Polygraph

Yazeed Alabdulkarim, Marwan Almaymoni, Shahram Ghandeharizadeh
Database Laboratory Technical Report 2017-02
Computer Science Department, USC
Los Angeles, California 90089-0781
Email: {yalabdul,almaymon,shahram}@usc.edu

May 8, 2017

Abstract

Polygraph is a tool to quantify system behavior that violates serial execution of transactions. It is a plug-n-play framework that includes visualization tools to empower an experimentalist to (a) quickly incorporate Polygraph into an existing application or benchmark and (b) reason about anomalies and the transactions that produce them. We demonstrate Polygraph's ability to plug-in to existing benchmarks including TPC-C, SEATS, TATP, YCSB, and BG. In addition, we show Polygraph processes transaction log records in real-time and characterize its scalability characteristics.

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*. Several benchmarks have argued the amount of anomalies should be a first class metric when evaluating systems and comparing them with one another [6, 14, 31, 29].

For example, Amazon's DynamoDB supports two kinds of reads: Strongly consistent and eventually consistent reads [1]. Eventually consistent reads are the default and 50% cheaper than strongly consistent reads. However, they may not reflect the result of a recently acknowledged write due to delayed propagation of the write to different replicas of data. DynamoDB claims 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 [20]. The amount of tolerated anomalies is both application and workload specific. Some applications are resilient to anomalies while others may tolerate either no or infrequent anomalies [26, 11, 17, 18, 38].

Workload parameters such as inter-arrival time of requests and how they reference data items impact the amount of observed anomalies (and whether they are observed at all). To

[1]Strict serial execution means a transaction that starts after the commit point of another transaction cannot be reordered before it [28, 9].

illustrate, if inter-arrival time for read and write requests referencing a data item exceeds DynamoDB's one second threshold then the amount of anomalies is zero.

Polygraph 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. 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, 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. Unlike techniques that check for Conflict and View Serializability [8, 28], Polygraph is not provided with the precise order of actions by each transaction, see Section 5 for details. Hence, Polygraph computes a serial schedule 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.

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 update to all replicas. While there is a valid serial schedule because the read and the update did not overlap in time, the read observes a stale value due to use of DynamoDB's eventually consistent reads.

We plug Polygraph in several benchmarks including BG [6] and OLTP-Bench implementation of TPC-C, YCSB, SEATS and TATP [15]. These confirm that an experimentalist may incorporate Polygraph with a benchmark to quantify both the amount of anomalies as well as traditional performance metrics such as throughput.

The rest of this paper is organized as follows. Section 2 provides an overview of Polygraph. We detail its components in Section 3. Section 4 evaluates Polygraph and how it is used to model 5 different benchmarks, reporting on its vertical scalability with both a read-heavy and a write-heavy benchmark. We describe related work in Section 5. Brief conclusions and future research directions are outlined in Section 6.

2 Overview of Polygraph

This paper uses the following terminology. A read transaction consists of one or more read actions. A write transaction consists of one or more insert, update, and 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, we do 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.

A key feature of Polygraph is to utilize parallelism to quantify the amount of anomalies produced by an application. This enables Polygraph to utilize multiple cores of a CPU to scale vertically and multiple nodes to scale horizontally. Polygraph provides this feature by dividing the task of computing serial schedules into multiple subtasks where the transactions

assigned to each subtask reference entities/relationships that are mutually exclusive.

Polygraph represents a transaction as a log record. It employs a messaging system named Apache Kafka [4] to partition these records for parallel processing by Polygraph servers. Choice of Kafka is arbitrary and one may use an alternative such as RabbitMQ [30]. As detailed below, the key idea is for Polygraph to compute the partitioning attribute of a log record (transaction) whose value partitions these records. (See Section 3.3 for details.)

