Distributed Computing Column 51
Large-Scale Transaction Replication

Idit Keidar
Dept. of Electrical Engineering, Technion
Haifa, 32000, Israel
idish@ee.technion.ac.il

Distributed storage systems have now become mainstream, partly due to exponential growth in data volumes. It is common for applications nowadays to access data that is partitioned or sharded across many nodes, and in addition, partially replicated. Large Internet companies further deploy applications that span multiple data centers.

For a few years, it was common to settle for weak consistency in such settings, adhering to the so-called "NoSQL" approach. Indeed, an article that appeared in this column four years ago[1] reported a "fear of synchronization" among chief architects who had built large-scale distributed systems in industry. But in the past couple of years, we see this trend reversing. Both academic and industrial projects have recognized the need for strong consistency and so-called ACID transactions in large distributed storage systems. Such strong consistency is the topic of the current column.

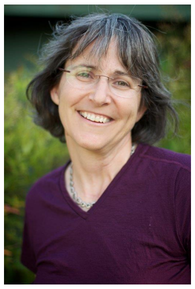
Our first contribution, by Dahlia Malkhi and Jean-Philippe Martin explains the concurrency control mechanism of Google's Spanner system. Our second contribution, by Ittay Eyal, discusses more generally fault-tolerant architectures for distributed ACID transactions. Many thanks to Dahlia, Jean-Philippe, and Ittay for their contributions!

Farewell. On a personal note, this is my last issue as editor of the distributed computing column. I have truly enjoyed performing this role for the past six years. Above all, I am grateful to the many authors who have contributed interesting and enlightening articles on diverse topics. They have made my job both easy and gratifying. Yet after six years, I feel that it is time to step down, and bring in a fresh perspective. I leave the column in the capable hands of Jennifer Welch, who will replace me as of the next issue. I wish Jennifer that her tenure as editor will be at least as enjoyable as mine was.

[1] "Toward a Cloud Computing Research Agenda" by Ken Birman, Gregory Chockler, and Robbert van Renesse, Distributed Computing Column 34, SIGACT News Volume 40, Number 2, June 2009, pages 68–80.

Spanner's Concurrency Control

Dahlia Malkhi Jean-Philippe Martin
Microsoft Research, Silicon Valley
{dalia,jpmartin}@microsoft.com



Abstract

The Spanner project reports that one can build practical large-scale systems that combine strong semantics with geo-distribution. In this review manuscript, we provide insight on how Spanner's concurrency control provides both read-only transactions which avoid locking data, and strong consistency.

1 Data Model

Spanner [2] is a transactional data service. It stores a collection of objects. Objects are partitioned among a set of servers. Each server is itself replicated using Paxos-driven state-machine replication, with a designated group leader at any moment in time. For most of the discussion, we gloss over replication internals, and treat each group of replicas as an undivided server. We denote objects with capital letters, e.g., A, B, C.

2 Read/Write Transaction Atomicity

R/W transactions are managed using strict concurrency control [1], which means that every data item accessed by a transaction is first locked, and no two concurrent transactions may hold a lock to the same data item if one of the locks is for write. All data objects modified by a transaction become visible only upon transaction commit time, making transactions effectively atomic. Spanner adopts a usual trick to enhance concurrency ("optimistic execution until commit") as follows. Each transactions defers its write-locks until commit time, at which time it performs two-phase commit protocol [1]: all buffered writes by the transaction attempt to acquire locks; data objects are actually updated only if the transaction commits, and then updates become externally visible. To illustrate this, say we have two objects, A and B, initially set to zero, and two example transactions as shown in Figure 1.

In the example, locking prevents EX-T1 incrementing A to 1 while simultaneously EX-T2 sets B to 1. Clearly, unconstrained lock acquisition may result in a deadlock.

