Transactional Failure Recovery for a Distributed Key-Value Store

Muhammad Yousuf Ahmad, Bettina Kemme, Ivan Brondino, Marta Patiño-Martínez, Ricardo Jiménez-Peris

Transactional failure recovery for a distributed key-value store

Muhammad Yousuf Ahmad[1], Bettina Kemme[1],
Ivan Brondino[2], Marta Patiño-Martínez[2] and Ricardo Jiménez-Peris[2]

[1] McGill University
[2] Universidad Politécnica de Madrid

Abstract. With the advent of cloud computing, many applications have embraced the ensuing paradigm shift towards modern distributed key-value data stores, like HBase, in order to benefit from the elastic scalability on offer. However, many applications still hesitate to make the leap from the traditional relational database model simply because they cannot compromise on the standard transactional guarantees of atomicity, isolation, and durability. To get the best of both worlds, one option is to integrate an independent transaction management component with a distributed key-value store. In this paper, we discuss the implications of this approach for durability. In particular, if the transaction manager provides durability (e.g., through logging), then we can relax durability constraints in the key-value store. However, if a component fails (e.g., a client or a key-value server), then we need a coordinated recovery procedure to ensure that commits are persisted correctly. In our research, we integrate an independent transaction manager with HBase. Our main contribution is a failure recovery middleware for the integrated system, which tracks the progress of each commit as it is flushed down by the client and persisted within HBase, so that we can recover reliably from failures. During recovery, commits that were interrupted by the failure are replayed from the transaction management log. Importantly, the recovery process does not interrupt transaction processing on the available servers. Using a benchmark, we evaluate the impact of component failure, and subsequent recovery, on application performance.

Keywords: Cloud computing, key-value store, transaction processing, OLTP, fault tolerance, failure recovery

1 Introduction

Traditional online transaction processing (OLTP) applications generally cannot compromise on the basic transactional guarantees of atomicity, consistency, isolation, and durability. On the other hand, modern distributed key-value data stores do not provide transactional semantics out of the box. One way to bridge this gap is through the integration of an independent transaction management component with a distributed key-value store. This approach can allow a traditional OLTP application to benefit from the elastic scalability of cloud computing

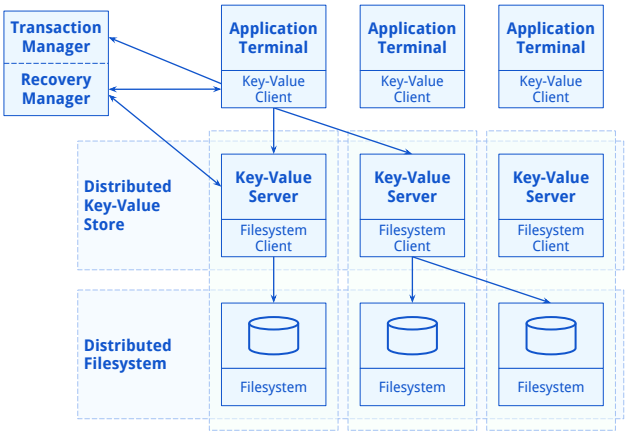


Fig. 1. System Architecture

infrastructure without sacrificing on transactional semantics. Providing transactional properties for key-value stores has been proposed in several contexts [7, 8, 17, 19, 20]. Figure 1 shows the principle architecture of our approach. The interface of the key-value store is enhanced to provide the transactional primitives *begin*, *commit*, and *abort*, and all read and write accesses to the key-value store are encapsulated in a transactional context. Existing solutions vary in how they implement the various transactional properties as well as how tightly or loosely coupled transaction management is with the data store. We believe that transaction management is ideally as much separated from the data storage as possible, i.e., it should mainly be the transaction management component, rather than the key-value store, that provides the transactional properties.

While the tasks of isolation and atomicity have been the main topic for most of the work so far, relatively little attention has been given to durability. In general, current key-value stores provide durability as they typically persist each update they receive, i.e., they provide durability on a per-object basis. This can be exploited by the transaction manager, since all updates are automatically durable at the time of commit. However, this per-object durability is very costly, in particular as many key-value stores use a reliable file system for persistence that leads to further distribution and replication. Thus, end-to-end latency suffers significantly.

In fact, the cost of writing all changes individually to stable storage before commit is already considered unacceptable in traditional monolithic database systems. Instead, they provide durability with the help of a recovery log [10]. All the changes a transaction performs are recorded in an append-only recovery log that is made persistent just before the transaction commits with a single I/O operation. In contrast, the updates on the actual data pages are not written to stable storage before commit, since this is expensive in terms of I/O. In case of a

failure, the updates in the recovery log are replayed to put the stable storage back into a consistent state. As it would be extremely costly as well as unnecessary to replay the entire recovery log since system startup (as most updates are likely to be already persisted), checkpoints are used to accelerate the recovery process.