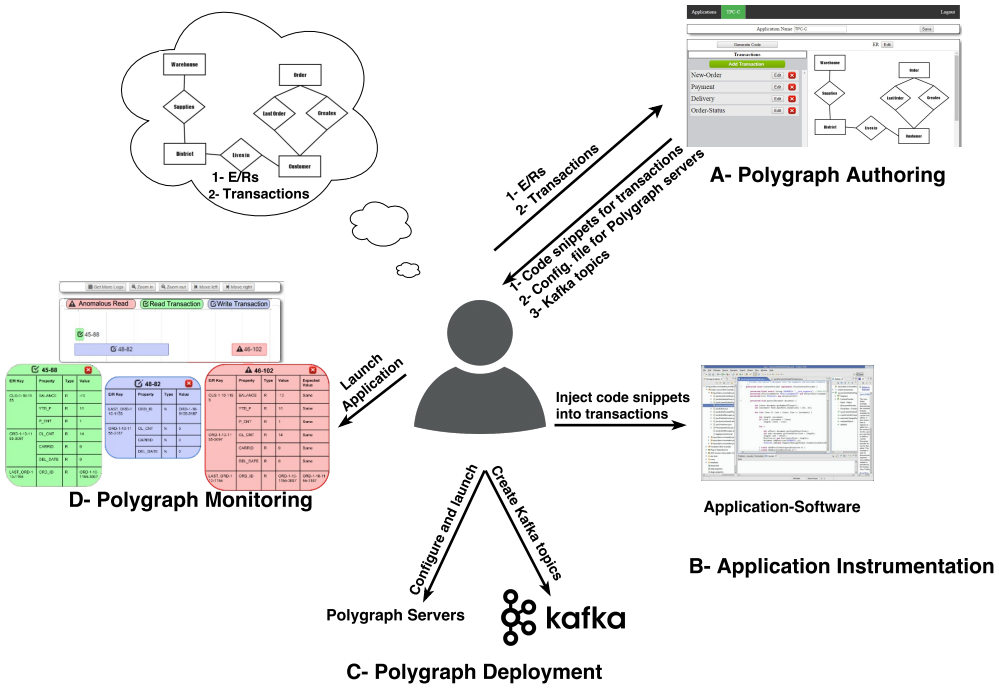

Figure 1: Overview of Polygraph.

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

Authoring produces the following three outputs. First, for each transaction, Polygraph generates 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 attribute of the log records. (See Figure 2.)

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

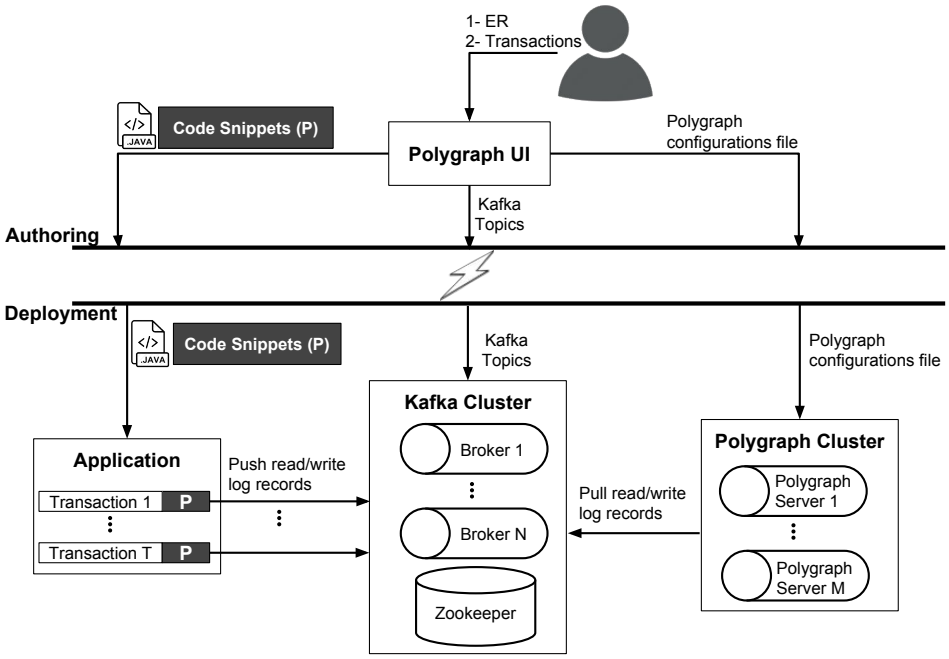

Figure 2: Authoring and deployment steps of Polygraph.

visualization 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.

Polygraph is a scalable framework. It partitions those log records that reference different entities and relationships to be processed independently. It does so 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. Log records in different partitions can be processed independent 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. This thread is termed a *validation* thread. When the number of partitions exceeds the number of validation threads, multiple partitions are assigned to one validation thread for processing.

Polygraph servers maintain a database that reflects the values of different entities and relationships, see Section 3.5. They compute candidate serial schedules that satisfy the isolation property of processing concurrent transactions and are *strictly serializable* [28, 9]. 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 they are

validated, 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, the value is marked as anomalous.

In enumerating candidate serial schedules, Polygraph constructs a snapshot of the value of entities/relationships to check for anomalies. One may configure Polygraph to either start with a snapshot of the database to check for anomalies or incrementally construct the state of the database as it processes read and write transaction (log records). The advantage of the later is that Polygraph does not incur a time consuming process to read the value of all entities/relationships. Its disadvantage is that Polygraph may fail to detect anomalous reads when the value of an entity or relationships is first established.

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 [2], YCSB [12], BG [6], SEATS [33], and TATP [37]. See Section 4.

3 Details of Polygraph

This section details components of Polygraph. We describe and motivate our use of a distributed stream processing framework such as as Kafka. Subsequently, we describe the structure of Polygraph records streamed into Kafka. Next, we detail how Polygraph's authoring tool computes the attribute used to partition log records to use parallelism to quantify the amount of anomalies. This is followed with how Polygraph servers detect anomalous reads.

