Ordering Transactions with Prediction in Distributed Object Stores

Ittay Eyal[1], Ken Birman[1], Idit Keidar[2], and Robbert van-Renesse[1]

[1]Department of Computer Science, Cornell Univerisity, Ithaca, NY, USA
[2]Department of Electrical Engineering, Technion, Haifa, Israel

Abstract

In cloud-scale datacenters, it is common to shard (partition) data across large numbers of nodes. Atomic transactions are typically implemented by running transactions speculatively, and then certifying them, aborting ones that cause conflicts. However, in high contention scenarios, this approach has drawbacks: rather than achieving any substantial level of concurrency, it prevents concurrency by aborting all but one of the contending transactions.

Our work explores a new option. We employ prediction, ordering transactions in advance based on the objects they are likely to access, providing ACID transactions in a Resilient Archive with Independent Nodes (ACID-RAIN). This preliminary ordering decreases abort rate, and eliminates aborts in error-free executions. To allow fast recovery from failures our scheme does not introduce any locks. The system consistency and durability rely on a single scalable tier of highly-available independent logs. Simulations using the Transactional-YCSB workloads show the scalability and benefits of ACID-RAIN.

1 Introduction

Large-scale data-center computing systems often maintain massive data sets, *sharded* over large numbers of storage nodes. When client transactions access data on multiple shards, the issue of consistency arises. Ideally, we would use a system with ACID transactions [2, 10, 1], because this model facilitates reasoning about system properties and makes possible a variety of high-assurance guarantees. Nonetheless, the ACID model is widely avoided in today's large-scale systems due to efficiency concerns [9]. Existing approaches typically run transactions speculatively and perform certification after they complete to preserve consistency, either committing or aborting each transaction depending for conflicts.

In this paper, we present ACID-RAIN — an architecture for ACID transactions in a Resilient Archive with Independent Nodes. The system orders transactions before they begin by employing *predictors* that estimate the set of objects each transaction will access. Such predictors can be implemented with machine learning tools [16]. To leverage prediction, a transaction *reserves* a version of each object it will use. When later accessing the objects, it will only see these reserved versions.

To run effectively at large scale, ACID-RAIN must tolerate performance hiccups, message loss, and crashes, all of which are common in such settings. Ideally, progress should never depend on the responsiveness of any single machine. Accordingly, ACID-RAIN requires reliable entities only at a single tier of the system — a set of independent highly available logs, used in a novel manner. All other entities may fail and can be replaced instantly on failure; the architecture maintains consistency even in the event of false suspicion. Additionally, reservations serve as suggestions, rather than strict guarantees of the kind that locking would provide, and a reservation that is not used because of a sluggish or dead owner is ignored.

The independence of all system elements allows for good scalability. Nevertheless, due to the interdependence of the log contents, garbage collection (GC) has to be carefully coordinated to maintain consistency.

We evaluate our architecture by simulation with the transactional-YCSB benchmark [7, 8] as our workload. We contrast the effectiveness of employing prediction and the scalability of ACID-RAIN with other approaches.

2 Model and Goal

We assume unreliable servers that may crash or hang, in an asynchronous, loss-prone network. To accommodate reliable storage, we employ highly-available, sequentially consistent logs, as explained in Section 3.1.

The system exposes a transactional data store supporting serializable transactions. A client invokes a

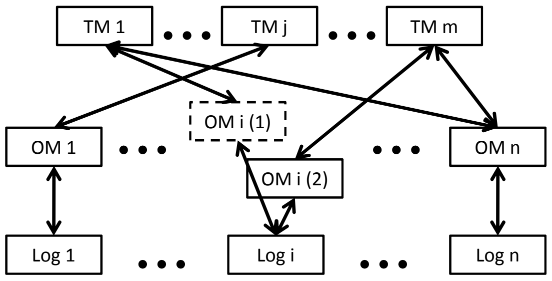


Figure 1: Schematic structure of ACID-RAIN. TMs access multiple objects per transaction. Objects are managed by OMs. $OM_{i(1)}$ is falsely suspected to have failed, and replaced by $OM_{i(2)}$, causing them to concurrently serve the same objects. OMs are backed by highly-available logs, where they store tentative transaction entries for serialization, and (later) certification results.

begin-transaction command, followed by read (e.g., a field from a table) and update (e.g., setting the value of a field in a table) operations. Finally the client invokes the end-transaction command, and the system responds with either a commit or an abort. Committed transactions form a serializable execution. TMs are equipped with predictors that foresee which objects a transaction is likely to access on its initiation.

