Testing Database Applications with Polygraph

Yazeed Alabdulkarim, Marwan Almaymoni, Shahram Ghandeharizadeh
Haoyu Huang, Hieu Nguyen

Database Laboratory Technical Report 2018-02

Computer Science Department, USC

Los Angeles, California 90089-0781

{yalabdul,almaymon,shahram, haoyuhua, hieun}@usc.edu

Abstract

Diverse applications implement read and write *transactions* using a data store. Testing whether transactions of database applications provide strong consistency is challenging. It requires an end-to-end testing as an application may consist of several components that impact the consistency of data. Polygraph is a conceptual plug-n-play framework to quantify the amount of anomalies produced by an application. We show several use cases of Polygraph for two major application classes: e-commerce and cloud. One long-term objective of Polygraph is to reduce the cost and time required to test a data driven application, so that developers may focus more time and effort on applications' features and requirements.

1 Introduction

Diverse applications implement read and write *transactions* using a data store. A read transaction may retrieve a Wikipedia page or look up a member profile on a dating site such as Tinder. A write transaction may purchase a book using an e-commerce site such as Amazon or extend a friend invitation to a member of a social networking site such as Facebook. Testing whether transactions of data driven applications provide strong consistency is challenging. It requires an end-to-end testing as an application may involve several components that impact data and its consistency. Employing a data store that provides strong consistency is not sufficient. For example, improper handling of cache race conditions may result in stale reads in cache augmented SQL systems [20]. An implementation may compromise consistency for a variety of reasons including: software and hardware bugs [28], undesirable race conditions [20], and a mismatch between an implementation and its intended design.

Polygraph[1] [7, 6] is a plug-n-play framework to quantify the amount of anomalies produced by a system. It is external to the system and its data storage infrastructure. Polygraph models an application at a conceptual level at the granularity of entities and their relationships. This makes it independent of the physical implementation of an application.

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

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 generate a log record providing its start time, end 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. Polygraph builds a database (at a conceptual level) of candidate values for entities/relationships as it processes these log records.

Unlike techniques that check for Conflict and View Serializability [14, 26], Polygraph is not provided with the precise order of actions by each transaction. Polygraph computes a strict serial schedule [26] to establish the *value* of an entity/relationship read by a transaction. If the observed value is not produced by a serial schedule then Polygraph has detected an anomaly. Moreover, computing strict serial schedules enables Polygraph to detect both transaction isolation anomalies [13] and linearizability anomalies [23]. This is because strict serializability is a stronger consistency model than both serializability [26] and linearizability [23, 24]. All executions that are not acceptable by serializability or linearizability are prohibited in strict serializability, but not vice-versa. For those data stores that provide transactional properties at the granularity of a single data item, an experimentalist may use Polygraph to detect linearizability anomalies by representing each action that manipulates a data item as one transaction [23].

In this paper, we share experiences using Polygraph to test different application classes. We demonstrate its usefulness to verify data consistency of an implementation to provide strong consistency. Polygraph has enabled us to focus on improving our designs to handle complex scenarios while gaining confidence about the correctness of our implementation. We have used it to detect software bugs, verify strong consistency, detect overlooked race conditions, and improve application designs to handle unexpected scenarios that impact consistency of data.

Polygraph is unique as a data consistency testing tool. Prior work focus on serializability violations and detect them by building a dependency graph. For example, [32] operates at a middle-tier and tags data items to obtain dependency information. 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. Polygraph does not have such requirements. This paper is different than our prior work [7] that describes Polygraph in that we focus on using it to test two classes of applications: e-commerce and cloud services.

The rest of this paper is organized as follows. Section 2 provides an overview of polygraph. We describe our experiences using Polygraph in Section 3. Brief future research directions are provided in Section 4.

