Parameterized and Runtime-tunable Snapshot Isolation in Distributed Transactional Key-value Stores

Hengfeng Wei, Yu Huang, Jian Lu
State Key Laboratory for Novel Software Technology, Nanjing University, China
{hfwei, yuhuang, lj}@nju.edu.cn

Abstract*—Several relaxed variants of Snapshot Isolation (SI) have been proposed for improved performance in distributed transactional key-value stores. These relaxed variants, however, provide no specification or control of the severity of the anomalies with respect to SI. They have also been designed to be used statically throughout the whole system life cycle. To overcome these drawbacks, we propose the idea of parameterized and runtime-tunable snapshot isolation. We first define a new transactional consistency model called Relaxed Version Snapshot Isolation (RVSI), which can formally and quantitatively specify the anomalies it may produce with respect to SI. To this end, we decompose SI into three "view properties", for each of which we introduce a parameter to quantify one of three kinds of possible anomalies: k_1-BV (k_1-version bounded backward view), k_2-FV (k_2-version bounded forward view), and k_3-SV (k_3-version bounded snapshot view). We then implement a prototype partitioned replicated distributed transactional key-value store called CHAMELEON across multiple data centers. While achieving RVSI, CHAMELEON allows each individual transaction to dynamically tune its consistency level at runtime. The experiments show that RVSI helps to reduce the transaction abort rates when applications are willing to tolerate certain anomalies. We also evaluate the individual impacts of k_1-BV, k_2-FV, and k_3-SV on reducing the transaction abort rates in various scenarios. We have found that which one of the two parameters k_1 and k_2 of RVSI plays a major role in reducing transaction abort rates relies on the issue delays between clients and replicas.

***Index Terms*—Transactional key-value stores, relaxed version snapshot isolation, runtime-tunable consistency.**

I. INTRODUCTION

Distributed key-value stores have been widely deployed underlying modern large-scale Internet services such as electronic commerce and social networking. Well-known distributed key-value stores include both open source projects like Apache Cassandra[1] [1] and MongoDB[2] and commercial products like Google's BigTable [2], Yahoo!'s PNUTS [3], and Amazon's Dynamo [4]. To achieve high performance, high availability, and high scalability, large data sets are typically partitioned into multiple data shards, and each data shard is then independently replicated over its own master storage node and possibly several slave nodes [5].

Though application developers are satisfied with the easy-to-use interfaces such as *put(K key, V val)* and *get(K key)* provided by distributed key-value stores, there are increasing interests in transactional semantics for operations over an arbitrary group of data items. Without transactions, an application must explicitly coordinate accesses to shared data to avoid *anomalies* such as race conditions, partial writes, and overwrites, which are known hard to deal with [6]. With transactions, however, these anomalies are treated as low-level implementations, hidden from the developers by means of formally defined *transactional consistency models*.

Snapshot Isolation (SI) [7], among all the previously proposed transactional consistency models, avoids many of the undesirable anomalies for applications. The key idea of snapshot isolation is to provide each transaction with the "latest" consistent snapshot of all data items while avoiding write conflicts among concurrent transactions. This decoupling of reads and writes is quite intuitive to developers and eases the burden on programming and reasoning.

A major obstacle to providing snapshot isolation in distributed key-value stores is that distributed transactions satisfying strong transactional semantics often require intensive coordinations among multiple storage nodes, hamper transaction throughput, and result in poor system performance. For improved performance, several relaxed transactional semantics based on snapshot isolation have been proposed [8], [9], [6], [10], as reviewed in Section VII.

We argue, however, that these relaxed variants of snapshot isolation suffer from two main drawbacks. First, they provide no specification or control of the severity of the anomalies that are originally forbidden by snapshot isolation but introduced due to relaxation. At the worst, a transaction would have been executed on extremely stale data, rendering itself effectively worthless. In addition, without any guarantees, it would be much harder for the developers to write programs and reason about them. Second, like most of other transactional consistency models, these relaxed variants of snapshot isolation have been designed to be used statically throughout the whole system life cycle. In other words, the consistency model needs to be determined at the system design phase and remains unchanged once the system is deployed. However, no single individual consistency model satisfies all users in all situations [11]. Therefore, it would be desirable to allow each individual transaction to dynamically choose or tune its consistency level at runtime.

[1]Apache Cassandra: http://cassandra.apache.org/.
[2]MongoDB: https://www.mongodb.com/.

To further motivate the requirements of explicitly specifying and quantitatively bounding the inconsistency and dynamically tuning consistency at runtime, consider an online bookstore application. Suppose we have a "table" *Books* and each book has several attributes: title, authors, publisher, tags, sales, inventory, ratings, reviews, and so on. Different users might execute different transactions on the same *Books* table with different consistency models for different purposes. A customer (user U_1) who wants to obtain the basic information about a particular book of interest, would be satisfied with a transaction T_1 that responds quickly, even at the risk of retrieving *out-of-date* reviews or tags associated with the book. Imagine a bookstore clerk (user U_2) is running transaction T_2 to check the inventory of a book (i.e., the number of copies in stock; besides other attributes such as title and publisher) in order to decide whether the stock needs to be replenished. In this scenario, transaction T_2 may want to read the inventory value updated by concurrent transactions that commit *after* T_2 starts. A third user U_3, as a sales analyst, is studying the relationship between sales and ratings of a book by periodically running a read-only transaction T_3. In this scenario, it is acceptable for T_3 to obtain stale or concurrently updated versions of both sales and ratings as long as they were from a consistent snapshot, or from two *separate snapshots* that differ by a bounded version distance.

To overcome these two drawbacks discussed above, we propose the idea of parameterized and runtime-tunable snapshot isolation. The two key challenges we address in this paper are: *1)* how to specify bounded inconsistency with respect to snapshot isolation; *and 2)* how to implement it so as to support dynamic consistency choices. Accordingly, we make two main contributions as follows.

First, we define a novel transactional consistency model called Relaxed Version Snapshot Isolation (RVSI), which provides a formal specification for quantifying the anomalies it allows with respect to snapshot isolation. Generally speaking, RVSI is weaker than snapshot isolation but stronger than Read Committed isolation (RC) [7]. As with RC, RVSI guarantees that each transaction is committed atomically and only committed transactions can be observed by others. As with SI, RVSI also avoids write conflicts: if multiple concurrent transactions write to the same data items, at most one of them will commit. This *write-conflict freedom* property prevents the *lost updates* anomaly [7]. On the other hand, RVSI, being weaker than SI, does not require each transaction to obtain the latest preceding snapshot before it starts. Therefore, RVSI will introduce anomalies which are originally forbidden by SI. To precisely capture the anomalies RVSI allows, we first decomposes SI into three "view properties" (besides the write-conflict freedom property mentioned above), for each of which we then introduce a parameter to quantify one of three kinds of possible anomalies: *1)* k_1-BV *(k_1-version bounded backward view):* RVSI allows transactions to observe stale data, as long as the staleness is deterministically bounded by k_1; *2)* k_2-FV *(k_2-version bounded forward view):* RVSI allows transactions to observe data that are updated by concurrent transactions, as long as the concurrency level is deterministically bounded by k_2; *and 3)* k_3-SV *(k_3-version bounded snapshot view):* RVSI allows some transaction to observe two data items coming from different snapshots, as long as the "distance" between these two snapshots is deterministically bounded by k_3.

Second, we implement a prototype distributed transactional key-value store called CHAMELEON which achieves RVSI and allows each individual transaction to dynamically tune its consistency level at runtime. CHAMELEON supports both data partitioning and data replication across multiple data centers. In CHAMELEON, all replicas of a partition consist of a unique master node and several slave nodes. To guarantee the atomic commitment of distributed transactions across multiple masters (i.e., partitions), CHAMELEON uses a two-phase commit (2PC) protocol [12], into which the checking of RVSI specification has been integrated. After a transaction commits on a master, it is propagated asynchronously to its slaves.