In this paper, we propose to use a similar mechanism within the transaction manager of our transactional cloud data store: the transaction manager takes responsibility for the durability of a committed transaction by means of a recovery log, while the key-value store does not need to persist its updates immediately upon receiving them. Rather, it can almost instantly store the updates in, and serve them out of its main memory, thereby reducing end-to-end latency. In fact, when durability is achieved through the recovery log maintained by the transaction manager, it might not even be necessary to send updates to the key-value store before the commit, further increasing the performance during standard processing. However, given the more complex architecture of the multi-layered system, both checkpointing as well as recovery become more complicated, since a transaction crosses several layers and might perform updates on several data store servers. As a result, figuring out the set of transactions that are actually affected by a failure becomes more complicated. Furthermore, both client and server failures have to be considered. In the event that a key-value server fails, updates that were still in the server's memory (and not yet persisted to the underlying reliable filesystem) must be replayed. In case a client fails, we must replay the updates of transactions that were already committed but whose writes were not yet fully flushed to the data store.

In summary, this paper presents a comprehensive solution to recovery in a multi-tier transactional key-value store where most of the transaction management tasks are performed by a middleware-based transaction manager. In particular we make the following contributions.

- We rely on efficient transaction logging for persistence in order to provide fast execution for OLTP workloads. As such, there are no forced flushes to stable storage at the key-value store. Updates can even be sent to the key-value store after commit.
- We handle failures of key-value clients and servers. Upon failure, we determine exactly which updates were not persisted at the server-side, and replay those updates before resuming normal processing. For this, we perform lightweight observations at clients and servers during normal processing, somewhat similar to checkpointing in a traditional database system.
- Our approach attempts to separate transaction management as much as possible from data processing. Our extensions to the key-value store to provide a transactional interface and failure recovery are kept to a minimum.
- We have implemented and evaluated our approach using HBase[3], a distributed key-value store that uses HDFS[4] for persistence. Our analysis shows that recovery-related tasks during normal processing incur little overhead and recovery is smooth and efficient.

[3] http://hbase.apache.org/
[4] http://hadoop.apache.org/

2 System Model

In this section, we discuss the main components of both the distributed key-value store and an independent transaction manager that are relevant for durability and atomicity. Note that, while we assume that the system also implements some form of concurrency control in order to achieve isolation, isolation and concurrency control are outside the scope of this paper.

2.1 HBase

HBase is a modern distributed key-value data store based on Google BigTable [5]. We believe that it is a representative candidate since many other key-value stores have similar features. HBase offers the abstraction of a table. Each row in a table represents a key-value pair – it is uniquely identified by a key and can have an arbitrary number of secondary columns. Each table is partitioned into one or more chunks called regions. A region is a contiguous set of rows sorted by key. Every region is assigned to one of the multiple region servers in the HBase cluster. A master server coordinates region assignment. Through well-balanced region placement, the application workload can be evenly distributed across the cluster. Moreover, when the existing region servers become overloaded, new region servers can be added dynamically, thus allowing for elastic scalability.

HBase persists its data in the Hadoop Distributed File System (HDFS), which is a reliable and scalable filesystem based on the Google File System [9]. In the HDFS layer, a region is physically stored as one or more immutable files.

An HBase region server serves read requests by fetching the requested data from the underlying filesystem. Typically, it has a large main-memory cache to reduce interactions with HDFS. Additionally, it also maintains, per region, an in-memory store (memstore) that stores the latest updates performed on that region. The contents of each memstore are flushed to HDFS in a batch.

When a server fails, its regions are re-assigned to other servers. Although the persisted data of a region can be read back from HDFS, any in-memory updates that had not yet been persisted to the filesystem are lost. To provide durability against this data loss, HBase first persists each incoming update to its write-ahead log (also stored in HDFS) before applying it to the in-memory store. In this way, after a server failure, the HBase recovery procedure is able to recover lost in-memory updates by replaying them from its log, thus bringing the data store back to a consistent state. As soon as a region has been re-assigned to an available server and recovered to a consistent state, it is made available once again. Each region server maintains a single write-ahead log, to which all updates, whichever region they belong to, are appended. Therefore, the recovery procedure first needs to split the log file and group the updates by region. Once this is done, any region server attempting to recover a region's data can simply read the group of updates associated with that region and apply them to a freshly initialized memstore. Note that, for performance reasons, HBase allows the deactivation of a synchronous flush of the write-ahead log to HDFS. In this case, the server may return from an update operation before the update is

persisted to HDFS. In this case, HBase's own recovery cannot guarantee that all updates executed before a server's failure will be recovered.