3.1 Why 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 addition and removal of Kafka brokers.

Kafka's topics with partitioned logs enable Polygraph to scale. With Polygraph, the key of a Kafka record is a transaction id, its value is a structured log record (see Section 3.2), and its timestamp is the system clock of the server hosting the application. These records are generated by the Polygraph code snippets embedded in each transaction and published 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 authoring phase, see Section 3.3). 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 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 the one identified in Section 3.4.

3.2 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 (i.e., 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.

3.3 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 2. In turn, Polygraph's user interface generates code snippet to embed in a transaction. The code snippet constructs a log record for the transaction and publishes it to a Kafka topic. Moreover, 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 (i.e., values) pertaining to these entities and relationships.

A key challenge is how Polygraph computes the Kafka topic and partitioning attribute of log records. Below, we detail Algorithm 1 to address this challenge.

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.

We observe there is a class of applications that have a tree structure schema and compute the partitioning attribute by using this structure [32]. A schema is tree structure if it has exactly one many-to-one or one-to-one relationship to its ancestor, except for the root. This class of application applies to many real-world OLTP workloads [32]. Log records of an application with a tree-structured schema can be partitioned using the primary key of the root table.

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, a composite key of the intersection set is assigned as the partitioning attribute. Selecting one

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 $PT_i = \{ p \mid p$ 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 $CT = \{ t \mid t$ 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[ ].

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 is represented as an edge from one to many. We add 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 be represented 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, we duplicate each tree in CT to represent each possibility. Next, we remove all invalid trees from CT. In Step 4.1.8, we loop 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 attribute depends on the number of transactions, T, and number of one-to-one relationship sets, N. For each transaction, we build a tree. For each one-to-one relationship set, we duplicate candidate trees. For each candidate tree, we perform 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.

3.4 Detecting Anomalies

A cluster of Polygraph servers (see Figure 2) pulls log records from Kafka and processes them to detect anomalies. Each server is multi-threaded and each thread is assigned one or more partition logs for processing. A thread builds a snapshot of the database (at a conceptual level) as it processes log records. Every time a read transaction observes a value that is not consistent with its snapshot then it has detected an anomaly.

When a Polygraph server is launched, its input specifies the number of Kafka partitioned logs and how many threads it must launch. Kafka partitions are assigned to threads using a hash function. A Kafka partition is assigned to at most one thread.

Each 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. A thread computes these using the structured logs fetched (see Section 3.2) from its assigned Kafka partitioned logs.

Algorithm 2 shows how each Polygraph thread processes log records that identify different transactions. Polygraph must establish the value of each entity/relationship observed by a 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 then it discards that set for all future reads that reference the same

entity/relationship. Below, we describe each property in turn. Subsequently, we illustrate the algorithm with an example.

To illustrate the first, consider the read and write transactions shown in Figure 3. 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.

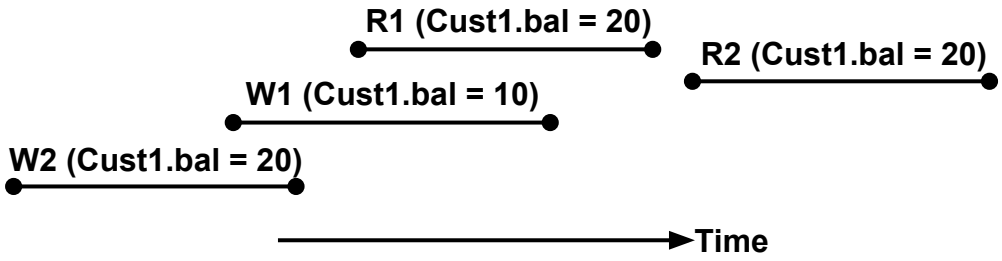

Figure 3: To identify read transactions that observe anomalous values, a Polygraph thread must consider all write transactions that overlap the read transaction transitively.

To illustrate the second, in Figure 4, 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).

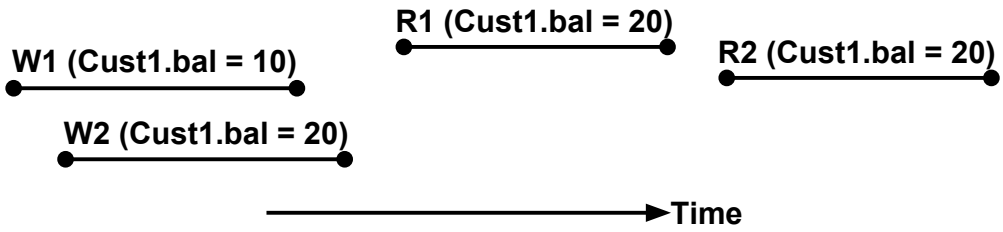

Figure 4: Discarded serial schedules to validate one read transaction are discarded for all future read transactions.