3 ACID-RAIN

We now describe the operation of ACID-RAIN. We start with an overview of the system's structure in Section 3.1, and proceed to describe the algorithm in Section 3.2.

3.1 System Structure

The structure of the system is illustrated in Figure 1. At the base of ACID-RAIN are a set of independent highly-available logs that together describe the state of the entire system. Each log is accessed through an *Object Manager*[1] (*OM*) that caches the data and provides the data structure abstraction — exporting read and write operations, while supporting transactions, which are managed by *Transaction Managers* (*TMs*).

TMs provide the atomic transaction abstraction. They receive instructions from clients to start and end a transaction, and operations to perform on individual objects within the transaction. On transaction start, the TMs predict which objects it is likely to access, and reserve these object versions. Then, they speculatively perform each operation with the help of the appropriate OMs and according to the order set by the reservations. Finally, they certify the transaction by checking for conflicts in each log (via its OM).

Membership monitors are in charge of deciding and publishing which machines perform which roles, namely which machines run the log and OM for each shard, and which TMs are available. Any client can access any TM for any given transaction. Other than the logs, server role assignment may be inconsistent. Each object (transaction) is supposed to be managed by a single OM (TM, resp.) at a given time, but this may change due to an unjustified crash suspicion whereupon an object (transaction, resp.) may temporarily be managed by two OMs (TMs, resp.) that do not know of one another.

Log Specification ACID-RAIN uses log servers for reliable storage. Each log server provides a sequentially consistent log object, i.e., update operations are linearizable, but reads may return outdated results[2]. Multiple machines may append entries to a log. Machines may register to the log; the log then sends to each all entries, from the first one in the log, to its end, and then new entries as they arrive. An OM may instruct the log to truncate its prefix.

3.2 Algorithm

We now describe the ACID-RAIN algorithm. We explain the reservation and certification protocol (illustrated in Figure 2), and then discuss prediction errors, garbage collection, and failure handling.

A transaction begins with the TM receiving a begin-transaction instruction from the client. The TM assigns it a unique *txnID*, and predicts which objects the transaction will access. It interrogates the OMs about all these objects, and they respond with the latest unreserved timestamp of each object. The TM chooses a timestamp larger than maximum among the responses, and asks the OMs to reserve the objects with this timestamp to *txnID*. The OMs confirm the reservation if no concurrent TM has reserved a larger timestamp in the meantime. The TM then proceeds to serve transaction operations by routing them to the appropriate OMs. Each operation is sent to the OM in charge of the object, along with the *txnID*. The OMs order accesses based on timestamp reservations, and respond only when the correct version is available.

Each committed transaction is assigned a timestamp. When reading an object, the timestamp of the

[1]In an implementation of the system one may use multiple OMs per log, dividing the log's object set, or the other way around, have multiple logs report to a single OM. The choice depends on the throughput of the specific implementations chosen for each service. In this paper we use a 1:1 mapping for simplicity of presentation.

[2]Such logs may be implemented with various techniques, from SMR to log chains [13, 14]; we abstract this away, and assume highly available logs.

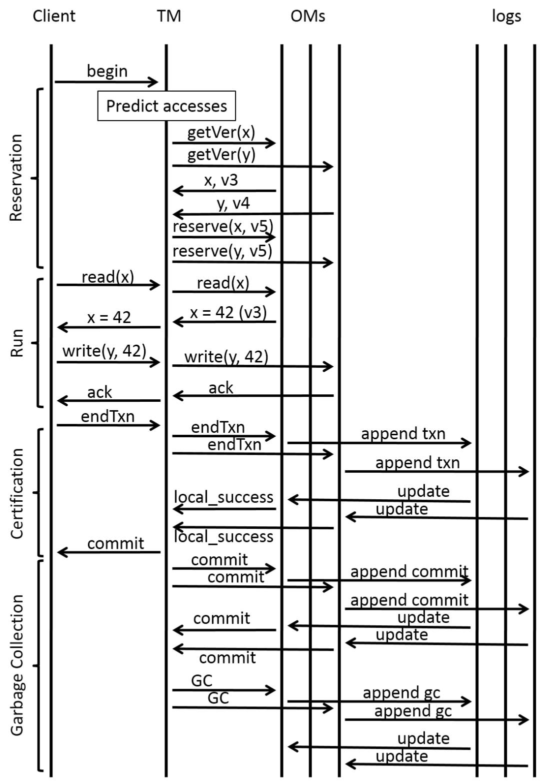


Figure 2: An example flow of the algorithm.