We deploy CHAMELEON on Alibaba Cloud (Aliyun)[3] which consists of 9 servers spanning 3 data centers in North China, South China, and East China. The experiments show that RVSI helps to reduce the transaction abort rates when applications are willing to tolerate certain anomalies. We also conduct extensive controlled experiments to evaluate the individual impacts of k_1-BV, k_2-FV, and k_3-SV on reducing the transaction abort rates in various scenarios. We find that which one of the two parameters k_1 and k_2 plays a major role in reducing transaction abort rates relies on the issue delays between clients and replicas. Long-lived transactions due to larger issue delays are more likely to obtain data versions that are updated by concurrent transactions and whether they will be aborted are thus more sensitive to the parameter k_2. As the issue delays get smaller, the impacts of k_1 for k_1-BV on reducing transaction abort rates emerge and become more significant. In addition, given that k_1-BV or k_2-FV is (slightly) relaxed, increasing the value of k_3 helps to further reduce the transaction abort rates.

This paper is organized as follows. Section II reviews snapshot isolation. Section III defines RVSI and discusses its properties. Section IV describes the CHAMELEON prototype system which achieves RVSI. Section V presents the RVSI protocols for both data replication and data partitioning. Section VI evaluates the impacts of RVSI specification on transaction abort rates in various scenarios. Section VII discusses the related work. Section VIII concludes.

II. TRANSACTIONS AND SNAPSHOT ISOLATION

In this section we review the terminology about transactions and the definition of snapshot isolation following [13].

A. Transactions and Histories

A distributed transactional key-value store consists of a set of data items, each of which may have multiple versions. Clients interact with the store by issuing read or write operations on the data items, in the form of transactions. A

[3] Alibaba Cloud (Aliyun): https://www.aliyun.com/.

transaction T_i begins with a start operation s_i, then contains a sequence of read or write operations o_i, and ends with a terminating operation, either a commit c_i or an abort a_i.

To keep the notation simple, we assume that each transaction performs at most one read and at most one write on any data item. If $c_i \in T_i$, then we say that T_i is committed, and if T_i writes x, the version x_i is committed and becomes a committed version at the time T_i commits. If $a_i \in T_i$, then T_i is aborted, and x_i does not become part of the committed state. We use $w_i(x_i)$ to denote transaction T_i writing version i of data item x, and $r_i(x_j)$ transaction T_i reading version j of data item x written by transaction T_j.

Executions of a distributed transactional key-value store are described through histories. A *history* h over a set of transactions is defined as an (irreflexive) partial order, called *time-precedes order* $\prec_h$, over operations (i.e., start, read, write, commit, or abort) of those transactions such that *1*) For every operation o_i in h, transaction T_i terminates in h; *2*) The partial order $\prec_h$ is consistent with the order in which operations within a transaction are executed. That is, for any two operations o_i and o'_i of the same transaction T_i, if o_i precedes o'_i in T_i, then $o_i \prec_h o'_i$ in h. Especially, $s_i \prec_h c_i$; *3*) For every read $r_i(x_j) \in h$, there exists a write $w_j(x_j) \in h$ such that $w_j(x_j) \prec_h r_i(x_j)$; *4*) For any two committed transactions T_i and T_j, either $c_i \prec_h s_j$ or $s_j \prec_h c_i$. Two transactions T_i and T_j are *concurrent* if $s_i \prec_h c_j$ and $s_j \prec_h c_i$; *and 5*) Any two write operations on the same data item are totally ordered by $\prec_h$. This is called the *version order* of the data item.

B. Snapshot Isolation

Intuitively, snapshot isolation requires that each transaction read data from the system snapshot as of the time the transaction started [7]. It can be formally defined in terms of two properties, Snapshot Read and Snapshot Write [13].

Definition 1: A history h is in *Snapshot Isolation* (SI) if and only if it satisfies

- (Snapshot Read). All reads of transaction T_i occur at T_i's start time.

$$\forall r_i(x_{j \neq i}), w_{k \neq j}(x_k), c_k \in h : \\ (c_j \in h \land c_j \prec_h s_i) \land (s_i \prec_h c_k \lor c_k \prec_h c_j).$$

- (Snapshot Write). No concurrent committed transactions may write the same data item.

$$\forall w_i(x_i), w_{j \neq i}(x_j) \in h \implies (c_i \prec_h s_j \lor c_j \prec_h s_i).$$

Due to the Snapshot Write property, the clause $c_k \prec_h c_j$ in the Snapshot Read property implies $c_k \prec_h s_j$ and $x_k \ll x_j$. Thus, the Snapshot Read property requires that only the *latest* committed version of each data item can be read. Appendix A characterizes SI in terms of anomalies [13].

III. Relaxed Version Snapshot Isolation

Relaxed Version Snapshot Isolation (RVSI) is a natural generalization of SI by relaxing the Snapshot Read property in three ways. RVSI allows a transaction to observe "stale" committed data versions as long as their staleness is bounded or "concurrent" committed data versions as long as the concurrency level is bounded. When a transaction reads more than one data item, say x and y, RVSI also allows it to obtain versions of x and y from two different snapshots, as long as the "distance" between these two snapshots is bounded. We next formally define RVSI.

Suppose that $r_i(x_j)$ is in transaction T_i. We consider two categories of bounded version constraints according to whether the transaction T_j *precedes* or runs *concurrently* with T_i. First, for each individual read operation $r_i(x_j)$ of transaction T_i, x_j can be any one of the latest k committed versions of x.

Definition 2: A history h satisfies k_1-*version bounded backward view* ($k_1 \in \mathbb{N}^+$), denoted $h \in k_1$-BV, if each read of transaction T_i obtains the data item of within the latest k_1 committed versions before T_i starts. Formally,

$$\forall r_i(x_j), w_k(x_k), c_k \in h \ (k = 1, 2, \ldots, m; k \neq j) : \\ \left(c_j \in h \land \bigwedge_{k=1}^{m} (c_j \prec_h c_k \prec_h s_i) \right) \implies m < k_1.$$

Second, a transaction T_i is allowed to read from concurrent update transactions.

Definition 3: A history h satisfies k_2-*version bounded forward view* ($k_2 \in \mathbb{N}$), denoted $h \in k_2$-FV, if each read of transaction T_i obtains the data item of within the earliest k_2 committed versions updated by transactions concurrent with T_i. Formally,

$$\forall r_i(x_j), w_k(x_k), c_k \in h \ (k = 1, 2, \ldots, m; k \neq j) : \\ \left(c_j \in h \land \bigwedge_{k=1}^{m} (s_i \prec_h c_k \preceq_h c_j) \right) \implies m \le k_2.$$

On the other hand, suppose that $r_i(x_j)$ and $r_i(y_l)$ are two read operations in transaction T_i. We constrain the "distance" between the two snapshots created by transaction T_j and transaction T_l, respectively.

Definition 4: A history h satisfies k_3-*version bounded snapshot view* ($k_3 \in \mathbb{N}$), denoted $h \in k_3$-SV, if a transaction T_i reads committed versions x_j and y_l ($x \neq y, j \neq l$), and assume, without loss of generality, that $c_j \prec_h c_l$, then in the time interval (c_j, c_l) there are at most k_3 other committed versions of x which are fresher than x_j. Formally,

$$\forall r_i(x_j), r_i(y_l), w_k(x_k), c_k \in h \\ (k = 1, 2, \ldots, m; k \neq j; j \neq l; x \neq y) : \\ \left(\bigwedge_{k=1}^{m} (c_j \prec_h c_k \prec_h c_l) \right) \implies m \le k_3.$$

As required by k_1-BV, k_2-FV, and k_3-SV, RVSI prevents transactions from reading non-committed data. Finally, RVSI satisfies the Snapshot Write property of SI, which is also referred to as the Write-Conflict Freedom (WCF) property [10].

Definition 5: A history h is in *relaxed version snapshot isolation* if and only if (where, $k_1 \in \mathbb{N}^+, k_2, k_3 \in \mathbb{N}$)

$$h \in k_1\text{-BV} \cap k_2\text{-FV} \cap k_3\text{-SV} \cap \text{WCF}.$$

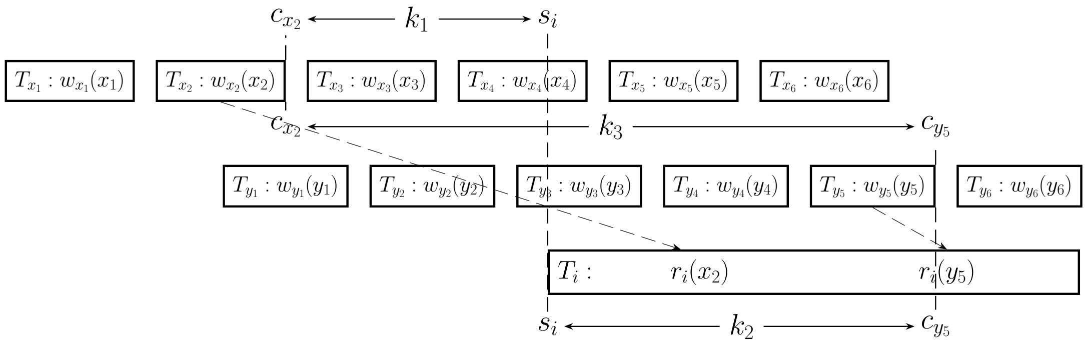

Fig. 1: Illustration of the definition of RVSI, include k_1-BV, k_2-FV, and k_3-SV.