Algorithm 2 is general purpose and works for those 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 that transactions are sorted based on their start time. We describe the algorithm with the example shown in Figure 5. The 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 first read, $R1$, is selected to be validated. All unprocessed write transactions with start time earlier than $R1$'s end time are placed into a set denoted as $Writes$, $Writes$=$\{W1,W2\}$. *W-Before-R1* is a subset of $Writes$ that includes writes with start time earlier than $R1$'s

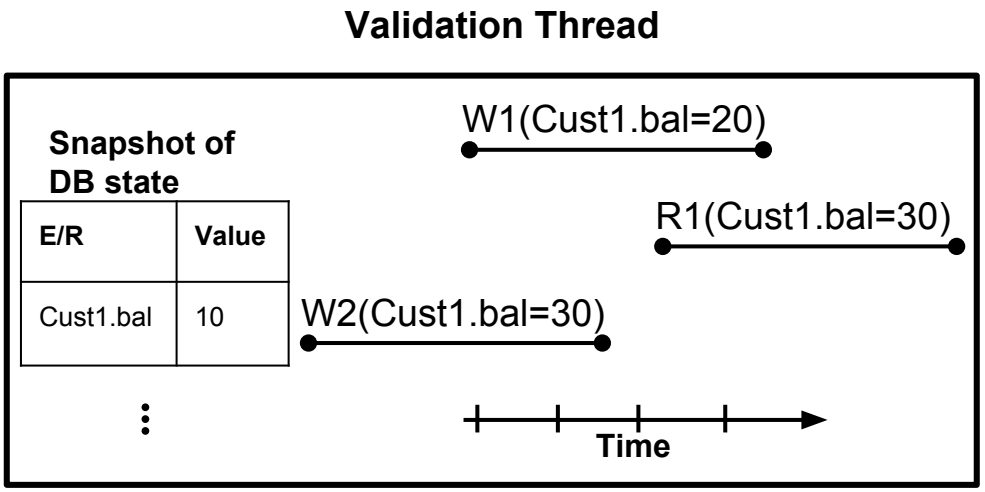

Figure 5: An example schedule consisting of two write transactions and one read transaction.

start time that reference an element of Q, *W-Before-R1*=$\{W1,W2\}$. Since the initial state of SSL is empty, nothing is removed in Step 5. Next, we compute all valid serial schedules using *W-Before-R1*, *SL-Before-R1*=$\{W1{:}W2, W2{:}W1\}$.

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

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

Then, we compute *Validate-R1*, which is a set of candidate serial schedules to validate $R1$. It is the result of the cartesian product between SSL and *SL-Overlap-R1*, *Validate-R1*=$\{W1{:}W2{:}W1{:}R1, W1{:}W2{:}R1{:}W1, W2{:}W1{:}R1, W2{:}R1{:}W1\}$. We remove 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, duplicate serial schedules in *Validate-R1* are removed, *Validate-R1* =$\{W1{:}W2{:}R1, W2{:}W1{:}R1, W2{:}R1{:}W1\}$. Now, *Validate-* $R1$ contains a set of candidate serial schedules to validate $R1$. We compare the values $R1$ observes with the expected values produced by each candidate serial schedule.

In the example of Figure 5, 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, $R1$ will be marked as a valid read transaction and the first and last serial schedules are maintained for $Cust1$.

3.4.1 Delayed log records

A multi-threaded application may create log records in a manner that does not represent the order of transactions executed by the application. A context switch may suspend the execution of a thread after it has executed a transaction and created its log and prior to

Algorithm 2 Detect Anomalies (Read Transaction $R1$)

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

its transmission of this log record to Kafka, see Thread 1 in Figure 6. 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 produced by the application. 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 there may exist other threads processing write transactions that reference the same entity/relationship and overlap them in time.

Polygraph's code snippet encourages an experimentalist to use the start and commit timestamp provided by a data store for a transaction. When this is not possible, Polygraph's code snippet sets the start timestamp of a log record before executing a transaction and its commit timestamp after the transaction commits[2]. Every time Polygraph encounters a log

[2]Arbitrary context switches between recording of start and end time increase the duration of a transaction. This in turn increases the number of possible serial schedules, increasing the likelihood of a true anomaly