latest transaction that wrote this object is returned to the TM. The transaction's timestamp is chosen to be larger than the largest timestamp returned by its operations, and not larger than its reserved timestamp (if possible).

Once a TM receives an end-transaction instruction from a client, it notifies the transaction's OMs, detailing the transaction's timestamp and *log-set* (the logs in charge of the shards it touched). When it receives such a notification, an OM appends to its log an entry consisting of the *txnID*, its timestamp, its read- and write-sets (read-set with the read timestamps, write-set with written values), and its log-set. It then waits for the entry to appear in the log.

A transaction is committed if and only if it is written to all logs, and it does not conflict with previous transactions on any of them. Conflicts are violations of read-write, write-read or write-write order. Each OM checks for local conflicts by checking timestamps in the prefix of the log up to the transaction entry, and sends its local result (success/failure) to the calling TM. If all return success, then the transaction has committed, otherwise it has aborted. The TM notifies the client of the transaction result and instructs the OMs to place this result in the logs. The OMs notify the TM once the results are logged.

Prediction Errors If there are no prediction errors, failures, or false suspicions, there are no aborts.

If the transaction accesses an object that was not predicted, this object has no reserved version for it. Accessing it can therefore result in a conflict of the transaction, or of the following ones. This conflict would be detected at certification time, and result in an abort of a transaction. Despite the harm to performance, it would not break consistency.

If a transaction does not access an object that was predicted, the TM must still release the reservation when the transaction ends. This reservation might slow the processing of other transactions that wait for its release, but would not break consistency.

If a TM is suspected as failed, its reservations are revoked. This too may harm performance, but cannot break consistency.

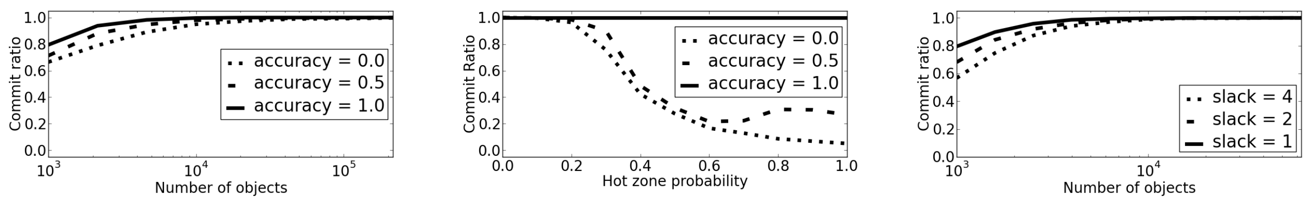
GC Logs are truncated to conserve resources and to reduce log replay time on OM recovery. Each OM occasionally summarizes the log prefix, and places this summary in the log. However, this snapshot is insufficient, since truncation must not break transaction certification. Each transaction should be either committed or aborted in all its logs, and therefore cannot be removed from any of them before the result is published. To verify this, the committing TM appends a GC entry to all the transaction's logs after receiving an acknowledgement that they all registered the transaction's result. An OM can invoke log prefix truncation if the prefix was summarized, and all its transactions have corresponding GC entries.

Robustness In case of a TM or OM crash, or a missing result or GC entry (due to message loss), another TM may read the transaction entry in one of the logs, find its log-set, and continue the certification and GC process.

If a TM places a transaction entry in a strict subset of the transaction's log set, when another TM is instructed to fix this, it cannot tell whether the original TM is crashed or slow. To overcome this, we introduce *poison* entries. The fixing TM places a poison entry in the logs that miss the original entry. A poison is interpreted as a transaction entry with a conflict. The original entry may either arrive eventually or not. The first entry/poison counts, and the following are ignored. Any TM can therefore observe the log and consistently determine the state of the transaction, without a race hazard.

