strong consistency. The chapter will also show how Polygraph has enabled software developers to focus on improving designs to handle complex scenarios while gaining confidence about the correctness of implementation. Finally, the chapter will describe the successful use of Polygraph to detect software bugs, verify strong consistency, detect overlooked race conditions, and improve application designs to handle unexpected scenarios that impact consistency of data.

The chapter describes use cases conducted by the author with Polygraph relating to two major application classes: e-commerce and cloud. A use of Polygraph with a simple financial application will illustrate the former, uses of Polygraph with a variety of applications representing different workloads will illustrate the latter.

8.1 E-Commerce Applications

E-commerce applications implement a variety of transactions such as making a payment or placing a new order. These transactions may read and/or write many data items. Moreover, they may have key dependencies that prevent determining their referenced data sets in advance. Verifying correctness of transactions

is complex as they must provide Atomicity, Consistency, Isolation and Durability semantics.

An experimentalist may use Polygraph to model a transaction at a conceptual level. It provides the flexibility to include multiple reads, updates, deletes and/or inserts that reference different entities/relationships constituting a transaction. These entities/relationships may not be known in advance, or have key dependencies. The following describes how Polygraph can verify the correctness of an implementation of a simple financial application.

8.1.1 A Financial Application

The author implemented a specific aspect of a financial application that, at a conceptual level, performs money withdrawal transactions from the same account. Each transaction deducts $100 and is physically implemented as two database transactions. The first transaction reads the available account balance and the second transaction updates the account balance after deducting $100. MySQL served as the back-end data store, configured to be ACID-compliant.

Polygraph represents the application as one account entity. This entity has one property representing the account balance with the account number as its primary key. Polygraph is a conceptual tool that enables a system designer to represent the money withdrawal as one transaction independent of its physical implementation. The author represented it as one read-write transaction. It reads the account balance and deducts $100 from its value. The author differentiates between the conceptual transactions, termed *ConXacts* and their physical implementation, termed *PhyXacts*.

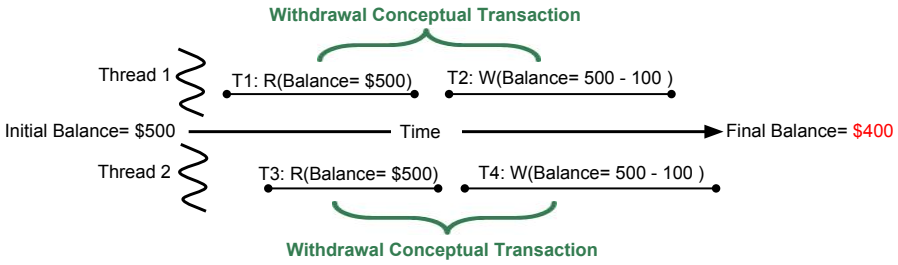


Figure 8.1: Two threads execute money withdrawal transactions concurrently which results in an anomaly.

This study used Polygraph to test the correctness of the implementation, PhyXacts, with respect to their conceptual abstraction, ConXacts, of the money withdrawal transaction. Although MySQL is configured to be ACID complaint, the application is not. Polygraph detects no anomalies with a single-threaded execution of the application. With multi-threaded execution, Polygraph detects anomalies. These anomalies occur due to the incorrect PhyXact of money withdrawal as two transactions, instead of one. Figure 8.1 shows an example of two concurrent threads performing two money withdrawals. There is no serial schedule for their interleaved execution that violates the Isolation property. These anomalies cause lost money. In Figure 8.1, the amount of lost money is $100 as the final account balance reflects one money withdrawal transaction, instead of two.

