IRWIN AND JOAN JACOBS
CENTER FOR COMMUNICATION AND INFORMATION TECHNOLOGIES

ACID-RAIN: ACID Transactions in a Resilient Archive with Independent Nodes

Ittay Eyal, Ken Birman, Idit Keidar and Robbert-van-Renesse

CCIT Report #827
March 2013

DEPARTMENT OF ELECTRICAL ENGINEERING
TECHNION - ISRAEL INSTITUTE OF TECHNOLOGY, HAIFA 32000, ISRAEL

CCIT Report #827 March 2013

ACID-RAIN: ACID Transactions in a Resilient Archive with Independent Nodes

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

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

Abstract

In cloud-scale data centers, it is common to shard data across many nodes, each maintaining a small subset of the data. Although ACID transactions are desirable, architects often avoid them due to performance concerns. We present a novel architecture for support of low-latency high-throughput ACID transactions in a Resilient Archive with Independent Nodes (ACID-RAIN). ACID-RAIN uses logs in a novel way, limiting reliability to a single scalable tier. A large set of independent fault-prone nodes form an outer layer that caches the sharded data, backed by a set of independent highly available log services. ACID-RAIN dramatically reduces concurrency conflicts by using prediction to order transactions before they take actions that would lead to an abort. Simulations using the Transactional-YCSB workloads demonstrate scalability and effective contention handling.

1 Introduction

Large-scale data-center computing systems often employ massive data sets, *sharded* (partitioned) over many storage nodes. When client transactions access shared data items, the issue of consistency arises. Ideally, we would use a system with ACID transactions [2, 18, 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 due to efficiency concerns [15].

In case ACID transactions are needed, one of two approach is commonly used. The first is to use a central highly available certification entity (e.g., [16, 23]) for serializing transactions. However, such a central certification entity has limited throughput, and therefore, this approach cannot scale beyond a certain point.

Another option is to use a combination of locking or optimistic concurrency control (OCC) with timestamped version management in each shard, together with two phase commit (2PC) across shards. However, 2PC is generally avoided in high-availability systems due to performance and fault-tolerance concerns. In case of failures, specifically, of the 2PC coordinator, which is not a rare event in a large-scale system, all potentially conflicting transactions must block until the failure is mended. To avoid that, existing systems [6, 9] replace the 2PC coordinator with a highly available one. This severely harms throughput, as we demonstrate in Section 5.

In this paper, we present ACID-RAIN — an architecture for ACID transactions in a Resilient Archive with Independent Nodes. It is depicted in Figure 1. Our approach uses logs in a novel manner. A set of independent highly-available logs collaboratively describe the state of the entire system, i.e., one would need to combine all logs in order to learn the global state. Each log is accessed through an *Object Manager* (*OM*) that caches the data and provides the data structure abstraction. *Transaction Managers* (*TM*s) provide the atomic transaction abstraction, certifying a given transaction by checking for conflicts in each log via its OM. The benefit of our approach is that other than the logs, no system entities are required to be highly-available. OMs and TMs that are suspected to have failed can be instantly replaced; safety is not violated by multiple copies running concurrently.

Our system uses a form of OCC: OMs respond to concurrent TM instructions with no locks. To improve latency, the OMs serve requests from speculative local data structures, referring to the logs only for certification. To decrease abort rate, we use *predictors* that foresee the likely access pattern of transactions; such predictors can be implemented with machine learning tools [25]. To leverage prediction, a transaction *leases* a version of an object for its use. Note that unlike locks, failure to respect a lease does not violate safety, and therefore does not delay OM restoration on failures.

We evaluate ACID-RAIN through simulation with the transactional YCSB benchmark [13, 14]. We demonstrate the algorithm's linear scalability, compared to the other approaches mentioned above, and the effectiveness of using accurate and inaccurate predictors.

2 Related Work

The holy grail of low-latency high-throughput ACID transactions has long fascinated the data management community. We detail below the most relevant work with respect to this paper.

One approach is to realize weaker consistency guarantees that enable better performance, e.g., [27, 22, 26]. It is also possible to avoid transactions altogether, an approach sometimes referred to as NoSQL [7, 28]. However, our target is full-fledged ACID transactions.