4 Evaluation

We evaluate ACID-RAIN by comparing its performance to the classical approach that does not use



(a) Uniform random

(b) Hot zone

(c) Slack

Figure 3: Ordering transactions in advance reduces conflicts and increases commit ratio. High conflict rates occur without with uniform access to a small number of objects (a), and high probability of accessing a hot-zone (b). Even inaccurate prediction is significant in high contention, compared to the the classical approach (accuracy=0). Commit ratio is affected if the predictor reserves unnecessary objects by a factor of *slack* (c).

prediction and compare its certification protocol with other certification schemes. We use a custom-built event-driven simulation, simulating each of the agents in the system — clients, TMs, OMs and logs. Our workloads are an adaptation of the transactional YCSB specification [7, 8], based on the original (non-transactional) YCSB workloads [5]. Each transaction has a set of read/update operations spread along its execution. Object accesses follow one of two different random distributions — (1) uniform, where each object is chosen uniformly at random, and (2) hot-zone, where some of the objects belong to a so called hot-zone, and each access is either to the hot-zone, or outside of it (chosen uniformly within each zone). For every run, we set an average transaction per unit-time rate (TPUT), and transactions arrivals are governed by a Poisson process with the required TPUT.

Prediction Our first test scenario imposes a load substantially below the system's capacity with 16 shards. Each transaction reads and writes 10 objects. The simulation is faithful to the algorithm, with the exception of a small shortcut – OMs grant reservations by arrival time rather than by timestamp. This results in deadlocks in high contention scenarios, and these are resolved with timeouts.

First we vary prediction accuracy, i.e., the average ratio of objects the predictor guesses out of the set the transaction eventually accesses. An accuracy of 0 is equivalent to no prediction and no reservation (the classical approach), and an accuracy of 1.0 means predicting all accesses.

We consider (1) uniform random load (Figure 3a), increasing contention by decreasing the number of objects, and (2) load with a hot-zone of 1000 out of 10^7 (Figure 3b), increasing contention by increasing the hot-zone access probability. Without prediction, commit rate drops as contention rises. Accurate prediction reduces or even eliminates this drop.[3] In highest contention scenarios, even with moderate prediction accuracy, we obtain significant improvement over the classical approach (prediction=0).

We define *slack* to be the average ratio between the number of accesses predicted and the number of objects accessed by the transaction. If a transaction accesses 10 object, then with a slack of 1.5, it would reserve another 5 random objects. In Figure 3c we compare (now with uniform random load and a variable number of objects) the effect of using a perfect predictor (slack=1) with predictors that overpredict by factors of 2 and 4. The impact of overprediction is surprisingly minor, a finding that should make it easier to create a practical predictor.

Certification scalability To evaluate the scalability of ACID-RAIN's certification mechanism, we avoid prediction and measure the maximal commit rate it can accommodate with an increasing number of shards. Each transaction performs 3 reads and 3 writes of objects chosen uniformly at random from a small set of 500 objects. We compare (Figure 4) ACID-RAIN against two approaches (more details in Section 5): *SMR TMs* is two-phase commit with reliable coordinators (TMs). *Global log* is an architecture where TMs submit all transactions to a single global log and check conflicts on that single log.

ACID-RAIN scales better than 2PC since its faster certification reduces contention. It has no bottleneck as with a global log (that has less overhead in small scale). While the parameters we choose are arbitrary, the trends are apparent; choosing other parameters would provide similar trends.

[3]Note that when all accesses are to the hot zone (Figure 3b at 1.0), commit rates are lower with imperfect prediction than in the uniform random case with 1000 objects (Figure 3a at 10^3). This is because all accesses to the hot-zone go through a single OM that becomes a bottleneck. On the bright side, since object access conflicts occur only at a single shard, the reservations prevent deadlocks and result in perfect commit ratio with perfect prediction.