2 Overview of Polygraph

Figure 1 shows how an experimentalist uses Polygraph in three distinct steps: Authoring (1.A and 1.B), Deployment (1.C), and Monitoring (1.D). During the authoring phase, the experimentalist employs Polygraph's visual user interface to identify entities and relationship sets, and those actions that constitute each transaction and their referenced entity/relationship sets. An action may insert or delete one or more entities/relationships,

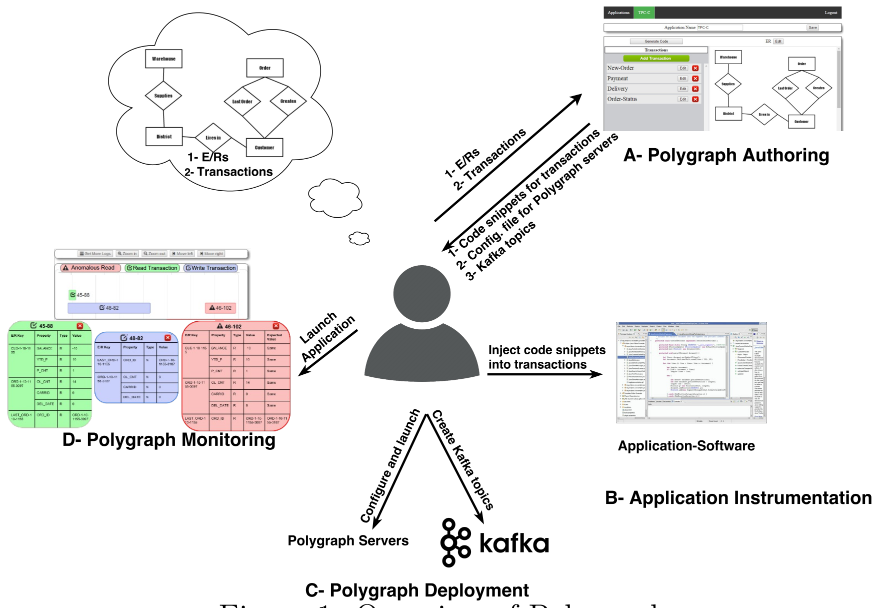

Figure 1: Overview of Polygraph.

and/or read or update one or more attribute values of an entity/relationship.

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 2.

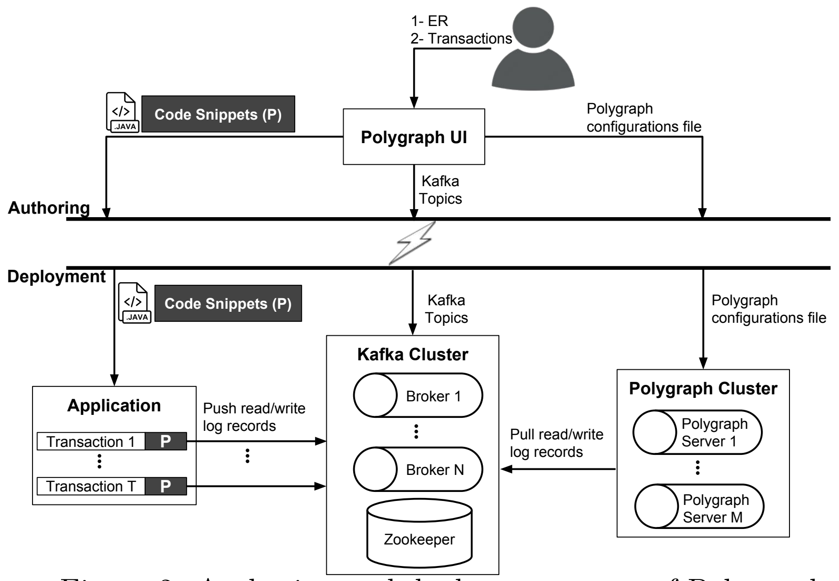

Figure 2: Authoring and deployment steps of Polygraph.

During the deployment phase, the 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.

During the monitoring phase (see D in Figure 1), the 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 [9], a distributed framework that stores streams in a fault-tolerant manner. Hence, an experimentalist may monitor an application in two possible modes. Either (a) offline by processing log records generated some time ago and buffered in Kafka, or (b) online by processing log records as they are produced by the application and streamed into Kafka.

Polygraph is a scalable framework that utilizes parallelism to quantify the amount of anomalies produced by an application, see Polygraph cluster in Figure 2. 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. The value of these attributes enables Kafka to construct partitions of log records, see Kafka brokers of Figure 2. 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 relationship.

Polygraph is general purpose and supports diverse applications with read-heavy and write-heavy workloads. We have used Polygraph to evaluate alternative implementations of benchmarks including TPC-C [5], YCSB [16], BG [12], SEATS [29], and TATP [30]. See [6] for details.

3 Applications

We show several use cases for Polygraph with two major application classes: e-commerce and cloud. For the first, we demonstrate Polygraph with a simple financial application and the TPC-C benchmark [5]. For the latter, we describe a variety of applications representing different workloads and our experience of using Polygraph with them.