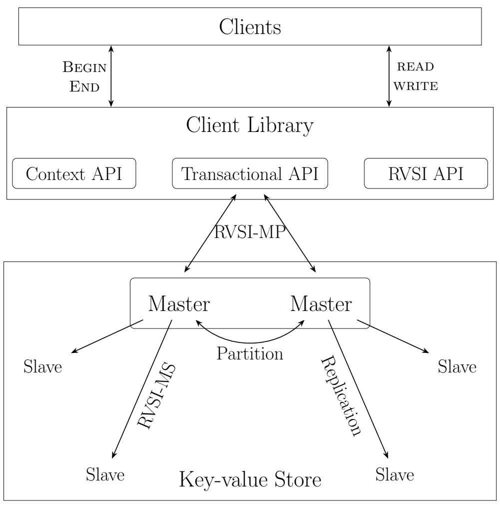

Fig. 2: The three main components of CHAMELEON: the transactional key-value store, the client library, and the RVSI protocol.

Figure 1 illustrates the definition of RVSI. Although k_1, k_2, and k_3 have been defined with respect to the whole transaction, they can be specified for each individual read operation (for k_1-BV and k_2-FV) or every pair of them (for k_3-SV). The properties of RVSI in terms of anomalies are discussed in Appendix B.

IV. System Design of Chameleon

We have implemented a prototype partitioned replicated distributed transactional key-value store called CHAMELEON and deployed it in a multiple data center cloud infrastructure. While achieving RVSI, CHAMELEON allows each individual transaction to dynamically tune its consistency level at runtime. Figure 2 illustrates the three main components of CHAMELEON: the transactional key-value store, the client library, and the RVSI protocol.

A. The Transactional Key-value Store

CHAMELEON uses the classic key-value data model, similar to that in Apache Cassandra [1]. Each key is composed of a row key and a column key. Sites in CHAMELEON are

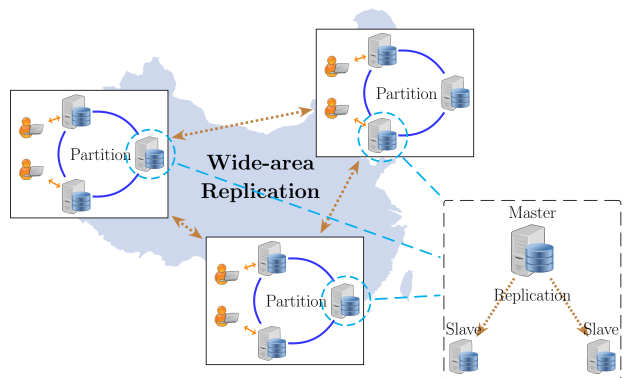

Fig. 3: The partitioned replicated system architecture of CHAMELEON.

divided into master sites and slaves sites, and each master site has several slave sites. For scalability, the whole keyspace is first partitioned among the master sites using, for example, consistent hashing [4]. For fault tolerance and availability, each master site then replicates the keys assigned to it over its slave sites [14], [9], [5], as illustrated in Figure 3. In this way, each key has a primary copy on some master site and several secondary copies on corresponding slave sites. Transactions are first executed and committed on the master sites and modify the primary copies. Committed transactions are then asynchronously propagated to slave sites where the secondary copies are updated [9]. For low latency, clients can read from nearby slave sites.

B. Client Library

The client library provides not only the traditional transaction APIs for beginning/ending a transaction and reading/writing data items, but also APIs for specifying RVSI constraints. Figure 4 illustrates how to write transactions with RVSI specifications using these APIs. `BVSpec` (resp. `FVSpec`) for k_1-BV (resp. k_2-FV) allows to specify different k_1's (resp. k_2's) for different read operations in a transaction, and `SVSpec` for k_3-SV allows to specify k_3's for pairs of read operations. For convenience, `BVSpec` (resp. `FVSpec`) collects a group of read operations with the same value of k_1

```
// Initialize some keys: ck, ck1, ck2
// Create an RVSI transaction
ITx tx = new RVSITx(/** context **/);

// Begin
tx.begin();

// Read and write
ITsCell tsCell = tx.read(ck);
ITsCell tsCell1 = tx.read(ck1);
tx.write(ck1, new Cell("R1C1"));
ITsCell tsCell2 = tx.read(ck2);

// Specify RVSI specs. (e.g., SVSpec)
RVSISpec sv = new SVSpec();
Set<Key> svSet = new HashSet<>();
// Add ck, ck1, ck2 into svSet here
sv.addSpec(svSet, 2);
tx.collectRVSISpec(sv);

// End. ca: committed or aborted
boolean ca = tx.end();
```

Fig. 4: Code snippet for writing RVSI transactions.

(resp. k_2). Similarly, the same value of k_3 in `SVSpec` for a set of read operations applies to every pair of them.

In the context APIs, the client library provides several implementations of data partitioning mechanisms, e.g., range partitioning or consistent hashing [4].

C. RVSI Protocol

The RVSI protocol has two parts: RVSI-MS for data replication in the master-slave architecture (Section V-A) and RVSI-MP for data partitioning across multiple partitions (Section V-B). RVSI-MS enforces RVSI in the simplified single-master, multiple-salves architecture. The issues to address include how to calculate RVSI constraints based on the RVSI specification, when to check the RVSI constraints, and how to handle the read/write requests. RVSI-MP extends the RVSI-MS protocol to support atomic commitment of distributed transactions over multiple partitions/masters.

V. The RVSI Protocol

We first propose the RVSI-MS protocol for enforcing RVSI in a Master-Slave data replication system, and then the RVSI-MP protocol for supporting Multiple Partitions.

A. The RVSI-MS Protocol for Replication

We describe the RVSI-MS protocol in terms of how its participants (i.e., clients, the master site, and slave sites) handle the events they are responsible for and how they interact with each other. The events consist of clients issuing a transaction (including BEGIN, READ, WRITE, and END), the master responding to BEGIN or END from clients by starting (START) or committing (COMMIT) a transaction respectively, and the master and slaves responding to READ from clients. In addition, the master will send messages to slaves, triggering their RECEIVED events.

The RVSI-MS protocol is shown in Algorithm 1 of Appendix C-A, where **remote call** is synchronous and blocking while **broadcast** is asynchronous and non-blocking. We now assume that in a transaction a write is not followed by any read on the same data item. The general case is discussed in Section V-A2.

1) The RVSI-MS Protocol: We first present the concurrency control scheme used in the RVSI-MS protocol, focusing on how a transaction executes at the master and the slaves. It leaves the stubs for enforcing the RVSI version constraints we discuss later in Section V-A2.

All transactions are started and committed/aborted on the master (Lines 2 and 9). The master employs an Multi-Version Concurrency Control (MVCC) protocol to locally implements SI and the first-committer-wins rule [13]. Each transaction T is assigned a globally unique start-timestamp (denoted $T.\mathit{sts}$) and a commit-timestamp (denoted $T.\mathit{cts}$).

Each read operation can be issued to any site which holds the data requested. The read of a transaction T issued to the master will obtain the latest data version before the timestamp $T.\mathit{sts}$, under the control of the local MVCC protocol on the master (Line 13). All write operations of a transaction T are buffered in $T.\mathit{writes}$ at the client side until T is about to commit (Line 6). At that moment, the client first calculates the version constraints, denoted $T.\mathit{vc}$, in terms of k_1, k_2, and k_3, based on the results of its read operations and the RVSI specification (Line 8). Then $T.\mathit{vc}$, along with $T.\mathit{writes}$, is issued to the master, which decides whether T can be committed by checking the version constraints (Line 15) and the write-conflict freedom property.