Polygraph visualizes concurrent conceptual transactions (ConXacts) to help reason about the cause of anomalies (see Figure 8.2). Each ConXact appears as a rectangle with a unique ID. The rectangle defines the start and the commit times of a ConXact relative to the elapsed time from the start of the application. ConXacts producing anomalies are red. Figure 8.2 shows eight ConXacts, five being anomalous. As the user puts her mouse over a money withdrawal ConXact, Polygraph shows its details. The observed account balance and the withdrawal amount appear here. Also shown is the expected account balance values for the

anomalous ConXacts. Figure 8.2 shows ConXact 5 observing the value $99,999,700 while its expected values are either $99,999,500 or $99,999,600. The two vertical dashed lines highlight the start and commit times of ConXact 5.

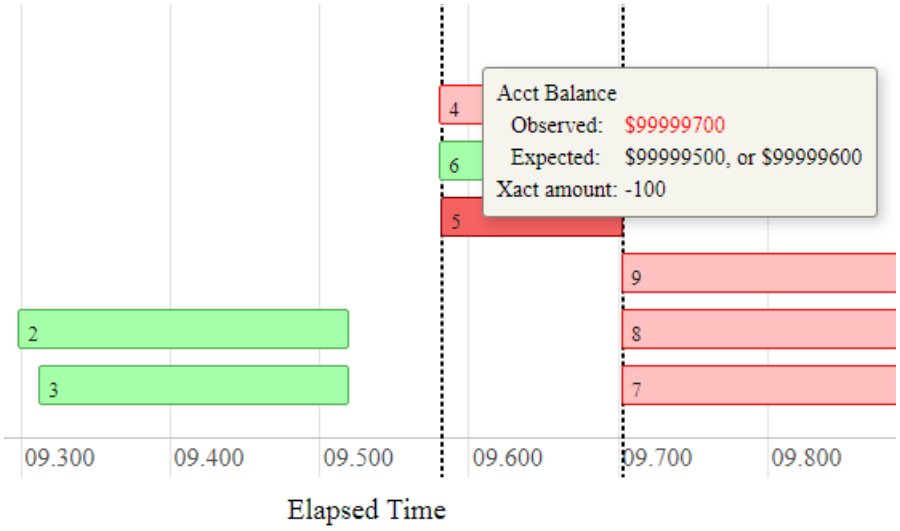


Figure 8.2: Polygraph monitoring tool visualizes concurrent money withdrawal conceptual transactions.

8.2 Cloud Applications

Cloud OLTP applications require 24/7 availability and process a large number of concurrent operations – e.g., Facebook receives billions of operations per second. Traditional SQL systems focus on correctness and consistency at the expense of availability and performance. Numerous cloud storage NoSQL systems have been designed in the last two decades to support cloud OLTP applications. NoSQL systems are drastically different from SQL systems in that they sacrifice consistency for high availability and scalability. Data is usually partitioned across many storage nodes and replicated to tolerate failures and sustain a high performance. NoSQL systems have limited support for read-only, write-only or single-row transactions.

Examples of NoSQL systems include key-value stores (Memcached), document stores (Mongodb) and extensible record stores (BigTable) (see [32] for details).

In the last decade, the rise of NewSQL systems, such as Google's Spanner [35], aims to provide semantics of SQL while achieving high availability, scalability and full transaction support. NewSQL has emerged since 1) many applications cannot afford to use NoSQL systems for weaker consistency (financial applications, political applications), 2) organizations find out that weaker consistency also decrease their developer's productivity since they spend more time handling inconsistent data in their code [35, 53].

NewSQL systems face many challenges in ensuring consistency: 1) tolerating arbitrary node failures and asymmetric network partitions, 2) synchronizing between different replicas in different geolocations, and 3) preserving consistency with frequent configuration changes.

Polygraph complements formal verification tools such TLA+ [77] in that it tests the correctness of the implementation. Polygraph also complements failure injection frameworks [18] in that it is an end-to-end testing suite. Polygraph eases the development of these NewSQL systems because developers can focus on the design itself and leave the correctness testing to Polygraph.

Database-as-a-Service providers release new features from development to production, undergoing several testing stages: 1) code compilation, 2) unit testing, 3) integration testing, and 4) gamma testing. Code compilation ensures that new features added to the software compiles successfully. Unit testing ensures functions implemented in each class are correct. Integration testing evaluates the interaction between different components in the system. Gamma testing is the last stage that tests the new software in an emulated production environment.