EX-T1: Read B and, if zero, increment A
EX-T2: Read A, and set B to $A + 1$

Figure 1: Running example with two transactions.

wound-wait is a method for performing distributed concurrency control in a deadlock-free manner. When a transaction requests to prepare an operation (read/write) which conflicts with another read/write operation in an on-going transaction, we select to either wait or wound based on transaction unique ids, where lower-id takes precedence and wounds, while higher-id waits:

wait: delay until the conflicting transaction terminates (via either abort or commit)

wound: broadcast a request for the conflicting transaction to restart and wait until it actually rewinds (or terminates). In some implementations, the wounded transaction may greedily continue executing, unless it incurs a waiting state, in which case it rewinds

Figure 2: Wound-Wait.

Spanner employs a two-phase locking protocol with wound-wait [4] lock acquisition strategy to prevent deadlocks. EX-T1 may arrive after EX-T2 acquired a read-lock on A and a write-lock on B; it causes EX-T2 to rewind, and sets A to 1; then EX-T2 re-executes and sets B to 2.

So far, there are no versions, timestamps, or any of the complicated mechanisms which make up most of the Spanner work. We now add fast read-only transactions, which add a whole new dimension to the system.

3 Read-Only Transactions

Most data accesses are for reading purposes only, and moreover, may entail long-lived computations such as data analysis. We often want to execute transactions which do not modify data differently than R/W transactions, and allow them to proceed without locking out R/W transactions over the same data. Spanner names such read-only transactions *lock-free*, to indicate that the transaction itself does not lock any data items, and thus, never prevents any read-write transaction from making progress.[2]

Unfortunately, even if R/W transactions execute atomically, reading multiple objects without using locks could end up with inconsistent *snapshots*. To illustrate this, consider again the example transactions EX-T1, EX-T2 above. We add one more transaction, EX-T3, which reads A and B. An unconstrained execution might result in EX-T3 seeing A=0 (before any commits) and B=2 (after both EX-T1, EX-T2 commit). We want a read-only transaction to view a state of the data store which could result from an atomic execution of the R/W transactions, and **the only known way to do this without locking for reads is to introduce data versioning, and potentially retain multiple versions of the same object until they can be evicted**.

Data versioning with a global source of ordering works as follows. We assign each version a *timestamp* according to a global transaction commit order and label all writes within a transaction with this timestamp. A client can read a consistent snapshot by requesting to read object versions not exceeding some designated timestamp.

[2]We note that the term might be confused with lock-freedom, which is a progress condition that stipulates execution progress under certain conditions. Therefore, we refer to these simply as read-only transactions.

In the above example, say that EX-T1 commits first, and we assign it timestamp 1; and EX-T2 commits next and obtains timestamp 2. A client performing EX-T3 could request to read snapshot 0, before any commits; snapshot 1, after EX-T1 commits but before EX-T2; or snapshot 2, after both commits. Importantly, data versioning also introduces potential delays. For example, in the above scenario, reading snapshot 2 must wait until EX-T2 completes. **So although read-only transaction do not lock any data themselves, they might block when waiting for R/W transactions to complete**.

4 Choosing Timestamps

We now describe how Spanner assigns timestamps to R/W transactions, as well as how to choose timestamps for read-only transactions, so as to (i) avoid central control, and (ii) minimize read blocking. The following two informal rules give insight on the selection of timestamps for transactions in Spanner.

Rule 1: The timestamp for T is a real time after all the reads have returned and before the transaction releases any locks.

Rule 2: Each participant contributes a lower-bound on the transaction timestamp T: The lower bound at each participant is greater than any timestamp it has written in the past locally. Jointly, these provide the following properties:

- from Rule 1, it follows that if transaction T starts after transaction T ends, then T must have a higher timestamp than T;
- from Rule 2, it follows that if transaction T reads something that transaction T wrote, then T must have a higher timestamp than T (note that this can happen even if T starts before T ends), and
- from Rule 2, it also follow that if transaction T overwrites something that transaction T previously wrote, then T must have a higher timestamp than T.

Additionally, these rules mean that a server never has to block before replying when asked for data with a timestamp that is lower than the bound that the server proposed for any pending transaction (i.e. one that hasnt yet committed).