To commit a transaction T, its updates $T.\mathit{writes}$ are installed on the master and will be lazily propagated to the slaves (Lines 18–22). The updates are then performed at each slave site, refreshing the latest data versions it has ever seen (Line 27). At each slave site, the updates of different transactions are performed neither necessarily atomically nor in their commit order.

2) Calculating Version Constraints: Before showing how to calculate the RVSI version constraints on the read operations of a transaction, namely, to implement the ADD-VC procedure at the client side (Line 8), we first introduce some notations involving data versions.

On the master site, each version of a data item x is associated with a globally unique timestamp, denoted $x.\mathit{ts}$, which equals the commit-timestamp of the transaction that installs this data version. Thus, timestamps induce a total order on all the versions of each data item on the master. We denote the position of some version of a data item x in the total order by $x.\mathit{ord}$ (*ord* stands for "ordinal"). A version of data item x, $x.\mathit{ver}$, is denoted by a triple $(x.\mathit{ts}, x.\mathit{ord}, x.\mathit{val})$, where $x.\mathit{val}$ denotes its value. For notational convenience, we define, for each data item x, a mapping $\mathcal{O}_x$ which takes as input a timestamp t and returns the ordinal number of the latest version of x committed before or at t. That is,

$$\mathcal{O}_x(t) = \max\{x.\mathit{ord} \mid x.\mathit{ts} \le t\}.$$

With these mappings maintained on the master site, we calculate the version constraints for k_1-BV, k_2-FV, and k_3-SV separately. In the following, we assume that transaction T_i is about to commit and calls the procedure ADD-VC (Line 8).

1) *Version constraint for* k_1-BV. Suppose that $r_i(x_j) \in T_i$ (namely, the read operation $r_i(x)$ of transaction T_i returns the version x_j installed by transaction T_j). Add the constraint

$$\mathcal{O}_x(T_i.sts) - \mathcal{O}_x(T_j.cts) < k_1$$

to transaction T_i's version constraint $T_i.vc$. Since x_j is associated with the commit timestamp of T_j, we have $\mathcal{O}_x(T_j.cts) \equiv ord(x_j)$.

2) *Version constraint for* k_2-FV. Suppose that $r_i(x_j) \in T_i$. Add the constraint

$$\mathcal{O}_x(T_j.cts) - \mathcal{O}_x(T_i.sts) \leq k_2$$

to transaction T_i's version constraint $T_i.vc$.

3) *Version constraint for* k_3-SV. Suppose that $r_i(x_j), r_i(y_l) \in T_i$. Without loss of generality, we assume that $T_j.cts < T_l.cts$. Add the constraint

$$\mathcal{O}_x(T_l.cts) - \mathcal{O}_x(T_j.cts) \leq k_3$$

to transaction T_i's version constraint $T_i.vc$.

As mentioned before, we have assumed in RVSI-MS, that in a transaction a write is not followed by any read on the same data item. If it is not the case, the "read-your-writes" rule is used: the read operation after some write operation in the same transaction on the same data item reads from the write buffer. In this way, both the k_1-BV and k_2-FV version constraints are trivially satisfied. Suppose that $r_i(y)$ reads from $w_i(y_i)$ of the same transaction T_i and that $r_i(x_j) \in T_i$. Given that $T_j.cts < T_i.sts$, the version constraint for the k_3-SV specification involving x_j and y_i is

$$\mathcal{O}_x(T_i.cts) - \mathcal{O}_x(T_j.cts) \leq k_3.$$

In other words, k_3-SV reduces to k_1-BV. Similarly, it reduces to k_2-FV if $T_j.cts > T_i.sts$.

B. The RVSI-MP Protocol for Partition

Distributed transactions spanning multiple masters/data partitions need to be committed atomically, namely, all participating masters should agree on whether to commit or abort a transaction. To this end, we adopt the classic two-phase commit (2PC) protocol [12]. The main task of the RVSI-MP protocol (Algorithm 2 of Appendix C-B) for committing distributed transactions with RVSI specification is to integrate the calculation and checking of RVSI version constraints into the prepared phase and the commit phase of the 2PC protocol.

Since distributed transactions span multiple master sites, the way of obtaining unique timestamps from the single master in RVSI-MS does not work any more. Instead, RVSI-MP assumes a timestamp oracle which generates globally unique timestamps [15] (Line 6). Clients make a request of the timestamp oracle for the start-timestamp of a transaction when the transaction begins (Line 2).

The participants involved in committing a transaction in RVSI-MP consist of the client, the timestamp oracle $\mathcal{T}$, a coordinator $\mathcal{C}$ for the 2PC protocol, and the master sites the transaction spans. When issuing a transaction T via END (Line 3), the client first calculates the RVSI version constraints $T.vc$ as described in Section V-A2 (Line 4), and then asks a coordinator to commit it (Line 5).

The coordinator responsible for performing the 2PC protocol among multiple masters first splits the RVSI version constraints $T.vc$ and the updates $T.writes$ into groups, one per master (Line 9) according to the data partitioning strategy. Although the k_3-SV specification involves two data items (e.g., x_j and y_l in Definition 4), the version constraint for it only refers to one of them (depending on whether $T_j.cts$ is larger than $T_l.cts$; see Section V-A2). Thus, the split step for RVSI constraints applies not only to k_1-BV and k_2-FV, but also to k_3-SV that involves two data items. Then, the coordinator brings the masters into the prepare phase of the 2PC protocol by invoking PREPARE (Line 11). In the prepare phase, each master independently checks all the version constraints assigned to it and the write-conflict freedom property (Line 24). If no violations occur on either master, the distributed transaction is ready to commit (Line 13). At the beginning of the commit phase, the coordinator first obtains a globally unique timestamp $T.cts$ from the timestamp oracle as the commit-timestamp of this transaction (Line 14) and then asks each master to commit. As in the RVSI-MS protocol, the masters apply updates locally and propagate them to slaves.

We discuss two additional issues (i.e., atomicity of the commit-timestamps and entity group) about the RVSI-MP protocol in Appendix C-C.

VI. EXPERIMENTAL EVALUATION

In this section we evaluate the impacts of RVSI specification on the transaction abort rates in various scenarios. Generally, the experimental results demonstrate that RVSI helps to reduce the transaction abort rates when applications are willing to tolerate certain anomalies. More specifically, we first deploy the CHAMELEON prototype system on Aliyun and observe that most transactions have been aborted because of violating k_2-FV. That is, on Aliyun the parameter k_2 plays a significant role in reducing transaction abort rates. To fully understand when the parameter k_1 for k_1-BV take effect, we explore more scenarios in controlled experiments by identifying and adjusting three kinds of delays among clients, masters, and slaves. We find that which one of the two parameters k_1 and k_2 plays a major role in reducing transaction abort rates relies on the issue delays between clients and replicas. In addition, given that k_1-BV and k_2-FV is (slightly) relaxed, increasing the value of k_3 helps to further reduce the transaction abort rates. Section VI-A presents the experiments on Aliyun and Section VI-B presents the controlled ones on local hosts.

A. Experiments on Aliyun

We deploy the CHAMELEON prototype system on Aliyun which comprises 3 data centers located in East China (ec), North China (nc), and South China (sc), respectively. Each data center comprises 3 nodes labeled, for example, ec1, ec2, and ec3. All nodes are with the same configuration: a single CPU, 2048MB main memory, and 2Mbps network. In CHAMELEON, nodes ec1, nc2, and sc3 are designated as masters, with nc1 and sc1 being slaves of ec1, ec2 and sc2 of nc2, and ec3 and nc3 of sc3. In this way, replicas are stored across data centers. Master nodes can serve as coordinators of the 2PC protocols. The timestamp oracle service is on ec1. The clients which issue transactions are on the hosts in our lab (located in East China).

According to the ping logs for a week among these 9 nodes[4], the 95th percentile of (one-way) communication delays among nodes within the same data center is about $1 \sim 2$ms, while that among nodes across data centers is about $15 \sim 25$ms. Additional ping experiments show that the communication delays among clients and replicas range from 15ms to 20ms.

1) Workload Parameters and Metrics: In the experiments on Aliyun, we explore the following three categories of workload parameters listed in Table I.