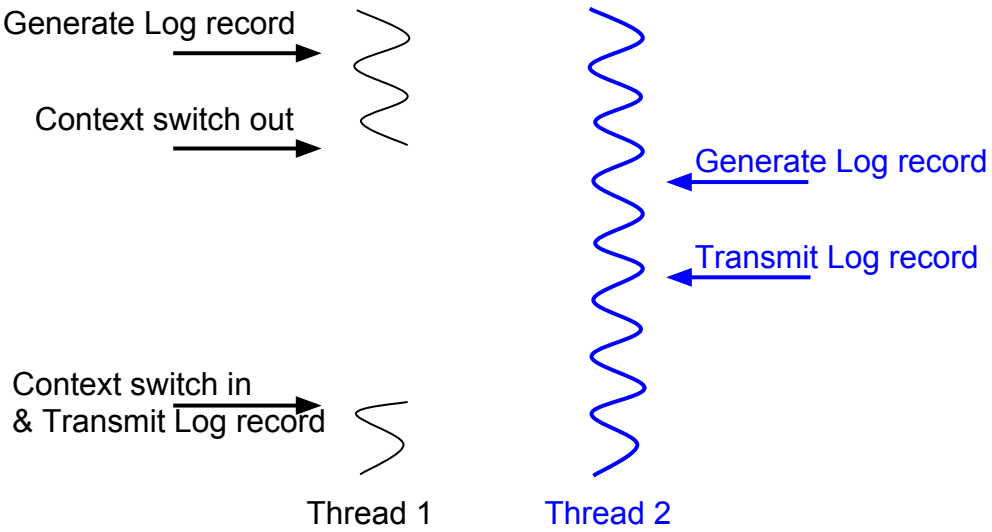

Figure 6: Log records transmitted out of order.

record with a start timestamp smaller than its last processed log record then it has detected an out of order transmission.

To process out of order log records, Polygraph snapshots its hash table that maintains the value of entities/relationships after processing a fixed number of log records (say 1000). The start timestamp of the last read transaction that Polygraph validated using this snapshot is assigned as the timestamp of the snapshot. Every time Polygraph encounters an out of order read or write log record ($T_{delayed}$), it goes back to a prior snapshot that is before the start time of $T_{delayed}$. It uses the values of entities/relationships in this state and all transactions that impact them (including $T_{delayed}$ in the correct order) to detect anomalous reads. To minimize the frequency of this operation, Polygraph maintains a batch of log records and sorts them based on their start timestamp prior to processing a log record.

3.5 Computing the value of entities/relationships

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 down 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, all subsequent reads are flagged as invalid even though they are correct. One way to mitigate the erroneous states from carrying over is to periodically reset the database value of entities/relationships marked as anomalous.

---

not being detected.

4 Evaluation

Table 1: Details of studied benchmarks.

Benchmark	**Target Application**	**Max Number of Partitions**	**Excluded Transactions**
BG	Interactive social networking actions	Number of members	We only considered 7 core actions, see Table 2
TPC-C	E-Commerce, wholesale supplier	Number of warehouses	Stock-Level
SEATS	Airline ticketing systems	1	Find Flights
TATP	Telecommunication	Number of subscribers	Get_New_Destination
YCSB	User profile cache, threaded conversations, etc.	Number of users	Scan

We evaluated the feasibility of Polygraph using multiple benchmarks including TPC-C [2], BG [6], SEATS [33], TATP [37], and YCSB [12]. The second column of Table 1 provides a brief description of each benchmark. Third column of this table shows the maximum number of partitioned logs supported by each benchmark. Except for SEATS, all other benchmarks provide for use of concurrent processing of log records[3]. We discuss this further when reporting on vertical scalability of Polygraph.

The last column of Table 1 highlights limitation of today's Polygraph, namely, its inability to support complex data driven functions and read actions that retrieve a set of values. Stock-Level transaction in TPC-C requires special handling to identify the items of the last 20 orders in a district. One may extend and tailor Polygraph to support Stock-Level, but this would render Polygraph to be workload specific. Polygraph does not support YCSB's Scan, SEATS' Find Flights and TATP's Get_New_Destination because it cannot validate range queries that retrieve a set of values. We intend to address this limitation as a part of our future research.

Table 2 shows the different transactions that constitute each benchmark. For each transaction, we identify the action (read, insert, update, and delete) performed on the entity/relationship using Polygraph's authoring visualization. In addition, we show the minimum, maximum, and average size of log records produced by the Polygraph generated code

[3]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, all log records must be sent to one partitioned log for processing.

Table 2: The type and size of log records generated by each transaction of five benchmarks studied in this paper.

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

snippet that is embedded in each transaction. 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 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., string representation of value 1245 is 4 bytes long while that of value 9 is one byte long.

4.1 Overhead Imposed by Polygraph

Polygraph code snippets embedded in the application generate log records for transactions and publish them to 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. Table 2 shows the minimum, average, and maximum log size for each transaction in bytes. We quantify the network overhead in bytes/second as follows: $T * \sum_{t \in transactions} frequency(t) \times logSize(t)$.

For example, with TPC-C, the workload mix of transactions is 45% New-Order, 43% Payment, 4% Order-Status, 4% Delivery, and 4% Stock-Level. If T is 10,000 transactions/sec, the minimum, average, and maximum network overhead is 5.2, 5.6, and 5.99 MB/second respectively.

4.2 A Case Study

This section describes how extending a transaction processing system with a cache such as memcached produces anomalies for TPC-C benchmark. It also demonstrates how Polygraph's monitoring tool identifies and displays transactions that participate to produce anomalies.

We implemented TPC-C using a cache augmented deployment consisting of MySQL and Twemcached [27]. With this implementation, a read transaction looks up its result set in the cache prior to querying the database. With a cache miss, the transaction issues its query to MySQL and stores the obtained result set in Twemcached for future reference. In the presence of updates, the application invalidates (deletes) impacted cached query result sets.