It is not hard now to construct a distributed protocol which upholds both rules. In fact, it is natural to incorporate this within a two-phase commit protocol at the end of transaction execution, as detailed in Figure 1 below: The first phase collects lower-bounds and determines a timestamp for the transaction, and the commit phase instructs participants to execute pending writes and make them visible, using the corresponding timestamp as their version number.

In order to account for clock skews, Spanner makes use of an approximate clock service called *TrueTime*, which provides a reading of an *interval* surrounding real time (akin to Marzullos time server [3]). The Spanner coordinator reads TrueTime at the beginning of the two-phase commit, and delays sufficiently long before commit to guarantee that the timestamp it chooses for the transaction has passed in real time.

Phase 1 (Prepare)

Non-coordinator: When asked to lock data for writes, each non-coordinator participant picks a *Prepare time* ts_{local}, which serves as lower-bound on the final timestamp the coordinator will assign for this transaction. ts_{local} satisfies (a) it is larger than any timestamp associated with its local data, and (b) it is larger than any lower-bound it sent in a previous Prepare.

Coordinator: The coordinators lower bound, in addition to being monotonically increasing with respect to its past writes, is constrained to be strictly higher than current real time. That is, its lower bound is greater than the time at which all of the transactions reads have completed. (As a practical note, the first action on the Spanner coordinator is the local reading of the clock, so as to minimize the forced wait in Phase 2, due to any clock skew. And the Spanner coordinator defers taking its own write-locks until after it collects responses from all participants, so as to maximize concurrency.)

Each participant (including coordinator) records the Prepare time so as not to serve read requests at a time higher than it until the transaction completes.

Phase 2 (Commit)

The transaction timestamp is computed as the maximum among the lower-bounds which were collected from participants. The coordinator forces a wait until real clock time has passed the transaction timestamp, before commencing to commit all writes and release locks.

Upon Commit, each participant removes the Prepare time associated with the transaction, and removes the restriction on reads which was associated with it.

Figure 3: 2-Phase Timestamp Selection.

5 Choosing Timestamps for Read-Only Transactions

In order for a read inside a read-only transactions to uphold linearizability, it must return the latest value written in a transaction that ends before the read starts. Asking for the upper-bound on TrueTimes current clock reading suffices, because any transaction that ended before that point has a lower timestamp.

However, using this timestamp may cause the transaction to wait. So in the special case of a read-only query addressed to a single server, Spanner instead sends a special read request that tells the server to use the latest timestamp is has written locally to any data. If there is no conflicting transaction in progress, then that read can be answered immediately.

Spanner also supports read-only transactions from arbitrary snapshot times, which are properly named snapshot-reads.

6 Concluding Remarks

As mentioned up front, we have omitted many details and distilled only the transaction concurrency control. In particular, transaction and timestamp management is intertwined in Spanner with the two-phase commit protocol. Therefore, a complete description of the protocol would include details pertaining to logging for recovery and Paxos replication management within each participant. Additionally, we did not specify how locks are reclaimed in case of failures. Spanner clients send *keepalive* message to a transaction leader, and all transaction participants are themselves replicated for fault tolerance.

We have encountered a subtlety in the Spanner paper which was glossed over above. The description of timestamp management in the paper requires that the coordinator contribute a lower bound which is larger than any "timestamp the leader has **assigned**[3] to previous transactions". It may not be completely unambiguous what the term "assigned" refers to, because only a transaction coordinator selects/assigns timestamp. In the description above, we disambiguate that the lower bounds should be greater than the time associated with any data "written" in the past.

An example where this matters may be constructed as follows: First the coordinator reads its local clock to set its own lower bound ts_{local}, then it receive a commit message as non-coordinator for some other transaction T whose timestamp is higher than ts_{local}, then it commits T with timestamp ts_{local}.

Finally, stepping back from details, we can summarize Spanner's concurrency control method as follows. Spanner uses straightforward two-phase commit in order to order read-write transactions. The particular variant of two-phase commit implemented in Spanner (i) delays write-lock acquisition until the end of a transaction, and (ii) uses Wound-Wait to resolve deadlocks.