Transaction-related parameters. The parameter #keys denotes the number of keys stored in CHAMELEON. In these experiments, we assume a small number of keys such that concurrent transactions are more likely to access the same data item. The parameter mpl denotes the number of clients, each of which will issue #txs/client transactions. The number of operations in each transaction is characterized by a Binomial random variable #ops/tx. The ratio of the number of read operations to that of write operations among all #ops/tx operations in a transaction is denote by rwRatio. We explore three cases of rwRatio $= 1:2$, rwRatio $= 1:1$, and rwRatio $= 4:1$, representing write-frequent, read-write balanced, and read-frequent workloads, respectively. The data item accessed by each operation is determined by a Zipfian distribution with parameters of #keys and zipfExponent, which is often used to generate workloads with "hotspot" data items [16].

Execution-related parameters. We use three parameters — minInterval, maxInterval, and meanInterval — to control the time interval between the finishing of a transaction and the starting of the next one. The actual time interval is computed as follows: Take a sample of an exponential random variable with parameter meanInterval, add it to minInterval, and take the minimum of the result and maxInterval.

RVSI-related parameters. We explore six RVSI specifications as listed in Table I. To evaluate the impacts of RVSI on the transaction abort rates, we adjust parameters mpl, rwRatio, and (k_1, k_2, k_3), as shown in Table I.

2) Experimental Results: Transactions abort for two reasons: One is "vc-aborted", namely, a transaction has obtained data items that do not satisfy the RVSI version constraints. The other is "wcf-aborted", namely, the write-conflict freedom property has been violated. The experimental results show that transaction abort rates due to "vc-aborted" are sensitive to different values of k_1, k_2, or k_3, but those due to "wcf-aborted" are not. This is because RVSI relaxes the Snapshot Read property of SI but not the Snapshot Write property.

[4]See https://github.com/hengxin/aliyun-ping-traces for raw data.

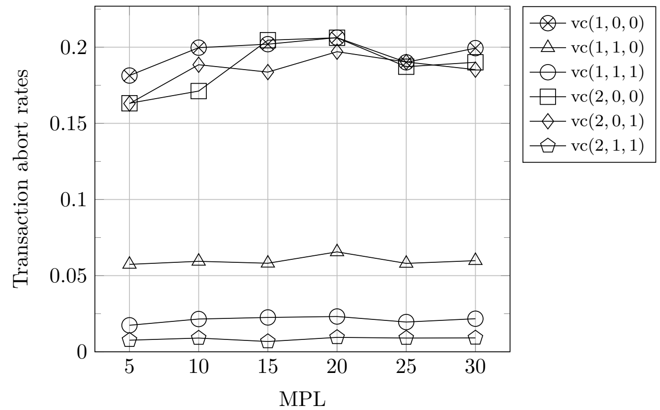

Fig. 5: The transaction abort rates due to "vc-aborted" under read-frequent workloads.

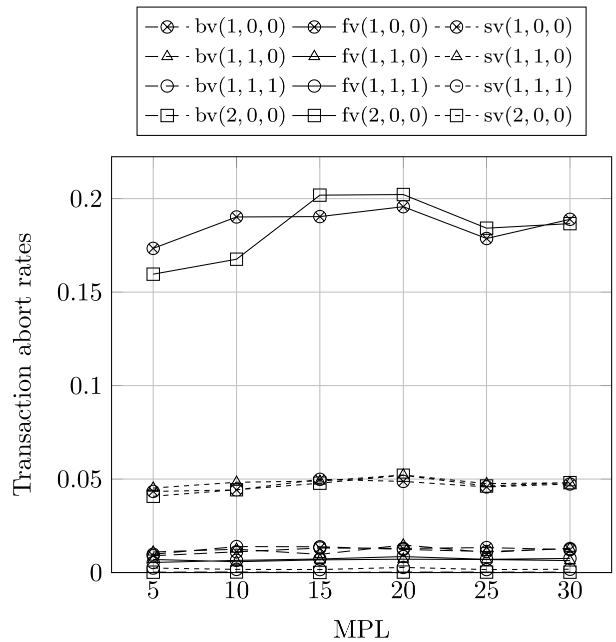

Fig. 6: The transaction abort rates because of violating k_1-BV, k_2-FV, or k_3-SV under read-frequent workloads.

Generally, as rwRatio increases, the transaction abort rates due to "vc-aborted" grow accordingly. Figure 5 shows the transaction abort rates due to "vc-aborted", denoted $vc(*,*,*)$, under read-frequent workloads[5]. It shows that the transaction abort rates due to "vc-aborted" can be greatly reduced by slightly increasing the values of k_1, k_2, or k_3. For example, mpl $= 30$, $vc(1,0,0)$ is 0.1994, while $vc(2,1,1)$ decreases it to 0.0091. Figure 6 explores the details of the transaction abort rates due to "vc-aborted" in Figure 5 by decomposing each $vc(*,*,*)$ into $bv(*,*,*)$ for "k_1-BV-aborted", $fv(*,*,*)$

[5]More experimental results can be found at https://github.com/hengxin/chameleon-transactional-kvstore.

TABLE I: Three categories of workload parameters for experiments on Aliyun and three kinds of delays for controlled experiments on local hosts.

Parameter	**Type**	**Value**	**Explanation**
#keys	Fixed	25 = 5 (rows) $\times$ 5 (columns)	size of keyspace
mpl	Variable	5, 10, 15, 20, 25, 30	multiprogramming level: number of concurrent clients
#txs/client	Fixed	1000	number of transactions per client
#ops/tx	Random	$\sim$ Binomial(20, 0.5)	number of operations per transaction
rwRatio	Variable	1:2, 1:1, 4:1	#reads/#writes
zipfExponent	Fixed	1	parameter for Zipfian distribution
minInterval	Fixed	0ms	minimum inter-transaction time
maxInterval	Fixed	10ms	maximum inter-transaction time
meanInterval	Fixed	5ms	mean inter-transaction time for exponential distribution
(k_1, k_2, k_3)	Variable	(1,0,0) (1,1,0) (1,1,1) (2,0,0) (2,0,1) (2,1,1)	for k_1-BV, k_2-FV, and k_3-SV
issueDelay	Variable	5, 8, 10, 12, 15, 20	delays between clients and replicas
replDelay	Variable	5, 10, 15, 20, 30	delays between masters and slaves
2pcDelay	Variable	10, 20, 30, 40, 50	delays among masters

for "k_2-FV-aborted", and $sv(*,*,*)$ for "k_3-SV-aborted" [6]. It turns out that most "vc-aborted" transactions in the Aliyun scenario have been aborted because of violating k_2-FV (compared to those violating k_1-BV), and correspondingly that such abort rates can be greatly reduced by slightly increasing the value of k_2. For example, with mpl $= 30$, $fv(1,0,0)$ is 0.1889 and $fv(2,0,0)$ is 0.1866, while $fv(1,1,0)$ is 0.0064. Moreover, given $k_2 \geq 1$, increasing the value of k_3 for k_3-SV helps to further reduce the transaction abort rates. For example, $sv(1,1,0)$ is 0.0480 while $sv(1,1,1)$ is 0.0018.

B. Controlled Experiments on Local Hosts

To fully understand when the parameter k_1 for k_1-BV takes effect, we explore more scenarios in controlled experiments on local hosts. Table I lists the controlled parameters denoting three kinds of (one-way) delays: the issue delays (issueDelay) between clients and replicas (consisting of masters and slaves), the replication delays (replDelay) between masters and its slaves, and the 2PC coordination delays (2pcDelay) among masters. Figure 7 shows that when the issueDelay gets shorter, the impacts of k_2-FV on transactions abort rates go weaker, and on the contrary the impacts of k_1-BV have begun to emerge. For example, with issueDelay $=$ 20ms, $bv(1,0,0)$ is 0.0057 while $fv(1,0,0)$ is 0.0251. With issueDelay $=$ 15ms, $bv(1,0,0)$ is 0.08225, larger than $fv(1,0,0) = 0.0393$. When issueDelay decreases to 5ms, $bv(1,0,0)$ increases to 0.01716, significantly larger than $fv(1,0,0) = 0.0045$. As illustrated in Figure 7, the turning point of the issue delays is about $15 \sim 20$ms. Moreover, given $k_1 \geq 2$, increasing the value of k_3 for k_3-SV helps to further reduce the transaction abort rates. For example, with issueDelay $=$ 5ms, $sv(2,0,0)$ is 0.0950 while $sv(2,0,1)$ is 0.0164.