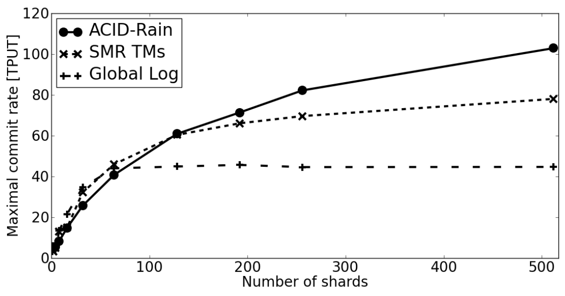


Figure 4: For an increasing number of shards, we run multiple simulations to find the maximal TPUT the system can handle. A global log forms a bottleneck, and 2PC with SMR TMs is blocked by contention much earlier than ACID-RAIN due to its longer certification time.

5 Related Work

Our transaction ordering protocol is based on a state-machine ordering mechanism suggested by Lamport [12], but we have generalized the protocol to work with arbitrary overlapping participant sets. We are unaware of work that uses prediction to order distributed transactions before certification.

We briefly review here work related to ACID-RAIN's certification protocol. One approach for certification is to use a single highly-available service that orders all transactions in the system, e.g. [4, 3]. A transaction commits if and only if it has no conflicts with previous committed transactions. When update (not read-only) transaction rate is high, such a global service becomes a bottleneck. In contrast, our system has no such bottleneck.

Many systems [2, 15, 10, 6] use two-phase commit for transaction certification. The downside of these approaches compared to ACID-RAIN is that they require a coordinator that performs transactions to be highly available. This requires another consensus (in addition to the one at the shard itself) for each transaction, increasing certification time, and therefore contention.

The approaches of MDCC [11] and S-DUR [17] are close to ACID-RAIN's certification mechanism. However, ACID-RAIN separates the OM abstraction from the highly-available log layer, facilitating its soft-leasing mechanism and fast recovery. We also address garbage collection, which cannot be done independently at the logs.

Sinfonia [1] uses an architecture similar to our certification mechanism, but addresses minitransactions that are submitted as a whole, with no attempt to order potentially conflicting transactions. We address full transactions, where the clients sequentially access objects before ending a transaction, and use prediction to order them in advance. We believe our techniques could be used to reduce abort rates of systems using Sinfonia or a similar certification mechanism.

6 Conclusion

Prediction of transaction behavior has potential to significantly decrease abort rates in large scale transactional systems with high contention. In addition, performance should never depend on a single machine that can suffer failure or a performance hiccup. In ACID-RAIN we employ prediction to obtain soft reservations and implement atomic transactions while requiring high availability only in a single tier of independent logs. This allows for low latency high throughput certification with fast recovery from failures and performance hiccups.

References

[1] Aguilera, M., Merchant, A., Shah, M., Veitch, A., and Karamanolis, C. Sinfonia: a new paradigm for building scalable distributed systems. In *ACM SIGOPS Operating Systems Review* (2007).

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

[3] Bernstein, P. A., Reid, C. W., and Das, S. Hyder-a transactional record manager for shared flash. In *CIDR* (2011).

[4] Camargos, L., Pedone, F., and Wieloch, M. Sprint: a middleware for high-performance transaction processing. *ACM SIGOPS Operating Systems Review* (2007).

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

[6] Corbett, J., et al. Spanner: Googles globally-distributed database. *OSDI* (2012).

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

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

[9] Gray, J., Helland, P., O'Neil, P., and Shasha, D. The dangers of replication and a solution. In *ACM SIGMOD Record* (1996).

[10] Kallman, R., et al. H-store: a high-performance, distributed main memory transaction processing system. *VLDB* (2008).

[11] Kraska, T., Pang, G., Franklin, M. J., and Madden, S. MDCC: Multi-data center consistency. *CoRR* (2012).

[12] Lamport, L. Using time instead of timeout for fault-tolerant distributed systems. *TOPLAS* (1984).

[13] Lamport, L. The part-time parliament. *TOCS* (1998).

[14] Malkhi, D., Balakrishnan, M., Davis, J., Prabhakaran, V., and Wobber, T. From Paxos to CORFU: a flash-speed shared log. *SIGOPS OS Review* (2012).

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

[16] Pavlo, A., Jones, E., and Zdonik, S. On predictive modeling for optimizing transaction execution in parallel oltp systems. *VLDB* (2011).

[17] Sciascia, D., Pedone, F., and Junqueira, F. Scalable deferred update replication. In *DSN* (2012).