The author envisions the offline Polygraph can be integrated into existing testing pipelines between integration testing and gamma testing to ensure new software features do not impact the correctness of the database. The online version of Polygraph complements the gamma testing in that it quantifies the number of anomalies in real time.

The following describes Polygraph's extensions to benchmarks that focus on the performance of cloud applications with correctness testing – e.g., Yahoo's Cloud Serving Benchmark [34] (YCSB) and BG [24] (a social networking benchmark). Table 8.1 shows the representation of YCSB and BG in Polygraph. YCSB has a variety of workloads that represent different applications. The following also discusses how Polygraph can be used to test three typical cloud applications: session stores, photo tagging and messaging applications.

Table 8.1: Conceptual representation of three benchmarks.

Benchmark	**Transaction name**	**Actions**
BG	View Profile	Read a member entity
	List Friends	Read a member entity
	View Friend Requests	Read a member entity
	Thaw Friendship	Update two member entities
	Reject Friend Request	Update a member entity
	Accept Friend Request	Update two member entities
	Invite Friend	Update a member entity
YCSB	Read	Read user entity
	Delete	Delete user entity
	Modify	Read and Update user entity
	Update	Update user entity
	Scan	Read multiple user entities
	Insert	Insert user entity

8.2.1 Session Store

A typical web application hosted on the cloud relies on a session store to record recent actions of a user. A session starts when the user logs in and ends when the user logs out. A session records information of a user – e.g., user profile, personalized data and behaviors. Therefore, a session store represents a write-heavy workload (50% read and 50% update). Cache Augmented Database Systems (CADSs) employ the write-back policy that is designed to enhance performance of an application with a write-heavy workload.

With write-back policy, a write request is acknowledged once it updates its in-memory copy and appends its delta change in memory (buffered writes). Worker threads merge these buffered writes and apply them to the data store asynchronously. In other words, the write-back policy removes the data store from the critical path during executing writes, enhancing performance and scalability. When implementing the write-back policy with CADS, this study used Polygraph to detect overlooked race conditions with concurrent threads and verify that buffered writes were applied correctly to the data store before querying it in the case of a cache miss.

Using Polygraph, the author detected the following bugs in the implementation. First, worker threads were merging buffered writes incorrectly, which corrupts their state. Second, the hashcode to store buffered writes was computed incorrectly, which causes them to be stored in the wrong cache server.

8.2.2 Photo Tagging Application

Photo tagging applications, such as [11], enable users to tag and display all tags of a photo. Adding a tag is a write action. Reading all tags of a photo is a read action. A photo-tagging application represents a read-heavy workload.

Systems in support of this application employ caches to enhance its performance characteristics [56].

Cache server failures are inevitable with a system at this scale. When a cache server fails, its in-memory content is lost, which degrades performance due to cache misses. With the advent of non-volatile memory (NVM), a cache server preserves cached content in NVM after a failure and reuses these data items upon its recovery. However, some data items may be updated during the failure and become stale. This study used Polygraph to quantify the stale reads per second an application observes after a cache server failure. A significant number of stale reads that peaked right after the cache server's recovery was observed (see Figure 8.3). This motivated the design and build of a caching layer, Gemini, that preserves consistency in the presence of cache server failures.