[6]To get rid of clutter, Figure 6 only shows the data for four RVSI specifications.

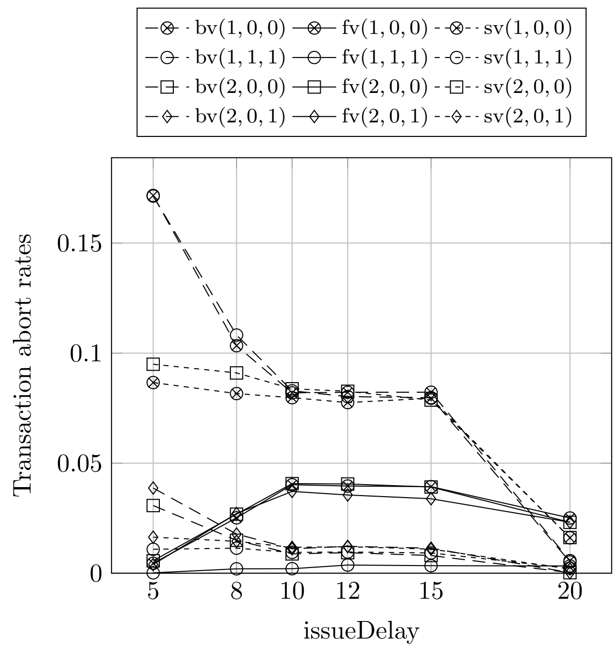

Fig. 7: The transaction abort rates because of violating k_1-BV, k_2-FV, or k_3-SV under write-frequent workloads with respect to issue delays (mpl $= 30$, #txs/client $= 800$).

Combing these experimental results on local hosts and those on Aliyun, we can conclude that which one of the two parameters k_1 for k_1-BV and k_2 for k_2-FV plays a major role in reducing transaction abort rates relies on the issue delays between clients and replicas. The rationale behind is as follows: In the Aliyun scenario, the issue delays between clients out of Aliyun and replicas in Aliyun are relatively longer than those manually set in our controlled experiments on local hosts. With larger issue delays, a transaction lasts longer and more transactions happen to be concurrent with it. These long-lived transactions are more likely to obtain data versions that are updated by concurrent transactions and whether they will be aborted are thus more sensitive to the parameter k_2. Moreover, the larger the rwRatio is, the more significant the impact of k_2-FV on reducing transaction abort rates is. Note that since neither 2pcDelay (i.e., the delays among masters) nor replDelay (i.e., the delays between masters and slaves) is counted in the read procedure, they have shown no direct relationship with the effects of k_1 or k_2 on transaction abort rates. On the other hand, as the issue delays get shorter, the impacts of k_1 for k_1-BV on reducing transaction abort rates emerge and become more significant. In addition, given that k_1-BV or k_2-FV is (slightly) relaxed, increasing the value of k_3 helps to further reduce the transaction abort rates.

VII. Related Work

We divide the related work into five categories.

Distributed partitioned replicated storage. For high scalability, and high availability, and low latency, distributed storage systems are typically partitioned and/or replicated [5]. The whole data set is first partitioned across multiple servers, allowing the system to scale up. Each partition may be then replicated on different servers for high availability and performance. Moreover, in a common system architecture, replication occurs among different data centers for fault-tolerance [17], [6], [18], [19]. We implement our prototype distributed transactional key-value stores in such a partitioned replicated cloud-based infrastructure.

Distributed transactional key-value stores. Recently, transactional semantics, though often with restrictions, has been widely adopted in distributed key-value stores. PNUTS [3] provides serializability on a per-record basis and read-modify-write transactions. BigTable [2] supports single-row transactions but not general transactions across rows. COPS [18] supports read-only transactions to obtain a consistent view of multiple keys. Eiger [19] extends COPS by supporting write-only transactions as well. Percolator [15] supports multi-row transactions with the snapshot isolation semantics. Megastore [17] provides transactional ACID guarantees within individual entity groups which define the *a priori* collection of logically closely related data. Spanner [20] is claimed to be the first globally-distributed storage system that supports externally-consistent (or, "strictly serializable" [21]) distributed transactions.

Our prototype transactional key-value store implements RVSI, an relaxed variants of snapshot isolation, and allows each individual transaction to dynamically tune its consistency level at runtime.

Relaxed variants of snapshot isolation. For improved performance, several relaxed transactional semantics based on snapshot isolation have been proposed. Forward Consistent View (PL-FCV) [13] allows a transaction T_i to observe the updates of transactions that commit after it started, i.e., read "forward" beyond the start point as long as T_i observes a consistent snapshot. Generalized Snapshot Isolation (GSI) [8] allows the use of "older" snapshots rather than the "latest" snapshot only. Strong Session Snapshot Isolation [9] enforces the real-time ordering constraints on transactions that belong to the same session but not across sessions. Parallel Snapshot Isolation (PSI) [6], as well as Causal Snapshot Isolation (CSI) [22], requires that hosts within a site observe transactions according to a consistent snapshot and a common ordering of transactions, but enforces only causal ordering of transactions across sites. Non-Monotonic Snapshot Isolation (NMSI) [23], [10] allows transactions to observe snapshots that are not monotonically ordered.

Unlike RVSI, the relaxed variants mentioned above provide no specification or control of the severity of the anomalies with respect to snapshot isolation. The basic idea of decomposing SI into three "view properties" is inspired by the work on NMSI. In particular, the k_2-FV property of RVSI resembles PL-FCV, except that k_2-FV by itself does not require a consistent snapshot.

Bounded transactional inconsistency. Being an extension of serializability (SR), Epsilon-Serializability (ESR) [24], [25] allows read-only epsilon-transactions to observe database states with bounded inconsistency introduced by concurrent update transactions. In an N-ignorant system [26], a transaction may be ignorant of the results of at most N prior transactions that it would have seen if the execution had been serial. Relaxed Currency and Consistency (C&C) semantics [27] allows a query to specify a currency bound referring to how up-to-date the query result should be and a consistency class grouping the data items that should be from the same snapshot. Relaxed Currency (RC-) Serializability [14] allows update transactions to read stale data satisfying their freshness constraints.

ESR allows read-only epsilon-transactions to observe uncommitted data updated by concurrent transactions. In contrast, RVSI enforces Read Committed (RC), meaning that only committed data is observable. The N-ignorant and RC-serializability semantics extend serializability while RVSI extends snapshot isolation. The RC-serializability and C&C semantics measure the bounded inconsistency by currency in real-time while RVSI does by data versions.

Dynamic consistency choices. Due to the parameters introduced for bounding transactional inconsistency, the systems implementing the SR, N-ignorant, RC-serializability, or C&C semantics naturally support dynamic consistency choices at runtime. Pileus [28] is a transactional key-value store with varying degrees of consistency dynamically chosen by applications. Specifically, each transaction reads from a snapshot determined by its requested consistency, covering strong, intermediate, and eventual consistency. Tripathi *et al.* [29] propose a transaction model which supports four consistency levels which are, from the strongest to the weakest, serializability, CSI, CSI with commutative updates, and CSI with asynchronous concurrent updates. A transaction executes at a specific level in this hierarchy and follows the so-called "Read-Up&Write-Down" rule.

Our prototype transactional key-value store allows each individual transaction to dynamically tune its consistency level at runtime by choosing proper parameter values for k_1-BV, k_2-FV, and k_3-SV specification.

VIII. Conclusion

In this paper we propose the idea of parameterized and runtime-tunable snapshot isolation. We first define a new transactional consistency model called Relaxed Version Snapshot Isolation (RVSI), which can formally and quantitatively specify the anomalies it may produce with respect to SI. We then implement a prototype partitioned replicated distributed transactional key-value store called CHAMELEON across multiple data centers. While achieving RVSI, CHAMELEON allows each individual transaction to dynamically tune its consistency level at runtime.

Through the extensive experiments on both Aliyun and local hosts, we have found that which one of the two parameters k_1 (for k_1-BV) and k_2 (for k_2-FV) of RVSI plays a major role in reducing transaction abort rates relies on the issue delays between clients and replicas. Since k_3-SV of RVSI involves multiple data items, evaluating its individual impacts on transaction abort rates is more challenging. We plan to study it more thoroughly in future work, probably with data mining technologies.