With this architecture, there is a race condition between read transactions caching results of queries and write transactions invalidating the cached query results. This race condition may produce anomalies when MySQL is configured with snapshot isolation. To illustrate, two concurrent read and write transactions referencing the same data item may execute as follows. First, the write transaction invalidates the cached query results it references, if any. Second, before the write transaction commits, the read transaction queries MySQL (configured with snapshot isolation) and inserts its resulting stale key-value pair in the cache. Consequently, a subsequent read transaction observes a stale value from the cache.

We used OLTP-Bench [15] implementation of TPC-C benchmark using MySQL version

5.6.31 with Twemcached version 2.5.3. In this experiment[4], Polygraph detected only one anomaly because TPC-C benchmark is a write heavy workload [13]. Figure 7 shows the anomalous read using Polygraph monitoring interface. The anomalous read R46-102, colored in red, is an Order-Status transaction. It reads a customer and order entities. It also reads a last order relationship, as shown in the red rounded rectangle. Polygraph monitoring interface shows the observed values by the read along with the expected values by Polygraph. For R46-102, the observed value for the order id property of the last order relationship is different than the expected value by Polygraph. R46-102 reads an old value for the last order that was placed in the cache by R45-88, colored in green. Write transaction 48-82 is a New-Order transaction which updates the last order id to ORD-1-10-1155-3187. R46-102 happens after W48-82, but does not observe its value. This happened because of a race condition between W48-82 and R45-88. W48-82 invalidates the cache entry for last order because it is going to insert a new order for that customer. Before W48-82 commits, R45-88 reads the last order for the customer from the database using snapshot isolation and places into the cache, causing a subsequent read by transaction R46-102 to read a stale value for the last order.

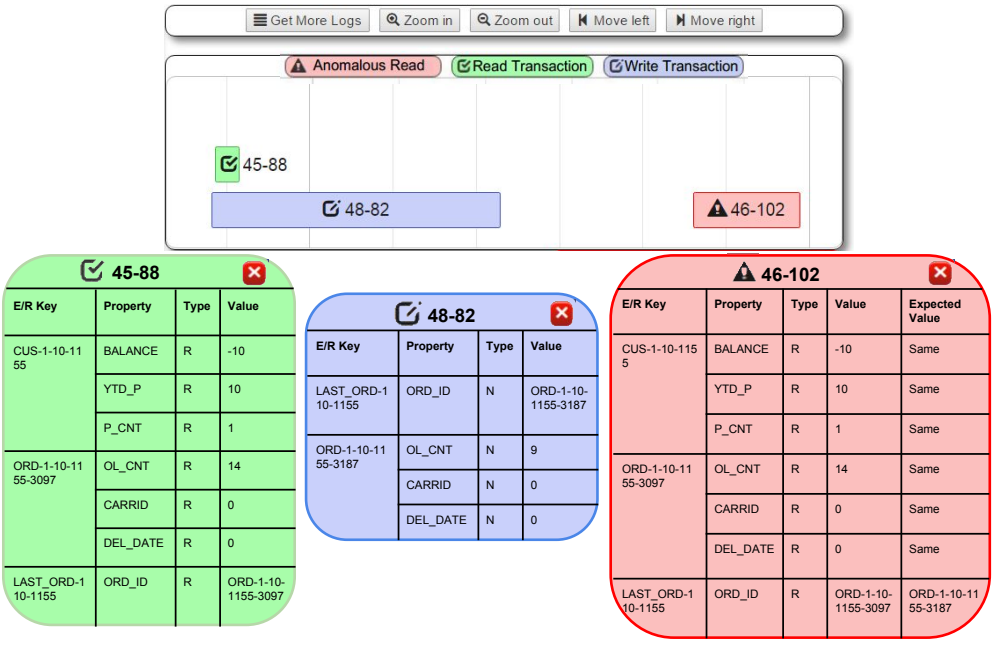

Figure 7: Polygraph visualization of a TPC-C anomaly using MySQL augmented with Twemcached.

4.3 Vertical scalability

Vertical scalability of Polygraph is dictated by the number of unique values for the partitioning attribute of the records. Third column of Table 1 shows this for the different benchmarks. Polygraph does not scale for SEATS because it supports at most one partitioned log. This means only one validation thread may process these records to compute the

[4]We ran TPC-C with one warehouse and 50 threads with the default workload for one minute.

amount of anomalies correctly. On the other hand, with benchmarks such as TPC-C and BG, the number of partitioned logs is dictated by the number of unique values for an entity set. In the following, we report on the scalability of Polygraph with these two benchmarks. We focus on Polygraph processing log records in an offline mode where a benchmark has generates all its log records and inserted them in Kafka. We do not consider an online version of Polygraph because its scalability is dictated by the rate at which a benchmark produces log records.