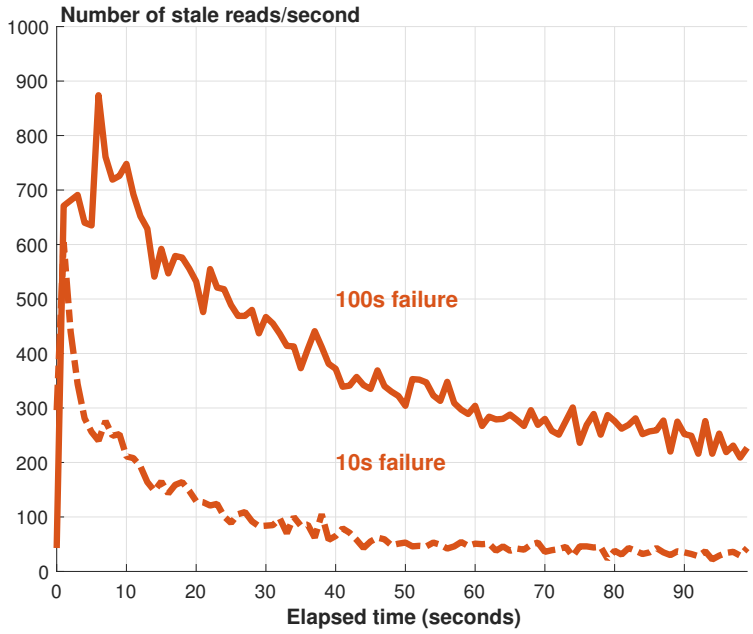


Figure 8.3: Number of stale reads per second after recovery from a 10 and a 100 second failure.

Polygraph verified that Gemini provides strong consistency in the presence of arbitrary cache server failures. This is to ensure the correctness of the design and its implementation. Gemini incorporated Polygraph's provided code snippets with minimal effort – less than 100 lines of Java code. Polygraph's obliviousness to how an action is executed greatly simplified the validation of Gemini's correctness. It helps one to focus on sophisticated failure and recovery scenarios – e.g., concurrent cache server failures, simultaneous recovery of multiple cache servers, failures of cache servers during the recovery of a cache server. All these failures were emulated and Gemini's correctness was validated without changing Polygraph or its code snippets to generate log records. In the early implementation of Gemini, Polygraph helped to detect undesirable race conditions that caused a recovering cache server to generate stale values. It also allowed us to make fast iterations by applying many optimizations to Gemini while ensuring its correctness.

Cache servers also consume a significant amount of power even during off-peak hours. Thus, database systems employ an auto-scaling component that elastically increases or decreases the number of cache servers to accommodate the system load. When data items are assigned to a different cache server, future read and write requests are directed to this new cache server. Old copies of data items are still maintained since discarding them may not be supported [21]. When these data items are assigned back to the previous cache server, the application may produce stale read anomalies due to the existence of previous old copies of data items. Here, Polygraph quantified the stale reads in real-world traces [36, 22], simulating configuration changes and an imposed system load. It motivated the design of a scalable online technique, Rejig [43], that manages key-to-cache server assignment changes while providing strong consistency.

8.2.3 Messaging Application

Messaging applications, such as WhatsApp [13], allow users to communicate with each other (e.g., threaded conversations). A user may scan all conversations in a message thread. The workload of a messaging application is different from the above applications in that it requires range queries. Caching these queries and looking them up maybe faster than computing them using an index structure. This motivated building RangeQC [42], a framework for caching range predicate query results. It provides strong consistency and supports write-around, write-through and write-back policies. It caches the results of range predicates using a set of interval trees. It consists of one or more Dendrites to process range predicates and facilitate communication with a cache manager such as memcached and a data store such as MySQL or MongoDB.

To validate a scan action, e.g., YCSB's scan, Polygraph was extended with a tree map (a sorted hash map based on the key) to validate range predicates. A serial schedule in Polygraph is extended to maintain a tree map of its referenced entities/relationships. This tree map is queried to validate entities returned by a scan action.

Polygraph detected an anomaly with an implementation of the interval tree component that was not returning the correct result of lookups. This testing was done with a single thread issuing YCSB's scan and update actions. The update action was extended to invalidate impacted K-V pairs in the cache by looking up an interval tree. A scan action was reading a stale value for a key in the cache due to interval tree lookups not returning the correct results of keys to be invalidated. Consequently, the incorrect implementation of the interval tree component was replaced with another one that produced no anomalies with a single thread.