Several systems [11, 6, 4, 28] offer atomic transactions in single shards; others [12, 10] offer mechanisms to collocate objects such that transactions always access collocated objects. However, in real-world scenarios (e.g., social network), constant object migration is going to be inefficient.

Several contemporary systems, such as Megastore and its variants [2, 24], H-store [18], and Spanner [9] use two-phase commit for cross-server transactions. Sinfonia [1] uses an architecture similar in many ways to ours, but employs locking to provide atomic transactions, and does not take advantage of prediction as ACID-RAIN does. The downside of these approaches compared to ACID-RAIN is that they require a coordinator that performs transactions on multiple objects to be highly available. This requires consensus for each operation, resulting in high latency. High latency reduces throughput, since conflicting transactions block one another, as we demonstrate in Section 5.

Sprint [5] and Hyder [3] order transactions by a global service (a multicast service, and a log, resp.). 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 both cases, the global service used is highly efficient, and sufficient for the target application. However, at a high enough scale, a global service becomes a bottleneck. In contrast, our system has no such bottleneck and achieves unbounded linear scale-out.

The approach of MDCC [19] is close to ACID-RAIN. However, unlike ACID-RAIN, MDCC requires storage nodes to keep the metadata of all transactions ever executed. If the failure detector suspects a transaction to be partially written due to a failure, it initiates a re-execution. To prevent transaction double execution due to a false suspicion, storage servers need to check this history on every vote.

3 Model and Goal

Our system is designed to run in a single data center. 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 4.1.

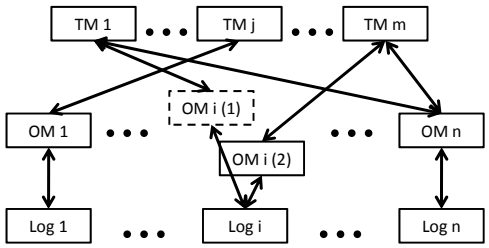


Figure 1: Schematic structure of ACID-RAIN. TMs access multiple objects per transaction. Objects are managed by OMs. $OM_{i(1)}$ is falsely suspected to have failed, and replaced by $OM_{i(2)}$, causing them to concurrently serve the same objects. OMs are backed by highly-available logs, where they store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

The system exposes a transactional data store supporting serializable transactions. A client invokes a begin-transaction command, followed by the transaction's operations. Each operation is either a read (e.g., a field from a table) or an update (e.g., setting the value of a field in a table or adding a key to a key-value store – KVS). Finally the client invokes the end-transaction command, and the system responds with either a commit or an abort. Servers are equipped with predictors that predict which objects a transaction is likely to touch during its run.

4 ACID-RAIN

4.1 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 in the KVS case, while supporting transactions, which are managed by *Transaction Managers* (*TM*s).

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. They speculatively perform each operation with the help of the appropriate OMs, and certify the transaction by checking for conflicts in each log (via its OM). We thus separate the consistency of individual objects, maintained with logs, from that of full transactions, achieved by our algorithm.

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

[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.

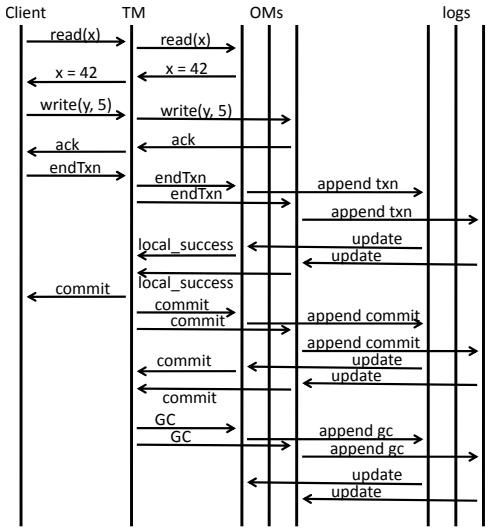


Figure 2: An example flow of the algorithm.

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 of data. Each log server provides a sequentially consistent log object, i.e., update operations are linearizable, but reads may return outdated results. 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. A machine may instruct the log to truncate its prefix.