Applications interact with HBase through an embedded HBase client, a library that provides an advanced interface with get/put operations to access individual key-value pairs and filtered scans for fetching larger result sets.

2.2 Transaction Management

Applications interact with the system through one or more client processes. The task of the independent transaction management is to provide transactional properties (isolation, durability and atomicity) to the application. The application must be provided with an appropriate interface so that it can start, commit, and abort transactions. Also, the system must be able to associate read and write operations to the data store with a unique transaction context. We assume each transaction is executed by a single client and it may touch one or more key-value servers. A client can execute multiple transactions concurrently.

Transaction Execution. The HBase client is the interface between the application and the HBase servers. Thus, the HBase client must be extended to offer a transactional interface to the client. It is also the key player that interacts with the transaction manager, while the modifications to the HBase server have been kept to a minimum. The extended client interface needs to provide the transactional primitives *begin*, *commit*, and *abort*. When the application calls *begin*, a transactional context is created, and all subsequent read/write calls, as well as a final commit/abort, can be associated with this context. The creation and management of the transactional context is the responsibility of the transaction management component and therefore is outside the scope this paper.

In our current approach, we assume that transaction execution follows a deferred-update approach, i.e., updates are not propagated to the HBase servers before commit. This approach is beneficial since it greatly reduces the communication between remote components. For our solution, it is not relevant where the updates are buffered (they could be buffered at the application, the HBase client, or the transaction manager). In our concrete implementation, we keep the write-set of a transaction (i.e., the set of values that a transaction inserts, updates, or deletes), at the HBase client. We assume that updates are idempotent, i.e., applying the write-set at the key-value store multiple times produces the same result every time. This is possible since HBase allows us to specify a version number for each update, and we stamp each transaction's write-set (i.e., each of its updates) with a unique version number, i.e., the commit timestamp of that transaction.

When the application calls *commit*, the transaction termination phase starts. If the transaction manager decides that the transaction can commit, the transaction receives a commit timestamp and its write-set, together with the commit timestamp and a client identifier, is flushed to the recovery log to make it persistent. At this point, the transaction is considered committed. The write-set is

flushed to the HBase servers only after the commit. Note that a transaction might have updates for several servers and many updates for a single server. Thus, the flush is usually a non-atomic operation. Once the entire write-set has been flushed, transaction termination is completed. Depending on the concurrency control mechanism implemented in the transaction manager, there could be several transactions terminating concurrently and they could flush their write-sets in any order. The transaction manager would have to ensure that serializability guarantees are not violated (basically, these transactions should not conflict). Focusing on durability and recovery, we assume that commit timestamps are monotonically increasing and that the commit timestamp determines the serialization order for transactions. In other words, if the recovery procedure applies write-sets in commit timestamp order, then this produces a correct execution.

If the application submits an abort request or the transaction manager decides to abort a transaction, the buffered write-set can simply be discarded. It is not stored in the recovery log nor flushed to the HBase servers.

As the transaction manager guarantees the immediate durability of committed write-sets, we deactivate the synchronous flushing of the HBase write-ahead log. Thus, upon receiving an update, the HBase server first appends it to its (in-memory) write-ahead log buffer, then applies it to the memstore, and then immediately returns to the client. Shortly thereafter, (i.e., asynchronously), we sync the write-ahead log buffer to HDFS. At some point later, the actual updates in the memstore are eventually persisted to HDFS.

Transaction Phases. In summary, an update transaction can be in one of the following states at any given time.

- *executing* – the transaction has been started but not yet committed or aborted by the transaction manager
- *aborted* – the transaction manager has aborted the transaction
- *committed* – the transaction manager has declared the transaction as committed, having persisted the write-set to its log
- *flushed* – the committed write-set has been received by all participating HBase servers and applied to their in-memory stores
- *persisted* – the flushed write-set has been persisted by all participating servers (at least the HBase write-ahead log was persisted to HDFS)

3 Recovery management

In this section, we present our implementation of a failure detection and recovery service for the integrated system. The primary aim is to detect client and server failures and recover from them reliably so that we do not violate the atomicity and durability of committed transactions. The HBase client and server components have been enhanced to provide the necessary support for recovery purposes. Furthermore, we implement a recovery manager, which is a middleware service associated with the transaction manager, that coordinates failure detection and recovery actions across clients and servers.