Our experimental setup consists of a Kafka cluster with 5 brokers and one Zookeeper instance, and no replication of log records for data availability. A Polygraph cluster consisting of one server with 8 hyperthreaded i7-4770 (4 core) 3.4 Ghz CPU, 16 GB of memory, and a 1 Gbps network card.

We repeat each experiment 5 times. The deviation between the observed results is negligible with both TPC-C and BG uniform trace. It is less than 16% with BG skewed trace, see Figure 8.

4.3.1 BG

With BG, we consider two different workloads using either a skewed or a uniform access pattern. With a skewed access pattern, roughly 62% of requests reference 20% of data items. Both pertain to a mix of read and write actions shown in Table 3, producing[5] approximately 17.1 million read log records and 2.6 million write log records.

Figure 8 shows Polygraph scales better with the uniform access pattern. This is due to log records being partitioned across validation threads more evenly. With 32 threads (maximum x-axis value) 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, explaining its superior scalability characteristics[6].

Table 3: 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]The social graph consisted of 10,000 members and 100 friends per member. BG produced load using 100 concurrent threads for one hour.

[6]With 32 threads, Polygraph processes uniform in 5.78 minutes versus 7.68 minutes with skewed.

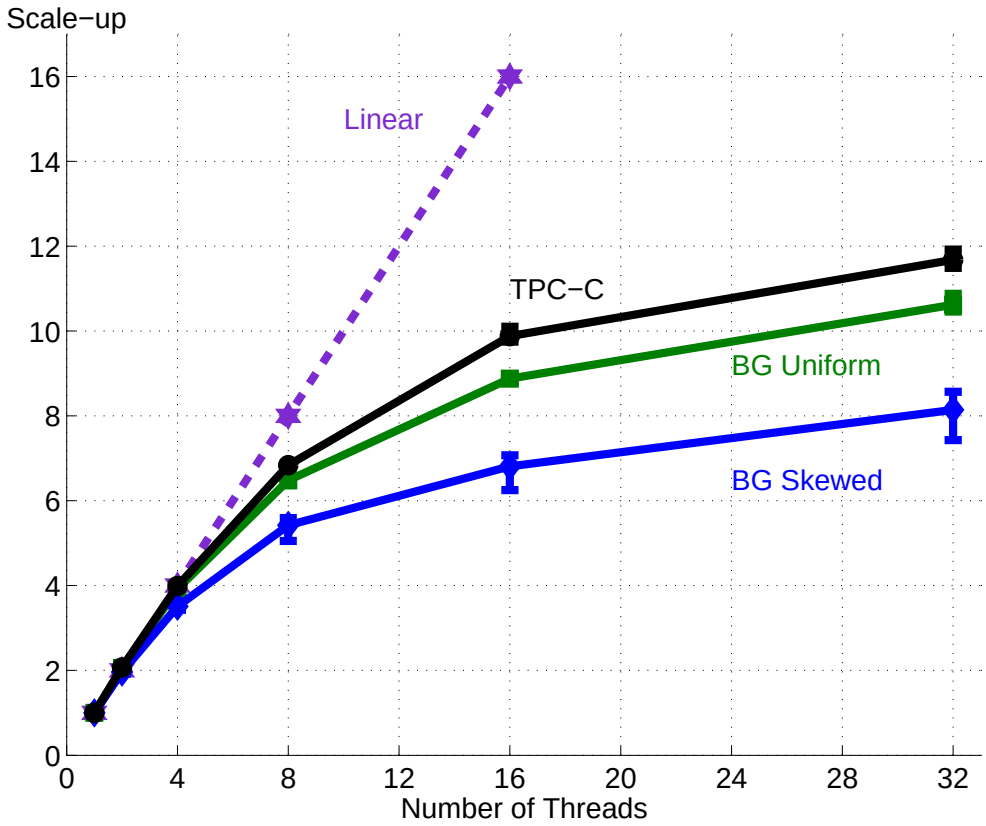

Figure 8: Vertical scalability with TPC-C and BG.

4.3.2 TPC-C

We used OLTP-Bench [15] to generate TPC-C traces. We ran TPC-C with 100 warehouses, 300 threads, and default workload for three hours against MySQL. The total number of transactions is 2,008,619. Polygraph validates these transactions in 58 minutes with 1 thread. This time is reduced to 5 minutes with 32 threads.

On an eight core CPU, with hyperthreading, Polygraph scales vertically, see Figure 8. From 1 to 8 threads, the speedup is linear. With 8 and more threads, the CPU cores are 80-100% utilized on the average, capping vertical scalability of Polygraph.

4.3.3 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, scalability of Kafka dictates Polygraph's scalability characteristics.

5 Related Work

Polygraph is motivated by our earlier work on the BG benchmark [6] that elevated the amount of anomalies produced by a system to a first class metric. A limitation of BG is that it is applicable to one workload, interactive social networking actions. In [7], we hypothesized the feasibility of a general purpose framework that quantifies the amount of anomalies for diverse classes of applications. Polygraph is the result of this hypothesis.