Polygraph detected many anomalies in the initial implementation. Most were attributed to undesirable race conditions. Here, two examples are presented. First, a lock manager was maintained for ranges of values which were released prior to updating the database in an update action. As a result, a different thread was able to place a stale value in the cache. A cache hit for these values would cause Polygraph to detect an anomaly. Second, with multiple cache servers, a corner case is when an update action impacts multiple keys across multiple servers. The commit for the update was not atomic. One cache server committed the update before the other participating servers. As a result, two scan actions that executed subsequent to the update would be served as follows. One scan observed the update because it executed on the server that committed the update. The second scan would not observe the update even though it started after the commit time of the first scan action. This scan was executed using the server that had not committed the update. Polygraph proved useful in detecting these implementation errors, which consequently were fixed by committing updates across multiple cache servers atomically.

Chapter 9

Future Work

Polygraph is a plug-and-play framework to quantify anomalies. There are two main future research directions. The first would extend Polygraph to provide explanations for anomalies. The second would use Polygraph to quantify anomalies with blockchain applications. The following sections describe these future research directions.

9.1 Explanation For Anomalies

Polygraph detects anomalies and visualizes read transactions producing them. For each anomalous transaction, Polygraph shows the transactions that read/wrote the entities/relationships referenced by the violating transaction. This may be insufficient to reason about and understand the cause of anomalies. Moreover, all detected violations are considered as anomalies, without any classification.

A future research direction is to provide explanation for anomalies. One possible solution is to extend Polygraph with adapters to provide patterns of transaction interactions that correspond to different types of anomalies. For example, one adapter may provide patterns for lost update anomalies. Another adapter may correspond to a consistency setting of a SQL system such as Read Committed. These adapters can help to classify and detect the cause of an anomaly.

9.2 Polygraph and Blockchain

Blockchain [68] is a distributed ledger technology used by diverse applications, such as financial, supply chain, health, etc. It provides a decentralized control, and its underlying database layer may support SQL and NoSQL data stores.

A blockchain solution may produce anomalies due to the use of a data store with weak consistency guarantees and/or improper configuration that results in weak consistency. For example, configuring Bitcoin client with a low confirmation number, such as 1, to consider a transaction as committed may result in reversing committed transactions [30, 76, 39]. In Bitcoin, the confirmation number of a block represents the number of blocks submitted to the network after that block. The higher the confirmation number, the more likely the block is not going to be discarded. Bitcoin recommends waiting for at least six confirmations to consider transactions in a block as committed. So, if a user configures the Bitcoin client with a low confirmation number to consider a transaction as committed, this transaction may get reversed due to a fork in the chain. Forks in the chain occur for a variety of reasons [51]. For example, two blocks may be submitted to the network at the same time (see Figure 9.1). Branch B is discarded as participants eventually agree on Branch A, which has more blocks. As a result, transactions in block 2-B are reversed. Polygraph's plug-and-play and application agnostic features make it suitable to quantify anomalies for diverse blockchain applications.

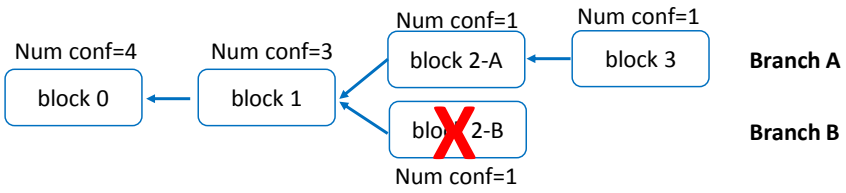


Figure 9.1: A fork in the blockchain causes one of the branches to be discarded

Reference List

[1] Amazon DynamoDB FAQs-Amazon Web Services (AWS), https://aws.amazon.com/dynamodb/faqs/.

[2] Another Bitcoin Exchange Bites the Dust - NBC News. https://www.nbcnews.com/business/markets/another-bitcoin-exchange-bites-dust-n43981. (Accessed on 10/01/2017).