When an HBase client fails, the recovery manager's recovery procedure replays from the transaction manager's recovery log any of the client's write-sets that were committed but not yet completely flushed to their participating servers. This is necessary because our client holds the write-set until commit time, so we may lose the committed write-set if a client failure occurs before or during the flush phase. Note that write-sets that were not yet committed to the transaction manager's recovery log are permanently lost when the client fails. These transactions are considered aborted. This is not a problem because only the durability of committed transactions must be guaranteed. Uncommitted transaction can be restarted on another client by the application.

When an HBase server fails, the recovery procedure has to replay all committed write-sets that this server participated in that were flushed by the client but not persisted before the server failure occurred. This is necessary because our server persists received write-sets asynchronously, so we may lose a write-set if a server failure occurs before or during the persist phase. Once a write-set has been fully persisted by its participating servers, we can then rely on the key-value store to guarantee the durability of the write-set.

In principle, it would be correct if the recovery manager simply replays all write-sets that exist in the recovery log, as replaying write-sets is idempotent. That is, if a write-set is already reflected in the data store, replaying it will not lead to a different state. However, replaying all write-sets would be extremely inefficient. In a traditional database system, relevant checkpointing information is added to the recovery log during normal processing to determine during recovery the subset of transactions that actually have to be replayed. In a distributed environment, obtaining such checkpointing information is, however, more complex. In our solution, the recovery manager relies on the HBase client and server components to provide the relevant information. On a regular basis, both send the information of what they have flushed and persisted, respectively. This information can be sent asynchronously, or even periodically, as it will serve as a lower bound on which write-sets have to be replayed in case of recovery.

In the following, we first show how the HBase client and recovery manager collaborate in order to: (1) be able to detect client failures, (2) keep track of the transactions that the client has already flushed to the servers, and (3) recover after a client failure. Then we show the corresponding steps for the HBase server.

3.1 Handling Client Failures

Algorithm 1 describes what is done within the HBase client. Algorithm 2 describes the actions performed by the recovery manager to handle client failures.

Client Failure Detection. To detect client failures, we implement a simple heartbeat mechanism. When a client initializes, it registers its heartbeat with the recovery manager, which then starts to monitor the heartbeat. The client regularly sends heartbeat messages to the recovery manager with a configurable frequency. When the client completes execution correctly, it unregisters from

Algorithm 1 At client c

```
1: On startup:
2:     register(c)                                  ▷ register with recovery manager
3:     T_F ← T*_F                                   ▷ local ts threshold
4:     FQ ← synchronized priority queue             ▷ committed txns in commit order
5:     FQ' ← synchronized priority queue            ▷ flushed txns in commit order
6: On shutdown:                                     ▷ clean shutdown
7:     heartbeat()                                  ▷ pre-shutdown heartbeat
8:     unregister(c)                                ▷ unregister with recovery manager
9: On heartbeat:                                    ▷ called periodically
10:    while |FQ| > 0 AND |FQ'| > 0 do
11:        if FQ.head = FQ'.head then               ▷ earliest tracked flush completed?
12:            T_F ← FQ'.head                       ▷ make local progress
13:            FQ.dequeue()                         ▷ remove its trackers
14:            FQ'.dequeue()
15:        else
16:            break                                ▷ respect local commit ordering
17:    send_heartbeat(c, T_F)                       ▷ to recovery manager
18: On receiving commit timestamp T:                ▷ received commit ts
19:    FQ.enqueue(T)                                ▷ add commit tracker
20: On post-flush of transaction T:                 ▷ called by commit protocol after flush
21:    FQ'.enqueue(T)                               ▷ add post-flush tracker
```

the recovery manager cleanly. However, if the recovery manager detects that the client has missed successive heartbeats, it declares the client dead and immediately initiates a recovery procedure. Since we treat a network partition as a crash failure, if any further messages are received from a dead client, they are ignored until the recovery procedure is completed. If a network partition is the cause, the client heartbeat will not be able to contact the recovery manager, which will result in it terminating itself.