Such logs may be implemented with various techniques, from SMR to log chains [21, 17, 16, 29, 23]; we abstract this away, and assume highly available logs.

4.2 Algorithm

We now describe the ACID-RAIN algorithm. An illustration of the algorithm's progress is given in Figure 2.

When receiving a begin-transaction from a client, the TM assigns the transaction a unique identifier *txnID* and awaits the transaction's operations. It then services the operations by routing them to the appropriate OMs. Each operation is sent to the OM in charge of the object, along with *txnID*. The response is delivered back to the client.

Each committed transaction is assigned a timestamp. When reading an object, the timestamp of the latest transaction that wrote this object is returned to the TM. The TM calculates the transaction's timestamp by incrementing the largest timestamp returned to it in any of the transaction's operations. Once a transaction is done, the TM also forms its *log-set*, the set of logs in charge of the shards it touched.

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

If the transaction was written to all logs, and it does not conflict with previous transactions on any of them, it is construed as having committed. Conflicts are violations of read-write, write-read or write-write order, including circular dependencies, and can be checked by comparing timestamps. Each OM can only detect local conflicts by checking the prefix of the log up to the transaction entry. The result of the transaction can only be certified by combining the information from multiple logs. A transaction that reads an object updated by a concurrent transaction, cannot be certified until the latter was certified.

Each OM sends its local result to the calling TM. If none of them has conflicts, 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 written to the log.

Running our algorithm for an extended period could lead to lengthy logs. This has two drawbacks. First, an OM that registers with the log has to replay this long log. Second, storage bounds prohibit infinite-length logs. Therefore we wish to occasionally truncate the log. To do that, each OM occasionally summarizes the log prefix, and places this summary in the log. However, this summarization is not sufficient, 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 (Garbage Collect) 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 entry (due to message loss), resulting in a partially certified transaction, another TM may read the transaction entry in one of the logs, find its log-set, and restart/continue the certification and GC process. It is okay to have multiple result entries per transaction – they will be the same. However, each transaction entry is written once, with no retries on failure, to avoid duplicates.

A possible problem may arise 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 failed or is late, and the missing

entries would eventually arrive. Garbage collection may lead to inconsistencies, where some clients see a transaction and others do not, violating atomicity. 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. An entry that is preceded by a poison is ignored, and a poison preceded by an entry is ignored. Any TM can therefore observe the log and consistently determine the state of the transaction, without a race hazard.

Prediction In order to facilitate rapid progress when there are no failures (the common case), the OM maintains two copies of the data structure — a validated copy, and a speculative copy that applies all updates once they happen. An OM responds to the transactions' operations (reads and writes) from its speculative copy, acting as a non-consistent cache, and applies committed changes to the validated copy. Note that when using this mechanism, clients may observe inconsistent data during the course of a transaction, however only transactions with consistent views can commit.

ACID-RAIN leverages predictable transactions by employing leases at the OM layer. Note that these leases are advisory: failure to respect them harms efficiency (aborts can be triggered), but not safety. This means they can be ignored, and therefore do not create a risk of deadlocks.

When a transaction starts, a black-box machine learning mechanism predicts its read and write sets. Given these access predictions, the TM runs a simple two-phase protocol with the OMs to lease (reserve) a set of object versions valid at some instant in logical time, using an ordering mechanism introduced by Lamport [20]: When starting a transaction, the TM interrogates the OMs about all objects it is predicted to access, they respond with the latest timestamp of each object, and the TM chooses a timestamp larger than maximum among the responses. It asks the OMs to reserve the objects with this timestamp, and the OMs grant the lease if there is no lease with a larger timestamp. The TM then proceeds to run the transaction, and the OMs order accesses based on lease timestamps.

This scheme prohibits a lease loop deadlock, and in case of a timeout, transactions can proceed without violating safety, which is guaranteed by the certification procedure.

5 Evaluation

We use a custom-built event-driven simulation to evaluate the architecture of ACID-RAIN. We simulate each of the agents in the system — clients, transaction managers, object managers and highly available logs. 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.