3.1 E-Commerce Applications

E-commerce applications implement a variety of transactions such as making a payment or placing new order. These transactions may read and/or write several 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 (ACID) semantics. Database systems offer a variety of isolation levels [13]. Lower isolation levels may enhance performance but may compromise correctness as they are exposed to anomalies. A recent survey [27] shows the majority of 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 amount of anomalies [13] with different SQL systems.

How many anomalies does an application incur with weak isolation levels? Polygraph is able to quantify and answer this question. Polygraph 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. To illustrate, we provide the following two use cases.

3.1.1 A Financial Application

We 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. We used MySQL 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. We represented it as one read-write transaction. It reads the account balance and deducts $100 from its value. We differentiate between the conceptual transactions, termed *ConXacts* and their physical implementation, termed *PhyXacts*.

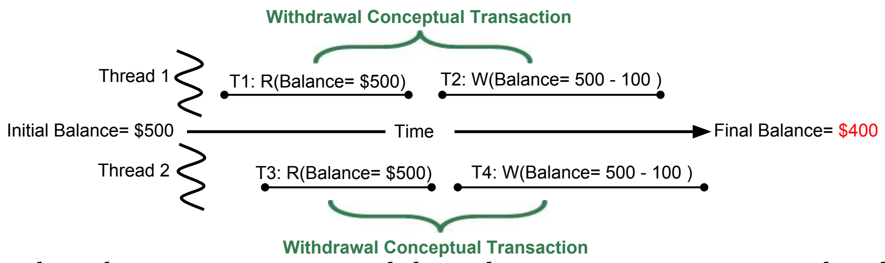
Withdrawal Conceptual Transaction
Thread 1
T1: R(Balance= $500)
T2: W(Balance= 500 - 100)
Initial Balance= $500
Time
Final Balance= $400
Thread 2
T3: R(Balance= $500)
T4: W(Balance= 500 - 100)
Withdrawal Conceptual Transaction

Figure 3: Two threads execute money withdrawal transactions concurrently which results in an anomaly.

We 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 3 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 3, 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 4. We show each ConXact 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 colored in red.

Figure 4 shows eight ConXacts, five are anomalous. As the user puts her mouse over a money withdrawal ConXact, Polygraph shows its details. We show the observed account balance and the withdrawal amount. In addition, we show the expected account balance values for the anomalous ConXacts. Figure 4 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.

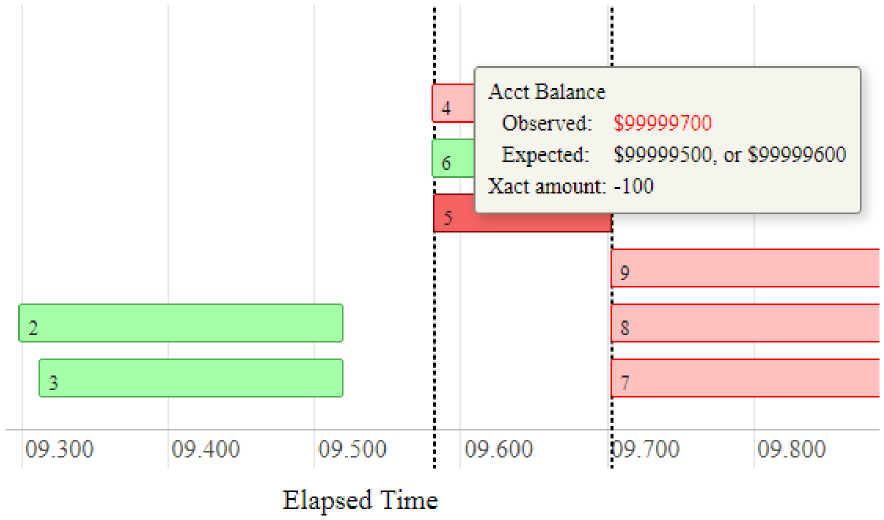

Figure 4: Polygraph monitoring tool visualizes concurrent money withdrawal conceptual transactions.

3.1.2 TPC-C

TPC-C is the defacto benchmark for evaluating online transaction processing systems. Its workload represents e-commerce or wholesale supplier applications. We modeled TPC-C transactions in Polygraph and extended every transaction to generate its corresponding log record, as shown in Table 2. For example, Order-Status reads a customer information and retrieves her last order. This transaction is represented as a read transaction that reads the last order relationship, a customer and an order entity. By representing the last order relationship, Polygraph is able to capture the last order for a customer since it is not known in advance when issuing an Order-Status transaction.