[3] Flexcoin Robbery - Business Insider. http://www.businessinsider.com/flexcoin-robbery-2014-3. (Accessed on 09/20/2017).

[4] Infinispan. http://infinispan.org/. (Accessed on 05/01/2018).

[5] IQ-Twemcached. http://dblab.usc.edu/users/IQ/.

[6] Java Garbage Collector. `https://docs.oracle.com/javase/7/docs/api/java/lang/management/GarbageCollectorMXBean.html`. (Accessed on 03/20/2018).

[7] Java TreeMap. `https://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html`. (Accessed on 03/20/2018).

[8] M. Inc. MongoDB. https://www.mongodb.com/.

[9] MySQL: Consistent Nonlocking Reads. `https://dev.mysql.com/doc/refman/8.0/en/innodb-consistent-read.html`. (Accessed on 05/01/2018).

[10] MySQL: Isolation Levels. https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html. (Accessed on 05/01/2018).

[11] Pinterest. `https://www.pinterest.com/explore/tag-photo/?lp=true`. (Accessed on 03/20/2018).

[12] Twemcache. https://twitter.com/twemcache.

[13] WhatsApp. `https://www.whatsapp.com/`. (Accessed on 03/20/2018).

[14] Withdrawal Vulnerabilities Enabled Bitcoin Theft from Flexcoin and Poloniex | PCWorld. https://www.pcworld.com/article/2104940/withdrawal-vulnerabilities-enabled-bitcoin-theft-from-flexcoin-and-poloniex.html. (Accessed on 09/20/2017).

[15] *TPC-C Benchmark (Revision 5.11.0).* The Transaction Processing Council, http://www.tpc.org/tpcc/, 2010.

[16] Y. Alabdulkarim, M. Almaymoni, and S. Ghandeharizadeh. Polygraph: A Plug-n-Play Framework to Quantify Anomalies. *ICDE*, 2018.

[17] Y. Alabdulkarim, S. Barahmand, and S. Ghandeharizadeh. Bg: A scalable benchmark for interactive social networking actions. *Future Generation Computer Systems*, 85:29 – 38, 2018.

[18] R. Alagappan et al. Correlated Crash Vulnerabilities. In *12th USENIX (OSDI 16)*. USENIX Association, 2016.

[19] E. Anderson, X. Li, M. A. Shah, J. Tucek, and J. J. Wylie. What Consistency Does Your Key-value Store Actually Provide? In *Proceedings of the Sixth International Conference on Hot Topics in System Dependability*, HotDep'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[20] Apache. Kafka: A Distributed Streaming Platform, http://kafka.apache.org/.

[21] S. Apart. Memcached Specification, . http://code.sixapart.com/svn/memcached/trunk/server/doc/protocol.txt.

[22] M. Arlitt and T. Jin. A Workload Characterization Study of the 1998 World Cup Web Site. *IEEE Network*, 14(3), May 2000.

[23] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica. Quantifying Eventual Consistency with PBS. *VLDB J.*, 23(2):279–302, 2014.

[24] S. Barahmand and S. Ghandeharizadeh. BG: A Benchmark to Evaluate Interactive Social Networking Actions. *Proceedings of 2013 CIDR*, January 2013.

[25] S. Barahmand and S. Ghandeharizadeh. Benchmarking Correctness of Operations in Big Data Applications. In *MASCOTS 2014, Paris, France*, 2014.

[26] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995.*, pages 1–10, 1995.

[27] D. Bermbach and S. Tai. Benchmarking Eventual Consistency: Lessons Learned from Long-Term Experimental Studies. In *2014 IEEE International Conference on Cloud Engineering, Boston, MA, USA, March 11-14, 2014*, pages 47–56, 2014.

[28] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison- Wesley, 1987.

[29] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal Aspects of Serializability in Database Concurrency Control. *IEEE Trans. Software Eng.*, 5(3):203–216, 1979.

[30] D. Bradbury. The Problem with Bitcoin. *Computer Fraud & Security*, 2013(11):5–8, 2013.