Acknowledgments

This work is supported by the National 973 Program of China (2015CB352202).

References

[1] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '06, 2006, pp. 205–218.

[3] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1277–1288, Aug. 2008.

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07, 2007, pp. 205–220.

[5] D. Agrawal, A. El Abbadi, and K. Salem, "A taxonomy of partitioned replicated cloud-based database systems," *IEEE Data Eng. Bull.*, vol. 38, no. 1, pp. 4–9, 2015.

[6] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, ser. SOSP '11, 2011, pp. 385–400.

[7] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ansi sql isolation levels," in *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD'95, 1995, pp. 1–10.

[8] S. Elnikety, W. Zwaenepoel, and F. Pedone, "Database replication using generalized snapshot isolation," in *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, ser. SRDS '05, 2005, pp. 73–84.

[9] K. Daudjee and K. Salem, "Lazy database replication with snapshot isolation," in *Proceedings of the 32Nd International Conference on Very Large Data Bases*, ser. VLDB '06, 2006, pp. 715–726.

[10] M. S. Ardekani, P. Sutra, and M. Shapiro, "Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems," in *Proceedings of the 32nd IEEE International Symposium on Reliable Distributed Systems*, ser. SRDS '13, 2013, pp. 163–172.

[11] D. Terry, "Replicated data consistency explained through baseball," *Commun. ACM*, vol. 56, no. 12, pp. 82–89, Dec. 2013.

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

[13] A. Adya, "Weak consistency: A generalized theory and optimistic implementations for distributed transactions," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, USA, 1999.

[14] P. A. Bernstein, A. Fekete, H. Guo, R. Ramakrishnan, and P. Tamma, "Relaxed-currency serializability for middle-tier caching and replication," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '06, 2006, pp. 599–610.

[15] D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10, 2010, pp. 251–264.

[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10, 2010, pp. 143–154.

[17] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*, ser. CIDR '11, 2011, pp. 223–234.

[18] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: Scalable causal consistency for wide-area storage with cops," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, ser. SOSP '11, 2011, pp. 401–416.

[19] ——, "Stronger semantics for low-latency geo-replicated storage," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI '13, 2013, pp. 313–328.

[20] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally-distributed database," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI '12, 2012, pp. 251–264.

[21] C. H. Papadimitriou, "The serializability of concurrent database updates," *J. ACM*, vol. 26, no. 4, pp. 631–653, Oct. 1979.

[22] V. Padhye and A. Tripathi, "Causally coordinated snapshot isolation for geographically replicated data," in *Proceedings of the 2012 IEEE 31st Symposium on Reliable Distributed Systems*, ser. SRDS '12, 2012, pp. 261–266.

[23] M. S. Ardekani, P. Sutra, N. M. Preguiça, and M. Shapiro, "Non-monotonic snapshot isolation," *CoRR*, vol. abs/1306.3906, 2013.

[24] C. Pu and A. Leff, "Replica control in distributed systems: As asynchronous approach," in *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '91, 1991, pp. 377–386.

[25] K. Ramamritham and C. Pu, "A formal characterization of epsilon serializability," *IEEE Trans. on Knowl. and Data Eng.*, vol. 7, no. 6, pp. 997–1007, 1995.

[26] N. Krishnakumar and A. J. Bernstein, "Bounded ignorance in replicated systems," in *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, ser. PODS '91, 1991, pp. 63–74.

[27] H. Guo, P.-A. Larson, R. Ramakrishnan, and J. Goldstein, "Relaxed currency and consistency: How to say "good enough" in sql," in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '04, 2004, pp. 815–826.

[28] R. Kotla, M. Balakrishnan, D. Terry, and M. K. Aguilera, "Transactions with consistency choices on geo-replicated cloud storage," Microsoft, MSR-TR-2013-82, Tech. Rep., Sep. 2013.

[29] A. Tripathi and B. Thirunavukarasu, "A transaction model for management of replicated data with multiple consistency levels," in *Proceedings of the 2015 IEEE International Conference on Big Data*, ser. BigData'15, 2015, pp. 470–477.

[30] S. Patterson, A. J. Elmore, F. Nawab, D. Agrawal, and A. El Abbadi, "Serializability, not serial: Concurrency control and availability in multi-datacenter datastores," *Proc. VLDB Endow.*, vol. 5, no. 11, pp. 1459–1470, Jul. 2012.

APPENDIX A
CHARACTERIZATION OF SI IN TERMS OF ANOMALIES

Following the terminology and notations in [13], SI identifies exactly the set of histories which avoid the *anomalies* of *G-1* and *G-SI* (consisting of *G-SIa* and *G-SIb*).

G-1 is defined through three properties of histories that are simple to check (and we omit the details here): *1) G1a*: Aborted Reads; *2) G1b*: Intermediate Reads; *and 3) G1c*: Circular Information Flow.

G-SI is defined through *SSG*, the "Start-ordered Serialization Graph" [13]. An *SSG* of a history h, denoted *SSG(h)*, is a directed graph with committed transactions in h as nodes and several types of dependencies as directed edges. Given two committed transactions T_i and T_j in h, we consider four possible types of dependencies between them:

1) T_j *start-depends (s-depends)* on T_i, denoted $T_i \xrightarrow{s} T_j$, if T_j starts after T_i commits, i.e., $c_i \prec_h s_j$;
2) T_j *directly read-depends (wr-depends)* on T_i, denoted $T_i \xrightarrow{wr} T_j$, if T_j reads the data version of, say x, written by T_i, i.e., $r_j(x_i) \in h$;
3) T_j *directly write-depends (ww-depends)* on T_i, denoted $T_i \xrightarrow{ww} T_j$, if they both write the same data item x, and x_i and x_j are *consecutive* versions of x in h's version order.
4) T_j *directly anti-depends (rw-depends)* on T_i, denoted $T_i \overset{rw}{\dashrightarrow} T_j$, if T_i reads version x_k written by a third committed transaction T_k, and T_j creates x's next version x_j (after x_k) in h's version order.

The *G-SIa* and *G-SIb* anomalies are defined as follows:

- *G-SIa: Interference.* A history h exhibits the *G-SIa* anomaly if *SSG(h)* contains a wr/ww-dependency edge from T_i to T_j, but without an s-dependency edge from T_i to T_j.
- *G-SIb: Missed Effects.* A history h exhibits the *G-SIb* anomaly if *SSG(h)* contains a directed cycle with exactly one rw-dependency edge.

APPENDIX B
ANOMALIES ALLOWED BY RVSI

RVSI is parameterized with three variables k_1, k_2, and k_3, controlling what types of anomalies RVSI allows and how severely it deviates from SI. In this section we examine some typical cases of RVSI, denoted RVSI(k_1, k_2, k_3). (Irrelevant variables in some cases are denoted by $*$'s.) First of all, no additional anomalies with respect to SI are introduced in the case of RVSI$(1, 0, *)$.

Theorem 1: RVSI$(1, 0, *)$ is equivalent to SI.

Proof: The proof of SI $\subseteq$ RVSI$(1, 0, *)$ is easy and thus omitted. In the following, we prove that RVSI$(1, 0, *) \subseteq$ SI. Consider some history $h \in$ RVSI$(1, 0, *)$. We shall prove that $h \in$ SI by showing that h does not exhibit the anomalies of *G1* or *G-SI*. Informally, *G1* requires that a transaction T_i can commit only if T_i has only read the updates of transactions that have committed by the time T_i commits [13], which is the case in RVSI. The formal proof is omitted here.

h does not exhibit G-SIa. Suppose that *G-SIa* is allowed and there is a wr-/ww-dependency from T_i to T_j in *SSG(h)* without a corresponding s-dependency (i.e., $(T_i \xrightarrow{s} T_j) \notin$ *SSG(h)*). If $(T_i \xrightarrow{ww} T_j) \in$ *SSG(h)*, then $h \notin$ WCF. If $(T_i \xrightarrow{wr} T_j) \in$ *SSG(h)*, then $s_i \prec_h c_j$. Therefore, we have $k_2 \geq 1$, contradicting $k_2 = 0$.