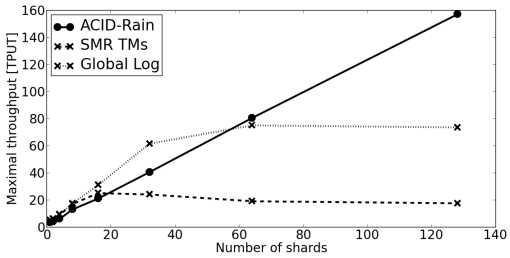


Figure 3: For an increasing number of shards, we run multiple simulations to find the maximal TPUT the system can handle. We observe linear scaling for ACID-RAIN, whereas 2PC and global log reach a bound.

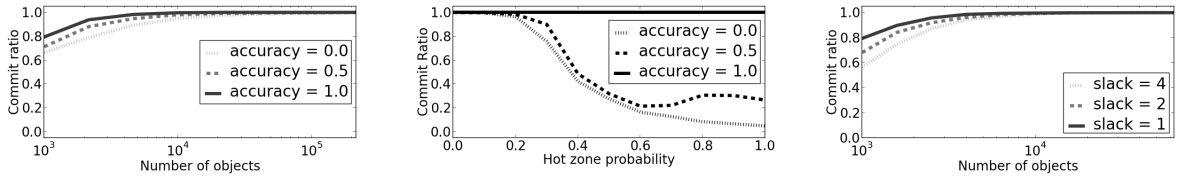
Our workloads are an adaptation of the transactional YCSB specification [13, 14], based on the original (non-transactional) YCSB workloads [8]. Each transaction has a set of read/update operations spread along its execution. Object accesses follow one of two different random distributions — (1) uniform, 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).

Scalability To evaluate the scalability of ACID-RAIN, we measure the maximal TPUT it can accommodate with an increasing number of shards (with 3 reads and 3 writes per transaction of 10^5 objects with uniform access). The result, depicted in Figure 3 demonstrates a linear scaling. This is expected, as the conflict rate in is negligible, and our system is scaled without forming any bottlenecks.

We compare ACID-RAIN with the approaches of (1) using 2PC and highly available independent TMs, implemented as replicated state machines and (2) a global log. In both cases, we allow the systems to skip garbage collection (while ACID-RAIN does perform it). We simulate highly available TMs by increasing the TM's latency to that of a single point to point message (reality would require at least RTT for Paxos or an equivalent, i.e., even worse). To simulate a global log, we bound the TMs' total throughput to about 400 operations per unit time. All other parameters are identical.

While the parameters we choose are arbitrary, the trends are apparent; choosing other parameters would provide similar results, though perhaps at different scales. Improving the efficiency of the highly available TM or the global log would allow them to handle more load than in this example, but they would both reach a bottleneck, at some point.

Prediction We demonstrate the benefits of prediction of different qualities. In all runs we use an incoming workload well below the system's capacity with 16 shards. Each transaction reads and writes 10 objects. The



(a) Uniform random (b) Hot zone (c) Slack

Figure 4: With poor prediction quality, high conflict rates, and hence high abort rates, occur with (a) uniform access to a small number of objects, and (b) high probability of accessing a hot-zone. These problems vanish if predictions have good coverage of object-access sets. Commit ratio is affected if the predictor leases too many objects (c).

simulation is faithful to the algorithm, with the exception of a small shortcut – an OM grants leases by arrival time rather than by timestamp. This results in deadlocks in high contention scenarios, and these are resolved with timeouts. Our full leasing algorithm would eliminate these deadlocks; here, we simply abort when they occur.

First, we consider uniform random load (Figure 4a). We see how commit rate drops as the number of objects decreases. We vary the accuracy, i.e., the ratio of objects the predictor leases in advance. Better prediction means a better commit ratio, however even with perfect prediction deadlocks appear with small numbers of objects.

With a hot-zone of 1000 (Figure 4b), increasing the probability of hot-zone access increases the abort rate. Note that at probability 1.0 the rates are significantly smaller than in the uniform random case (with 1000 objects), since all accesses to the hot-zone go through a single OM that becomes a bottleneck. On the bright side, since object access conflicts occur mostly in a single shard, the leases prevent deadlocks and result in perfect commit ratio with perfect prediction.

We define *slack* to be the ratio of the size of the predicted object access set to the actual object access set. In Figure 4c 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.