[31] L. Brutschy, D. Dimitrov, P. Müller, and M. Vechev. Serializability for Eventual Consistency: Criterion, Analysis, and Applications. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 458–472. ACM, 2017.

[32] R. Cattell. Scalable SQL and NoSQL Data Stores. *SIGMOD Rec.*, 39:12–27, May 2011.

[33] B. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. *VLDB*, 1(2), Aug. 2008.

[34] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Cloud Computing*, 2010.

[35] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.*, 31(3):8, 2013.

[36] E. Cortez et al. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *SOSP*, 2017.

[37] A. Dey, A. Fekete, R. Nambiar, and U. Röhm. YCSB+T: Benchmarking Web-scale Transactional Databases. In *Workshops Proceedings of the 30th International Conference on Data Engineering Workshops, ICDE 2014, Chicago, IL, USA, March 31 - April 4, 2014*, pages 223–230. IEEE Computer Society, 2014.

[38] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudré-Mauroux. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB*, 7(4):277–288, 2013.

[39] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan. Blockbench: A Framework for Analyzing Private Blockchains. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1085–1100. ACM, 2017.

[40] A. Fekete, S. N. Goldrei, and J. P. Asenjo. Quantifying Isolation Anomalies. *Proceedings of the VLDB Endowment*, 2(1):467–478, 2009.

[41] F. Freitas, J. Leitão, N. M. Preguiça, and R. Rodrigues. Characterizing the Consistency of Online Services (Practical Experience Report). In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016, Toulouse, France, June 28 - July 1, 2016*, pages 638–645, 2016.

[42] S. Ghandeharizadeh et al. RangeQC: A Framework for Caching Range Predicate Query Results. Technical Report 2018-03, http://dblab.usc.edu/Users/papers/rangeqc.pdf, USC Database Laboratory, 2018.

[43] S. Ghandeharizadeh et al. Rejig: A Scalable Online Technique for Cache Server Configuration Changes. Technical Report 2018-05, http://dblab.usc.edu/Users/papers/rejig.pdf, USC Database Laboratory, 2018.

[44] S. Ghandeharizadeh, J. Yap, and H. Nguyen. Strong Consistency in Cache Augmented SQL Systems. *Middleware*, 2014.

[45] W. Golab, M. R. Rahman, A. AuYoung, K. Keeton, and X. S. Li. Eventually Consistent: Not What You Were Expecting? *Communications of the ACM*, 57(3):38–44, 2014.

[46] W. Golab, M. R. Rahman, A. A. Young, K. Keeton, J. J. Wylie, and I. Gupta. Client-centric Benchmarking of Eventual Consistency for Cloud Storage Systems. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 28:1–28:2, New York, NY, USA, 2013. ACM.

[47] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[48] H. T. Kung and C. H. Papadimitriou. An Optimality Theory of Concurrency Control for Databases. In P. A. Bernstein, editor, *Proceedings of the 1979*

ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, May 30 - June 1., pages 116–126. ACM, 1979.

[49] L. Lamport. On Interprocess Communication. *Distributed computing*, 1(2):86–101, 1986.

[50] L. Lamport. How to Make a Mulitprocessor Computer that Correctly Executes Multiprocess Programs. In *Readings in computer architecture*, pages 574–575. Morgan Kaufmann Publishers Inc., 2000.

[51] I.-C. Lin and T.-C. Liao. A Survey of Blockchain Security Issues and Challenges. *IJ Network Security*, 19(5):653–659, 2017.

[52] S. Liu, S. Nguyen, J. Ganhotra, M. R. Rahman, I. Gupta, and J. Meseguer. Quantitative Analysis of Consistency in NoSQL Key-Value Stores. In *Quantitative Evaluation of Systems, 12th International Conference, QEST 2015, Madrid, Spain, September 1-3, 2015, Proceedings*, pages 228–243, 2015.

[53] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd. Existential Consistency: Measuring and Understanding Consistency at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 295–310, New York, NY, USA, 2015. ACM.