h does not exhibit G-SIb. Suppose that *G-SIb* is allowed and *SSG(h)* contains a cycle of the form $\langle T_1, T_2, \ldots, T_n, T_1 \rangle$ with exactly one rw-dependency edge. Without loss of generality, suppose that the only rw-dependency edge is from T_1 to T_2, that $T_1 \overset{rw}{\dashrightarrow} T_2$ is due to the data item x, and that $r_1(x_k) \in T_1$, $w_2(x_2) \in T_2$. There exists another transaction T_k which writes x and x_k is the immediately previous version of x before x_2. Therefore, we have $c_k \prec_h s_2 \prec_h c_2$ due to $T_k \xrightarrow{ww} T_2$. We also have $c_2 \prec_h s_3 \prec_h c_3 \prec_h \cdots \prec_h s_n \prec_h c_n \prec_h s_1$ due to the other dependency edges. Putting it together, we obtain $c_k \prec_h c_2 \prec_h s_1$ in h and conclude that $k_1 \geq 2$, contradicting $k_1 = 1$. ■

$\boxed{T_1 : r_1(x) = 0, r_1(y) = 0, w_1(x) = 1}$ $\boxed{T_2 : r_2(x) = 1, r_2(y) = 0}$

$\boxed{T_3 : r_3(x) = 0, r_3(y) = 0, w_3(y) = 1}$ $\boxed{T_4 : \boldsymbol{r_4(x) = 0}, r_4(y) = 1}$

Fig. 8: A history in RVSI$(2, 0, \infty)$ which exhibits the "write skew" anomaly. To satisfy SI, $r_4(x) = 0$ in transaction T_4 should be $r_4(x) = 1$.

As with SI, RVSI$(>1, 0, *)$ allows the *write skew* anomaly where concurrent transactions make updates to different data items causing the state to fork [7], [6]. For example, in Figure 8, concurrent transactions T_1 and T_3 both read from the same data items x and y, but then update different ones. With SI, the state merges after both transactions commit. Therefore, the transaction T_4 in Figure 8 should observe the updated x and y. RVSI$(2, 0, \infty)$, however, allows T_4 to observe the stale x by looking "backward". Similarly, RVSI$(1, >0, \infty)$ allows a transaction to observe the updates of transactions that commit after it started by looking "forward" beyond its start time. Lastly, RVSI$(>1, >0, >1)$ allows a transaction to obtain versions of different data items from different snapshots.

APPENDIX C
THE RVSI PROTOCOL

A. The RVSI-MS Protocol for Replication

Algorithm 1 shows the RVSI-MS protocol for replication.

B. The RVSI-MP Protocol for Partition

Algorithm 2 shows the RVSI-MP protocol for partition.

C. Additional Issues about the RVSI-MP Protocol

Atomicity of the commit-timestamps. The definition of SI explicitly refers to global timestamps [7]. Particularly, for any two transactions, SI requires that $c_i \prec_h s_j$ or $s_j \prec_h c_i$. In implementation, we rely on additional concurrency control mechanism to ensure the "atomicity of the commit-timestamps" of transactions: No transactions can obtain the

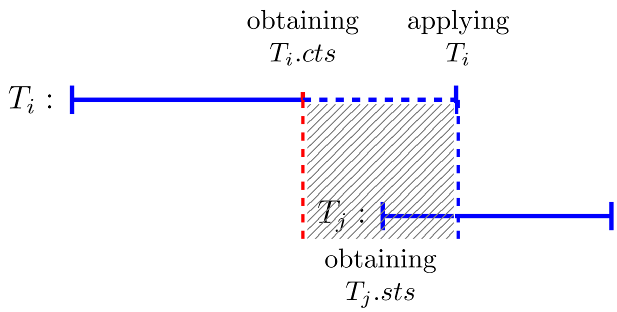

Fig. 9: Illustration of the atomicity of the commit-timestamp of transaction T_i.

start-timestamps from the time some transaction has just obtained its commit-timestamp to the time its updates have been actually applied; as illustrated in Figure 9. This can be achieved by synchronizing the timestamp oracle and the coordinator for the 2PC protocol (omitted in Algorithm 2).

Entity group. Recall that a typical k_3-SV specification involves two transactions (e.g., T_j and T_l) and two data items (e.g., $w_j(x_j) \in T_j$ and $w_l(y_l) \in T_l$), and its corresponding version constraint refers to $\mathcal{O}_x(T_l.cts)$. This implies that the timestamps associated with T_j and T_l should be comparable, although they update different data items. Nevertheless, if we know *a prior* that some data items will not appear in the same k_3-SV specification, then we can split all the data items into several independent *entity groups* [17], [30]. Each entity group performs a separate instance of the RVSI-MP protocol, using its own timestamp oracle.

Algorithm 1 RVSI-MS: RVSI Protocol for Replication (for Executing Transaction T).

Client-side methods:

procedure BEGIN()
  $T.sts \leftarrow$ **rpc-call** START() at master $\mathcal{M}$
procedure READ(x)
  $x.ver \leftarrow$ **rpc-call** READ(x) at any site
procedure WRITE(x, v)
  add (x, v) to $T.writes$
procedure END(T)
  $T.vc \leftarrow$ ADD-VC()
  $c/a \leftarrow$ **rpc-call** COMMIT($T.writes, T.vc$) at $\mathcal{M}$

Master-side data structures and methods:

$\mathcal{M}.ts$: for start-timestamps and commit-timestamps
$\{x.ver = (x.ts, x.ord, x.val)\}$: set of versions of x

procedure START()
  return ++$\mathcal{M}.ts$
procedure READ(x)
  return the latest $x.ver$ installed
procedure COMMIT($T.writes, T.vc$)
  if CHECK-VC($T.vc$) && write-conflict freedom **then**
    $T.cts \leftarrow$ ++$\mathcal{M}.ts$
    ▷ *apply $T.writes$ locally and propagate it*
    $T.upvers = \emptyset$ ▷ *collect updated versions*
    for $(x, v) \in T.writes$ **do**
      $x.new\text{-}ver \leftarrow (T.cts, ++x.ord, v)$
      add $x.new\text{-}ver$ to $\{x.ver\}$ and $T.upvers$
    broadcast $\langle$PROP$, T.upvers\rangle$ to slaves
    return c denoting "committed"
  return a denoting "aborted"

Slave-side data structures and methods:

$x.ver = (x.ts, x.ord, x.val)$: the latest version of x

procedure READ(x)
  return $x.ver$
upon RECEIVED($\langle$PROP$, T.upvers\rangle$)
  for $\big(x.ver' = (x.ts', x.ord', x.val')\big) \in T.upvers$ **do**
    if $x.ord' > x.ord$ **then**
      $x.ver \leftarrow x.ver'$

Algorithm 2 RVSI-MP: RVSI Protocol for Partition (for Executing Transaction T).

Client-side methods:

procedure BEGIN()
  return rpc-call GETTS() at $\mathcal{T}$
procedure END()
  $T.vc \leftarrow$ ADD-VC()
  $c/a \leftarrow$ **rpc-call** C-COMMIT($T.writes, T.vc$) at $\mathcal{C}$

Timestamp oracle methods:

$\mathcal{T}.ts$: for start-timestamps and commit-timestamps

procedure GETTS()
  return ++$\mathcal{T}.ts$

Coordinator-side data structures and methods:

The coordinator $\mathcal{C}$ executes the 2PC protocol with masters $\mathcal{M}$ involved in T.

procedure C-COMMIT($T.writes, T.vc$)
  split $T.writes$ and $T.vc$ with the data partitioning strategy
  ▷ *the prepare phase:*
  rpc-call PREPARE($T.writes, T.vc$) at each $\mathcal{M}$
  ▷ *the commit phase:*
  if all PREPARE($T.writes, T.vc$) return *true* **then**
    $T.cts \leftarrow$ **rpc-call** GETTS() at $\mathcal{T}$
    rpc-call COMMIT($T.cts, T.writes$) at each $\mathcal{M}$
  else
    rpc-call ABORT() at each $\mathcal{M}$
    return a denoting "aborted"
  if all COMMIT($T.cts, T.writes$) return *true* **then**
    return c denoting "committed"
  else
    return a denoting "aborted"

Master-side methods:

procedure PREPARE($T.writes, T.vc$)
  return CHECK-VC($T.vc$) && write-conflict freedom
procedure COMMIT($T.cts, T.writes$)
  ▷ *apply $T.writes$ locally and propagate it*
procedure ABORT()
  ▷ *abort*