Most of the complication in the concurrency-control protocols is due to read-only transactions. Supporting these transactions entails (i) maintaining multiple versions of each data item, (ii) choosing timestamps for read-write transactions, and (iii) serving "read at timestamp" requests.

Acknowledgements: We are grateful to Spanner's Wilson Hsieh, and to Idit Keidar, for helpful comments on earlier versions of this paper.

References

[1] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[2] James C. Corbett and other. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, 2012.

[3] Keith Marzullo. *Maintaining the time in a distributed system.* PhD thesis, Stanford University, Department of Electrical Engineering, February 1984.

[4] Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis, II. System level concurrency control for distributed database systems. *ACM Trans. Database Syst.*, 3(2):178–198, June 1978.

[3] Boldface added for emphasis

Fault Tolerant Transaction Architectures

Ittay Eyal
Department of Computer Science
Cornell University
ittay.eyal@cornell.edu

1 Introdution

A data store that spans multiple partitions (also called shards) and implements atomic transactions must coordinate transactions that span multiple partitions. Each partition is represented by an Object Manager (OM); users access the system through Transaction Managers (TMs) that export the transaction API (start-transaction, read/write, end-transaction) and initiate the aforementioned coordination to certify the transaction (decide commit/abort). This simplified structure is illustrated in Figure 1. Coordination becomes a challenge in the face of message loss and machine crashes, faults that are a norm in today's large scale systems. We review here several contemporary architectures, discuss the tradeoffs among them, and compare them through simulation.

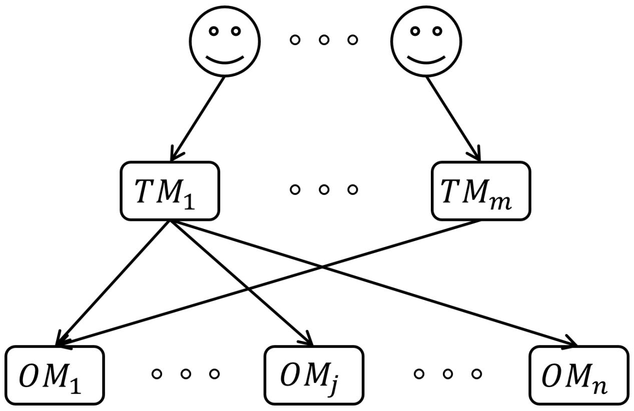


Figure 1: The simplified architecture of an object store with transactions. Each partition is maintained by an OM; TMs provide the API to the clients, initiate transaction certification (not shown in figure) and store data in the OMs.

2 Global Serialization

Perhaps the most direct approach for serializing transactions is using a global highly-available log service that serializes all transactions in the system. When a transaction is complete, the TM submits it to this global log. The result of each transaction, commit or abort, is determined by the order of previous transactions. A transaction commits if and only if it has no conflicts with previous committed transactions in the log. The global log must be highly efficient, since it is in the critical path of all transactions. It also has to be resilient to the expected failures.

Sprint [3] uses an atomic multicast service as the global serializer. Transactions are atomically multicast to the relevant OMs, and each finds its local result according to delivery order – success if there are no conflicts with previous transactions in its partition, or failure otherwise. If there are no collisions in any of the OMs, the transaction commits.

Hyder [2] serializes all transactions in a single global log. A transaction is then committed if it doesn't conflict with transactions that precede it in the global log.

This architecture works in various cases, as demonstrated above. Specifically, it is suitable for read-dominated workloads, as read transactions form no contention on the shared log. Recovery from failures is simple and efficient, requiring a replay of the single log. However, its scalability for updating (i.e., not read-only) transactions is limited by the throughput of the centralized log, which has to achieve consensus on the value of each entry. This means that all transactions in the system contend for their position in the log.