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 instrument it to produce log records to detect anomalies [3, 25, 38, 39, 36, 31, 18, 29]. Several detect serializability violations by building a dependency graph [38, 39, 10]. They focus on order of actions performed by transactions inside the DBMS. On the other hand, Polygraph is external to the application. Hence, it computes a serial schedule to establish the value of an entity/relationship read by a transaction.

Some studies use a metric different than the amount of anomalies [29, 31]. Δ-atomicity [31] captures time based staleness, 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 take the maximum to compute the global Δ for the system. YCSB++ [29] measures the time lag from one client completing an insert until a different client successfully observes the value. 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 [36]. Violation rate defined as the number of transactions that violate integrity constraints divided by the number of committed transactions is the focus of [16]. YCSB+t [14] employs a user specified constraint to compute an anomaly score. Finally, ϕ-consistency [25] 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 [25]. It develops an offline framework that detect consistency anomalies for linearizability, per-object sequential and read-after-write consistency models. Polygraph is different in that it quantifies the amount of anomalies for transactions violating a serial schedule.

Safety, atomicity, and regularity of a key value store is the focus of [3]. 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 4 shows a comparison of studies quantifying anomalies.

6 Conclusion

Polygraph is a plug-n-play framework to quantify the amount of anomalies produced by a system. For each transaction producing an anomaly, it empowers an experimentalist to display the transactions that referenced the same entities/relationships to hypothesize about

Table 4: Comparison of studies quantifying anomalies.

Study	**Consistency model(s)**	**Reported metric**
Polygraph	Strict Serializability [28, 9]	Amount of anomalies
Anderson et al. [3]	Safety, atomicity (equivalent to linearizability) and regularity [22]	Number of violations (cycles)
Bailis et al. [5]	Eventual consistency [35]	Probabilistic bounds on data staleness in versions and time
Fekete et al. [16]	Serializability [8]	Integrity constraints violation rate
Dey et al. (YCSB+t) [14]	Serializability [8]	Anomaly score using user specified constraints
Golab et al. [31]	Linearizability [19]	Δ-atomicity
Golab et al. [18]	Linearizability [19]	Γ (Gamma)
Liu et al. [24]	Read-your-write and linearizability [34]	Probability of satisfying a consistency model
Lu et al. [25]	Linearizability [19], per-object sequential [11, 23] 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++) [29]	Eventual consistency [35]	Measures the time lag from one client completing an insert until a different client successfully observes the value
Wada et al. [36]	Read-your-write and monotonic read consistency	Freshness confidence [7]
Zellag et al. [39]	Serializability [8]	Number of violations (cycles)

the cause of the anomalous value.

Our future research directions are several folds. First, we plan to extend Polygraph to report additional metrics. For example, quantifying Freshness Confidence [7] maybe important for applications that can tolerate anomalies for a certain amount of time. Second, we are extending Polygraph to validate read transactions that retrieve a set of entities/relationships using a range predicate. This would support Scan of YCSB and Get_New_Destination of TATP. Third, we plan to enhance the vertical scalability of Polygraph when the number of partitions is limited to one, e.g., SEATS benchmark. The challenge is how to use multiple threads to manipulate shared data structures on behalf of one partition. Finally, we

are expanding Polygraph to support other consistency models such as causal [21], rendering Polygraph suitable for evaluating data center deployments.

References

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

[2] *TPC-C Benchmark (Revision 5.11.0)*. The Transaction Processing Council, 2010.

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

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

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

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

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

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

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

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

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

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

[13] DeBrabant, J., Pavlo, A., Tu, S., Stonebraker, M., and Zdonik, S. B. Anti-Caching: A New Approach to Database Management System Architecture. *PVLDB 6*, 14 (2013), 1942–1953.

[14] Dey, A., Fekete, A., Nambiar, R., and Röhm, U. 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* (2014), IEEE Computer Society, pp. 223–230.

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

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

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

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

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

[20] Kung, H. T., and Papadimitriou, C. H. An Optimality Theory of Concurrency Control for Databases. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, May 30 - June 1.* (1979), P. A. Bernstein, Ed., ACM, pp. 116–126.

[21] Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM 21*, 7 (1978), 558–565.

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

[23] Lamport, L. How to Make a Mulitprocessor Computer that Correctly Executes Multiprocess programs. In *Readings in computer architecture* (2000), Morgan Kaufmann Publishers Inc., pp. 574–575.

[24] Liu, S., Nguyen, S., Ganhotra, J., Rahman, M. R., Gupta, I., and Meseguer, J. 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* (2015), pp. 228–243.

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

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

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

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

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

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

[31] Rahman, M. R., Golab, W., AuYoung, A., Keeton, K., and Wylie, J. J. 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.

[32] Stonebraker, M., Madden, S., Abadi, D. J., Harizopoulos, S., Hachem, N., and Helland, P. 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* (2007), VLDB Endowment, pp. 1150–1160.

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

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

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

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

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

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

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