A limitation of Polygraph is that it is not capable of performing query processing. It looks-up values of entities and relationships using their primary keys. Stock-Level transactions in TPC-C require special handling to identify items of the last 20 orders in a district. One may extend and tailor Polygraph to support Stock-Level. This renders Polygraph to become workload-specific.

We used Polygraph to quantify the number of anomalies with OLTP-Bench's implementation of TPC-C [19] using MySQL. MySQL supports Serializable, Repeatable-Read, Read-Committed and Read-Uncommitted isolation levels [2]. With all four, a transaction releases its write locks at its commit/abort point. They differ in 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 [1]. 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. With Read-Uncommitted, a read observes the latest value for a data item, which may result in dirty read anomalies [13].

We augment TPC-C benchmark with Polygraph to quantify the number of anomalies when MySQL is configured with the alternative isolation levels. For each, we show the number of anomalies and new-order transactions per minute, see Table 1. These results show the DBAs folklore that lower isolation levels result in a higher throughput to be accurate. When comparing Read-Committed with Serializable, it 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: Number of detected anomalies with different database isolation levels of MySQL with TPC-C.

Isolation level	**Number of 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 anomalies represent lost updates [13] due to concurrent read-modify-write payment transactions. Figure 5 shows an example of these anomalies, 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. In addition, Polygraph detected dirty read anomalies with Read-Uncommitted. These represent order-status transactions observing the values of aborted new-order transactions.

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 causing lost updates. This may result in dirty reads if one of these transactions abort. On the other hand, with Read-Committed and Repeatable-Read, it is more likely that these concurrent transactions will 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.

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

These results are interesting for two reasons. First, it shows 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, it highlights the importance of measuring the anomalies observed by the application, rather than relying on the semantics of the isolation level as it may behave differently depending on the workload. An application may deem 99.99% accuracy sufficient given the performance gains and Polygraph quantifies this accuracy, see percentages reported in Table 1. These results highlight Polygraph as a timely and useful tool.

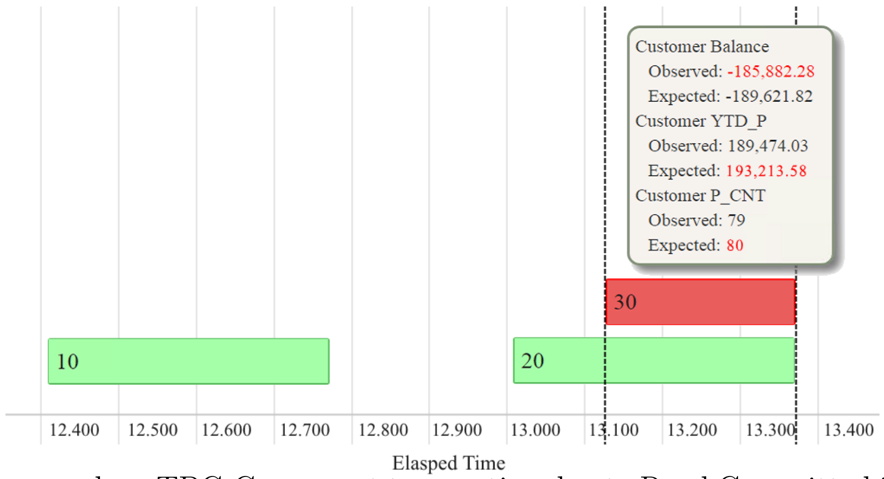

Figure 5: An anomalous TPC-C payment transaction due to Read Committed isolation level.

3.2 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 transactions, 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 [15] for details.

In the last decade, the rise of NewSQL systems, such as Goolge's Spanner [17], aim to provide semantics of SQL while achieving high availability, scalability and full transaction support. NewSQL emerges since 1) many applications cannot afford to use NoSQL systems for weaker consistency (financial applications, voting systems), 2) organizations find out that weaker consistency also decrease their developer's productivity since they spend more time handling inconsistent data in their code [17, 24].

NewSQL systems must address the following challenges: 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 as TLA+ [31] in that it tests the correctness of the implementation. Polygraph also complements failure injection frameworks [8] in that it is an end-to-end testing suite. Polygraph eases the development of these NewSQL

Table 2: Conceptual representation of three benchmarks.