[54] S. Lu, A. J. Bernstein, and P. M. Lewis. Correct Execution of Transactions at Different Isolation Levels. *IEEE Trans. Knowl. Data Eng.*, 16(9):1070–1081, 2004.

[55] D. L. Mills. A Brief History of NTP Time: Memoirs of an Internet Timekeeper. *SIGCOMM Comput. Commun. Rev.*, 33(2):9–21, Apr. 2003.

[56] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *NSDI*, pages 385–398, Berkeley, CA, 2013. USENIX.

[57] Oracle. Coherence Developer's Guide. http://download.oracle.com/docs/cd/E18686_01/coh.37/e18677/gs_intro.htmBABDDBAD.

[58] C. H. Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of the ACM (JACM)*, 26(4):631–653, 1979.

[59] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi. YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores. In *Cloud Computing*, New York, NY, USA, 2011. ACM.

[60] A. Pavlo. What Are We Doing With Our Lives?: Nobody Cares About Our Concurrency Control Research. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 3–3, New York, NY, USA, 2017. ACM.

[61] Pivotal. RabbitMQ, https://www.rabbitmq.com/.

[62] M. R. Rahman, W. Golab, A. AuYoung, K. Keeton, and J. J. Wylie. Toward a Principled Framework for Benchmarking Consistency. In *Presented as part of the Eighth Workshop on Hot Topics in System Dependability*, Berkeley, CA, 2012. USENIX.

[63] F. B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Trans. Comput. Syst.*, 2(2):145–154, May 1984.

[64] A. Silberschatz, H. F. Korth, S. Sudarshan, et al. *Database System Concepts*, volume 4.

[65] E. G. Sirer. NoSQL Meets Bitcoin and Brings Down Two Exchanges: The Story of Flexcoin and Poloniex. http://hackingdistributed.com/2014/04/06/another-one-bites-the-dust-flexcoin/. (Accessed on 10/01/2017).

[66] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The End of an Architectural Era: (It's Time for a Complete Rewrite). In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1150–1160. VLDB Endowment, 2007.

[67] M. Stonebraker and A. Pavlo. The SEATS Airline Ticketing Systems Benchmark. http://hstore.cs.brown.edu/projects/seats.

[68] D. Tapscott and A. Tapscott. *Blockchain Revolution: How the Technology Behind Bitcoin is Changing Money, Business, and the World*. Penguin, 2016.

[69] Terracotta. Ehcache, http://ehcache.org/documentation/overview.html.

[70] D. Terry. Replicated Data Consistency Explained Through Baseball. *Communications of the ACM*, 56(12):82–89, 2013.

[71] W. Vogels. Eventually Consistent. *ACM Queue*, 6(6):14–19, 2008.

[72] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu. Data Consistency Properties and the Trade-offs in Commercial Cloud Storages: The Consumers' Perspective. In *CIDR*, 2011.

[73] T. Warszawski and P. Bailis. ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 5–20, New York, NY, USA, 2017. ACM.

[74] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):255–270, Dec. 2002.

[75] A. Wolski. TATP Benchmark Description (Version 1.0). http://tatpbenchmark.sourceforge.net, 2009.

[76] R. Yasaweerasinghelage, M. Staples, and I. Weber. Predicting Latency of Blockchain-based Systems Using Architectural Modelling and Simulation. In *Software Architecture (ICSA), 2017 IEEE International Conference on*, pages 253–256. IEEE, 2017.

[77] Y. Yu et al. Model Checking TLA+ Specifications. In *Correct Hardware Design and Verification Methods*, 1999.

[78] K. Zellag and B. Kemme. How Consistent is Your Cloud Application? In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 6:1–6:14, New York, NY, USA, 2012. ACM.

[79] K. Zellag and B. Kemme˙Consistency Anomalies in Multi-tier Architectures: Automatic Detection and Prevention. *The VLDB Journal*, 23(1):147–172, Feb. 2014.