3 Two-phase commit

In order to avoid a potential bottleneck with a single global element in the critical path, many systems opt for using two-phase commit. This is performed by means of two-phase locking. When a transaction is ready to certify, the TM requests locks for its objects from the appropriate OMs. Each OM grants the locks if they are available, and if all locks are successfully acquired, the TM instructs the OMs to commit the transaction and release the locks. If the TM cannot acquire a lock, it can either wait for it to become available (without reaching a deadlock) or release all its locks and abort the transaction. Replicating OMs is therefore insufficient when using 2PC, since if a TM fails while holding a lock for an object, no other transaction would be able to access this object.

One way of overcoming such deadlocks is to replicate the TMs. Megastore and its variants [1, 11] and Spanner [5], which target read-dominated workloads, use this strategy. However, this extends the certification time of transactions that update objects, since in addition to the replication at the OMs, the TMs have to replicate their operations on the locks.

4 Single Reliable Tier

In order to avoid the cost of highly available TMs, one needs a mechanism to overcome the failure of the TM coordinating a transaction.

4.1 Error-Prone TMs with Accurate Failure Detectors

Warp [8] orders all OMs of a transaction in a chain, and each of them, in order, tries to lock its objects. If all lock acquisitions are successful, the chain is traversed backwards, executing the transaction and releasing the locks. On failure (or abort due to conflicts), the chain or its prefix are traversed backwards, releasing the locks without executing the transaction. MDCC [10] uses advanced Paxos techniques to perform low latency commits. When submitting a transaction, the TM sends to all participating OMs the list of all OMs in the transaction. In case of a TM failure, the OMs can replace it with another, which will either complete the transaction or abort it.

Both architectures rely on accurate failure detectors. Accurate failure detectors can be constructed by having a central entity assign roles by leases. With leases a failed node is considered failed only after its lease has expired.

4.2 Error-Prone TMs with Inaccurate Failure Detectors

Without an accurate failure detector, one runs into the risk of false positive failure detection, where a functioning element, a TM, is falsely suspected and replaced. Scalable Deferred Update Replication (S-DUR) [12] uses unreliable TMs, but avoids the use of a central leasing entity by allowing multiple TMs to try to pass contradicting decisions on a transaction. Of course only one decision may be accepted. Each transaction is partitioned and submitted to the OMs that maintain the objects it touches. Each OM serializes the transaction locally, and conflicts are checked for each object separately. If there are no conflicts for any of the objects, the transaction is committed.

If a TM is falsely suspected as failed while trying to pass a transaction, another TM is requested to terminate the transaction in its place. This second TM will submit to the OMs a suggestion to abort the transaction. However, since the original TM has not failed, it sends its own messages to the OMs, suggesting to commit. The messages from the two TMs can reach the OMs in any order, and the one that is serialized first is the one that counts. Subsequent suggestions are simply ignored. The result of the transaction (commit/abort) is then determined unambiguously by the first suggestion reaching each of its OMs.

5 ACID-RAIN: ACID transactions in a Resilient Archive of Independent Nodes

In ACID-RAIN [9] we use a single reliable tier with inaccurate failure detectors. However, we separate the OMs, which serve as an advanced cache, from logs, which serve as the only highly available elements in the system. Each OM is backed by a single log, and the logs are independent of one another.

Once a transaction is ready for certification, a TM submits it to the relevant OMs, and each OM serializes a corresponding transaction entry in its log. The order in each log implies a local success or a local abort decision, depending on conflicts with the previous entries in this log. The decisions in all logs determine the fate of the transaction.

We address the delicate issue of garbage collection (GC), which is in fact a truncation of a log. Since logs are interdependent, with entries of the same transactions, their garbage collection has to be performed in a coordinated fashion. Consider a transaction T with entries in logs A and B, and suppose its entries in both logs have no conflicts with previous transactions. It should be committed. If the entry of T is then GCed from A, but, due to failures or message loss, B is not informed of the commit, a recovery is initiated. However, now we cannot distinguish between this execution and one where the transaction entry has never reached A. Therefore, a TM trying to recover sees T in the OM backed by log B, but it cannot determine whether it should commit or abort.