Conclusion

Encouraged by these promising simulation results, we plan to provide a full implementation of ACID-RAIN, and publish a rigorous correctness proof.

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 *Proc. of CIDR* (2011).

[4] CALDER, B., WANG, J., OGUS, A., NILAKANTAN, N., SKJOLSVOLD, A., MCKELVIE, S., XU, Y., SRIVASTAV, S., WU, J., SIMITCI, H., ET AL. Windows azure storage: a highly available cloud storage service with strong consistency. In *SOSP* (2011).

[5] CAMARGOS, L., PEDONE, F., AND WIELOCH, M. Sprint: a middleware for high-performance transaction processing. *ACM SIGOPS Operating Systems Review* (2007).

[6] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W., WALLACH, D., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. Bigtable: A distributed storage system for structured data. *TOCS* (2008).

[7] COOPER, B., RAMAKRISHNAN, R., SRIVASTAVA, U., SILBERSTEIN, A., BOHANNON, P., JACOBSEN, H., PUZ, N., WEAVER, D., AND YERNENI, R. PNUTS: Yahoo!'s hosted data serving platform. *VLDB* (2008).

[8] COOPER, B., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *SoCC* (2010), ACM.

[9] CORBETT, J., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., ET AL. Spanner: Googles globally-distributed database. *OSDI* (2012).

[10] CURINO, C., JONES, E., ZHANG, Y., AND MADDEN, S. Schism: a workload-driven approach to database replication and partitioning. *VLDB* (2010).

[11] DAS, S., AGRAWAL, D., AND EL ABBADI, A. Elastras: An elastic transactional data store in the cloud. *HotCloud 2* (2009).

[12] DAS, S., AGRAWAL, D., AND EL ABBADI, A. G-store: a scalable data store for transactional multi key access in the cloud. *SOCC* (2010).

[13] 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).

[14] 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).

[15] GRAY, J., HELLAND, P., O'NEIL, P., AND SHASHA, D. The dangers of replication and a solution. In *ACM SIGMOD Record* (1996).

[16] HUNT, P., KONAR, M., JUNQUEIRA, F., AND REED, B. Zookeeper: Wait-free coordination for internet-scale systems. In *ATC* (2010).

[17] JUNQUEIRA, F., REED, B., AND SERAFINI, M. Zab: High-performance broadcast for primary-backup systems. In *DSN* (2011), IEEE/IFIP.

[18] KALLMAN, R., KIMURA, H., NATKINS, J., PAVLO, A., RASIN, A., ZDONIK, S., JONES, E., MADDEN, S., STONEBRAKER, M., ZHANG, Y., ET AL. H-store: a high-performance, distributed main memory transaction processing system. *VLDB* (2008).

[19] KRASKA, T., PANG, G., FRANKLIN, M. J., AND MADDEN, S. Mdcc: Multi-data center consistency. *CoRR* (2012).

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

[21] LAMPORT, L. The part-time parliament. *TOCS* (1998).

[22] LLOYD, W., FREEDMAN, M., KAMINSKY, M., AND ANDERSEN, D. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *SOSP* (2011), ACM.

[23] MALKHI, D., BALAKRISHNAN, M., DAVIS, J., PRABHAKARAN, V., AND WOBBER, T. From Paxos to CORFU: a flash-speed shared log. *SIGOPS Operating Systems Review* (2012).

[24] 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).

[25] PAVLO, A., JONES, E., AND ZDONIK, S. On predictive modeling for optimizing transaction execution in parallel oltp systems. *VLDB* (2011).

[26] PENG, D., AND DABEK, F. Large-scale incremental processing using distributed transactions and notifications. In *OSDI* (2010), USENIX.

[27] SOVRAN, Y., POWER, R., AGUILERA, M., AND LI, J. Transactional storage for geo-replicated systems. In *SOSP* (2011), ACM.

[28] THE APACHE SOFTWARE FOUNDATION. Apache hbase. http://hbase.apache.org, retrieved Dec. 2012.

[29] VAN RENESSE, R., AND SCHNEIDER, F. Chain replication for supporting high throughput and availability. In *OSDI* (2004).