Benchmark	**Transaction name**	**Conceptual representation**
TPC-C	New-Order	Insert order entity, read and update district entity, update the last order relationship
	Payment	Read and Update customer entity
	Delivery	Update order and customer entities
	Order-Status	Read the last order relationship, read customer and order entities
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

systems in that they 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 tests the interaction between different components in the system. Gamma testing is the last stage that tests the new software in an emulated production environment.

We envision 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.

We describe Polygraph's extensions to benchmarks that focus on the performance of cloud applications with correctness testing, e.g., Yahoo's Cloud Serving Benchmark [16] (YCSB) and BG [12] (a social networking benchmark). Table 2 shows the conceptual representation of YCSB and BG transactions. YCSB has a variety of workloads that represent different applications. Below, we discuss how Polygraph is used to test three typical cloud applications: session stores, photo tagging and messaging applications.

3.2.1 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 workloads (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, we 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, we detected the following bugs in the implementation. First, worker threads were merging buffered writes incorrectly which corrupts their state. Second, at times buffered writes were lost because their hashcode that mapped them a cache server was incorrect.

3.2.2 Photo Tagging Application

Photo tagging applications, such as [3], enable users to tag and display all tags of a photo. Add a tag is a write action. An example read action is to retrieve all tags of a photo. A photo tagging application represents a read-heavy workload. Systems in support of this application employ caches to enhance its performance characteristics [25].

Cache server failures are inevitable with a large deployment. When a cache server fails, its in-memory content is lost which degrades performance due to cache misses. With the advent of persistent caches including those that envision the use of non-volatile memory (NVM), a cache server preserves its content after a failure and reuses these data items upon its recovery. Should data items be updated during the failure then some persisted cached content may become stale. We used Polygraph to quantify the number of stale reads per second an application observes after a cache server failure. We observed a significant number of stale reads that peaked right after the cache server's recovery, see Figure 6. This motivated us to design and build a caching layer, Gemini, that preserves consistency in the presence of cache server failures.

Polygraph was used to verify 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 helped us 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. We emulated all these failure and validated Gemini's correctness without changing

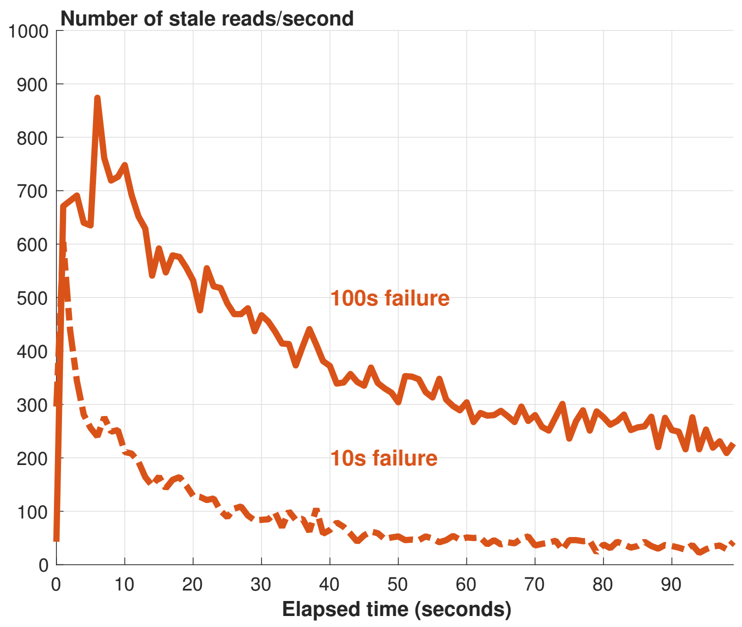

Figure 6: Number of stale reads per second after recovery from a 10 and a 100 second failure.

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. A deployment may 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 [10]. 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. We used Polygraph to quantify the number of stale reads in real-world traces [18, 11] simulating configuration changes and an imposed system load. It motivated us to design a scalable online technique, Rejig [22], that facilitates configuration changes (such as adding and removing cache servers) while providing strong consistency.