To overcome this, a TM first submits the transaction entry to all OMs. After they are all logged, it submits the transaction decision to all OMs, and after these are all logged, it submits a GC entry to all OMs. The OMs log these entries and will only truncate the log at a point where all previous transaction entries are followed by a corresponding GC entry. This workflow is demonstrated in Figure 2, starting from the endTxn message sent from the client to the TM. If a transaction is missing its initial entry in some of its logs, the TM may place a *poision* entry, which will cause a local failure if this entry is serialized prior to the transaction entry. This poison would later be GCed like a regular entry.

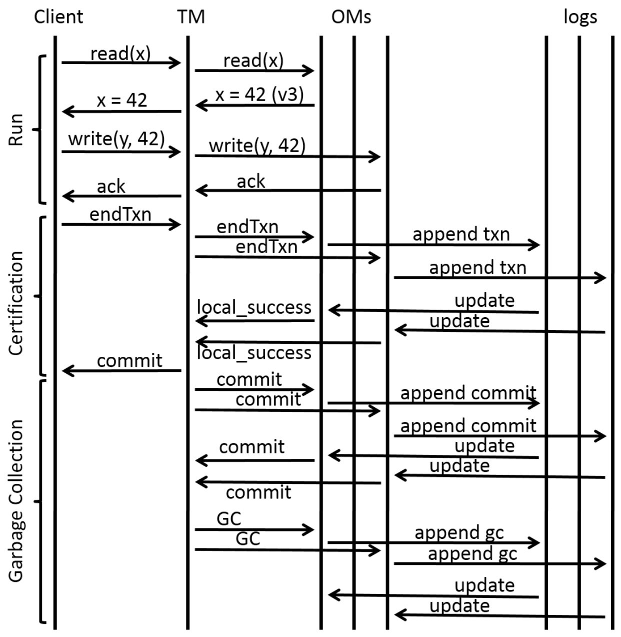


Figure 2: Progress of an ACID-RAIN transaction.

An important benefit of this architecture is that it allows for fast recovery from failures of OMs and TMs. A node (either a client or an OM) that detects a failure of a TM may instantly request any other TM in the system to take over and complete any transaction. Similarly, if an OM is suspected as failed, it is replaced by another. Though false suspicions may lead to duplicate TMs/OMs performing the same role, this does not violate correctness. The logs define the unique serialization point even when two OMs write to the same log. Such fast recovery is important since pending transactions form implicit locks, preventing conflicting transactions from completing until the fate of the former (commit/abort) is decided.

Prediction The mechanism described above allows for efficient and fault tolerant transaction certification. In order to decrease conflict ratio, we use prediction to order transactions before they commit. We assume that when starting a transaction the system can estimate which objects this transaction is likely to access. This can be done using machine learning tools analyzing previous transactions. Then, before starting the transaction, the TM asks all relevant OMs for the latest available version of each of these objects. Each OM returns the earliest version that was not previously reserved for another. The TM takes the maximum of these as its predicted commit timestamp, and asks all relevant OMs to reserve it its objects with this timestamp. When the transaction subsequently starts to perform reads, the OMs only respond once the predicted version is ready or after a timeout. Since certification follows the algorithm explained above, these reservations serve as hint, which may be ignored, and thus cannot form deadlocks.

6 Simulation

We compare the three architectures described above through simulation. We simulate each of the agents in the system with a custom-built event-driven simulator. Our workloads are an adaptation of the transactional YCSB specification [6, 7], based on the original (non-transactional) YCSB workloads [4]. Each transaction has a set of read/update operations spread along its execution. It chooses objects uniformly at random from a set of 500 objects. This small number of objects implies high contention.

The results are shown in Figure 3. We see the maximal commit rate (measured in Transactions per Unit Time) each architecture can accommodate with an increasing number of shards. The Global Log architecture is bounded by the throughput of the global log, scaling with the number of shards until this bound. The 2PC with reliable TMs architecture (SMR TMs curve) suffers from increased TM latency, equal to that of a reliable log. This increases certification time, and therefore increases contention, reducing scalability. ACID-RAIN represents the single reliable tier architecture, demonstrating its superior scalability in this high contention scenario.