3.2.3 Messaging Application

Messaging applications, such as [4], allows 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 requires range queries. Caching these queries and looking them up maybe faster than computing them using an index structure. This motivated us to build RangeQC [21], 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. Dendrites 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, we extended Polygraph 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 many anomalies in our initial implementation. Most were attributed to undesirable race conditions that we fixed. Here, we present two examples. First, we maintained a lock manager for ranges of values and were releasing locks prior to updating the database in an update action. As a result, a different thread was able to query the database and insert its 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. In our first implementation, the commit for these multi-node updates was not atomic. One cache server would commit independent of the others. 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 detected this scan and we were able to compare concurrent scans with one another. We resolved by committing updates across multiple cache servers atomically.

4 Future research

Our long-term objective is to reduce the cost and time required to test a data driven application. To realize this objective, Polygraph must evolve to become extensible. Use cases of this paper highlight extensibility along two dimensions. First, Polygraph must include replaceable adapters that customize it to detect different causes of anomalies. For example, an adapter may provide patterns of transaction interactions that correspond to the consistency setting of a SQL system such as Read Committed. Next, it would attribute a possible cause of the anomalous transactions of Figure 5 to the consistency setting of the data store. Such hints empower a developer to debug an identified anomaly faster. Second, Polygraph must expose interfaces that empower a developer to extend its core functionality to process queries. This would enable a developer to customize Polygraph to become application specific, e.g., to evaluate correctness of Stock-Level transaction of TPC-C. Both extensibility dimensions are applicable to our presented use cases and constitute our short-term research directions.

References

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

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

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

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

[5] *TPC-C Benchmark*. The Transaction Processing Council, 2010.

[6] Alabdulkarim, Y., et al. Polygraph. Technical Report 2017-02, USC Database Laboratory, 2017.

[7] Alabdulkarim, Y., et al. Polygraph: A Plug-n-Play Framework to Quantify Anomalies. *ICDE* (2018).

[8] Alagappan, R., et al. Correlated crash vulnerabilities. In *12th USENIX (OSDI 16)* (2016), USENIX Association.

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

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

[11] Arlitt, M., and Jin, T. A workload characterization study of the 1998 world cup web site. *IEEE Network 14*, 3 (May 2000).

[12] Barahmand, S., et al. BG: A Benchmark to Evaluate Interactive Social Networking Actions. *CIDR* (2013).

[13] Berenson, H., et al. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the ACM SIGMOD* (1995).

[14] Bernstein, P., et al. *Concurrency Control and Recovery in Database Systems.* Addison- Wesley, 1987.

[15] Cattell, R. Scalable SQL and NoSQL Data Stores. *SIGMOD Rec.* (2011).

[16] Cooper, B. F., et al. Benchmarking Cloud Serving Systems with YCSB. In *Cloud Computing* (2010).

[17] Corbett, J. C., et al. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst. 31*, 3 (2013), 8.

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

[19] Difallah, D. E., et al. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB* (2013).

[20] Ghandeharizadeh, S., et al. Strong Consistency in Cache Augmented SQL Systems. *Middleware* (2014).

[21] Ghandeharizadeh, S., et al. RangeQC: A Framework for Caching Range Predicate Query Results. Technical Report 2018-03, USC Database Laboratory, 2018.

[22] Ghandeharizadeh, S., et al. Rejig: A Scalable Online Technique for Cache Server Configuration Changes. Technical Report 2018-05, USC Database Laboratory, 2018.

[23] Herlihy, M. P., et al. Linearizability: A Correctness Condition for Concurrent Objects. *TOPLAS* (1990).

[24] Lu, H., et al. Existential Consistency: Measuring and Understanding Consistency at Facebook. SOSP '15.

[25] Nishtala, R., et al. Scaling Memcache at Facebook. *NSDI* (2013).

[26] Papadimitriou, C. H. The Serializability of Concurrent Database Updates.

[27] Pavlo, A. 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* (New York, NY, USA, 2017), SIGMOD '17, ACM, pp. 3–3.

[28] Stonebraker, M., and Cattell, R. 10 Rules for Scalable Performance in Simple Operation Datastores. *Communications of the ACM* (2011).

[29] Stonebraker, M., and Pavlo, A. The SEATS Airline Ticketing Systems Benchmark.

[30] Wolski, A. TATP Benchmark Description (Version 1.0), 2009.

[31] Yu, Y., et al. Model checking tla+ specifications. In *Correct Hardware Design and Verification Methods* (1999).

[32] Zellag, K., et al. Consistency Anomalies in Multi-tier Architectures: Automatic Detection and Prevention. *VLDB* (2014).