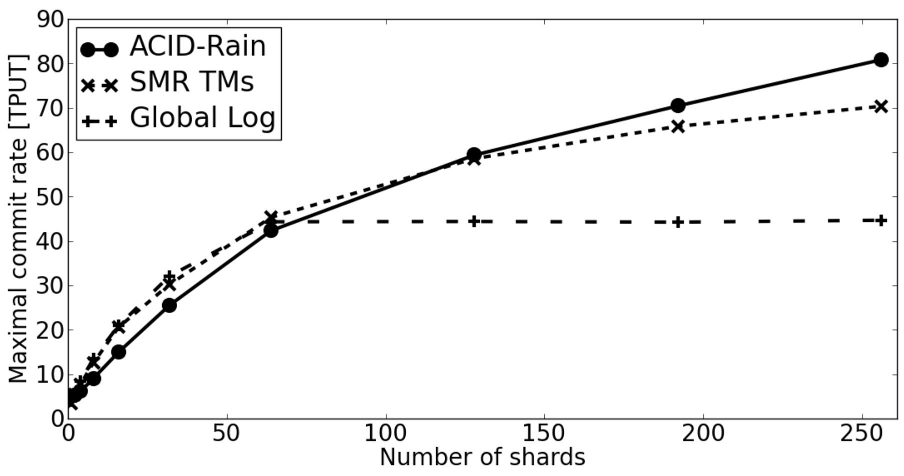


Figure 3: Comparison of the architectures.

7 Conclusion

Architectures for atomic transactions in partitioned object stores rely on multiple independent reliable and unreliable agents. The interplay of these elements in executions with and without failures determines the performance of the system. The single-log architecture allows for simple and efficient failure recovery as long as the global log does not form a bottleneck. In read-dominated workloads, a 2PC variant with highly available TMs allows for good scalability, despite increased latency for transactions that perform updates. If failures are infrequent, it is possible to reduce certification time by using unreliable TMs with an accurate failure detector. Finally, in large scale systems where failures are to be expected, it is possible to reduce failure detection time at the risk of false-positive suspicion. This entails a more elaborate recovery procedure and careful garbage collection.

This underlying infrastructure detects transaction collisions only on their certification. In ACID-RAIN we detect such potential conflicts in advance using prediction, and order the transactions accordingly. This ordering mechanism does not require strict locks that subsequently raise deadlock issues in error prone scenarios. Instead, it serves as a suggestion mechanism, backed by a lock-free certification protocol.

References

[1] J. Baker, C. Bond, J.C. Corbett, JJ Furman, A. Khorlin, J. Larson, J.M. Léon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, 2011.

[2] Philip A Bernstein, Colin W Reid, and Sudipto Das. Hyder-a transactional record manager for shared flash. In *Proc. of CIDR*, 2011.

[3] Lásaro Camargos, Fernando Pedone, and Marcin Wieloch. Sprint: a middleware for high-performance transaction processing. *ACM SIGOPS Operating Systems Review*, 2007.

[4] B.F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*. ACM, 2010.

[5] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Googles globally-distributed database. *OSDI*, 2012.

[6] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration. *VLDB*, 2011.

[7] A.J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *ACM SIGMOD*, 2011.

[8] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Warp: Multi-key transactions for key-value stores. Technical report, 2013.

[9] Ittay Eyal, Ken Birman, Idit Keidar, and Robbert van Renesse. ACID-RAIN: Acid transactions in a resilient archive with independent nodes. Technical Report CCIT 827, Technion, Israel Institute of Technology, March 2013.

[10] Tim Kraska, Gene Pang, Michael J. Franklin, and Samuel Madden. MDCC: Multi-data center consistency. *CoRR*, 2012.

[11] S. Patterson, A.J. Elmore, F. Nawab, D. Agrawal, and A. El Abbadi. Serializability, not serial: Concurrency control and availability in multi-datacenter datastores. *VLDB*, 2012.

[12] Daniele Sciascia, Fernando Pedone, and Flavio Junqueira. Scalable deferred update replication. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2012.