Client Tracking. The recovery manager relies on the HBase clients to keep track of when write-sets are flushed to the server. Each client piggybacks the relevant information on its heartbeat messages. In a simple approach, the client could simply send to the recovery manager the commit timestamps of all transactions for which it has completely flushed the write-set to all participating servers. However, that can incur considerable overhead in terms of message size. Instead, each client c maintains a threshold timestamp $T_F(c)$ and sends this timestamp with its heartbeat messages. $T_F(c)$ obeys the following local invariant: the write-set of every local transaction, executing at this client c, with commit timestamp T smaller than or equal to $T_F(c)$ (i.e., where $T \leq T_F(c)$) has been fully flushed to its participant servers. Periodically, we advance $T_F(c)$ as local transactions are committed and flushed. $T_F(c)$ increases monotonically, in increments that correspond to the local commit sequence. In other words, for any two local trans-

Algorithm 2 At recovery manager (client related)

1: **On register(c):** ▷ register client
2: C.add(c)
3: $T_F^r(c) \leftarrow T_F^*$
4: **On unregister(c):** ▷ unregister client
5: C.remove(c)
6: T_F^r.remove(c)
7: **On receive_heartbeat(c, T_F):**
8: $T_F^r(c) \leftarrow T_F$ ▷ keep track of threshold
9: $T_F^* \leftarrow \forall i \in C : min(T_F^r(i))$ ▷ update global flushed ts threshold
10: **On failure(c):** ▷ client failure detected (missed heartbeats)
11: $L \leftarrow$ fetch_logs(c, $T_F^r(c)$) ▷ fetch from log txns committed by c after $T_F^r(c)$
12: **for each** $(T, WS(T))$ **in** L **do**
13: c_R.flush(T, $WS(T)$) ▷ replay write-set using recovery client c_R

actions with commit timestamps $T_i < T_j$, $T_F(c)$ will always advance from T_i to T_j, even if the flush of T_j is completed before that of T_i.

Maintaining $T_F(c)$ is not trivial, since c may flush its transactions in any order. However, the transaction manager ensures that c's transactions receive commit timestamps that are monotonically increasing, i.e., if T receives its timestamp before T', then $T < T'$. In our implementation, a client keeps track of $T_F(c)$ with the help of two queues: FQ keeps track of all transactions in the commit phase, and FQ' keeps track of all transactions that have been successfully flushed. When the timestamps at the heads of both queues match up, we can dequeue that timestamp and advance $T_F(c)$ accordingly. Thus, by adding transactions to FQ in commit timestamp order, we guarantee that $T_F(c)$ is advanced in the proper order.

For each client c, the recovery manager keeps track of the threshold timestamp $T_F^r(c)$ it has received through the last heartbeat message received from c. Due to the periodic delay in heartbeat messages, $T_F^r(c)$ is a conservative threshold representing the flushing process at client c. In order words, while c might have progressed further than $T_F^r(c)$ since its last heartbeat was received, the recovery manager uses $T_F^r(c)$ to represent the current state of c's progress. No transactions with timestamp $T < T_F^r(c)$ have to replayed in case c fails.

Furthermore, the recovery manager maintains a global client threshold $T_F^* = \forall c : min(T_F^r(c))$, which is the lowest $T_F^r(c)$ among all clients. It represents a system-wide threshold that upholds the following global invariant: all transactions that were committed up until time T_F^* have been flushed to and received in full by their participant servers.

Note that maintaining a conservative threshold means that some write-sets might be replayed unnecessarily during recovery. However, this overhead only presents itself during the recovery process and does not affect performance during normal operation. Moreover, the number of write-sets that need to be recovered upon failure is bound by the client's throughput and heartbeat interval.

Finally, in order to ensure that T_F^* advances, clients that shut down properly have to unregister cleanly so that the recovery manager does not take them into consideration anymore for maintaining T_F^*.

Client Recovery. When the recovery manager detects that a client c has failed, it will fetch and replay from the transaction management log those write-sets that were committed by c *after* time $T_F^r(c)$, which is the $T_F(c)$ received by the recovery manager with the most recent heartbeat from c (before it failed). The recovery manager replays these updates via its local client c_R, which differs from a regular client in that it replays the recovered updates using the commit timestamp of the original transaction, rather than obtaining a new one.

3.2 Handling Server Failures

Algorithm 3 describes what is done within the HBase server. Algorithm 4 describes the actions performed by the recovery manager to handle server failures.

Server Failure. For server failure, we depend on HBase to notify us whenever one of its servers dies. Internally, HBase, too, uses heartbeats to monitor server health. When HBase detects that one of its servers has died (due to a crash failure or network partition), the master server initiates a recovery procedure that reassigns the regions of the failed server to other live servers. We added a hook in the master server that notifies our recovery manager whenever a server fails. Each affected region, upon being reassigned to some live server, undergoes HBase's internal recovery procedure during initialization as outlined in Section 2.1. Note that different regions can be assigned to different servers leading to parallel recovery. Recovery replays any un-persisted updates that are associated with this region from the HBase write-ahead log of the failed server. We add another hook in the region initialization process that notifies our recovery manager once this internal recovery procedure is completed, and then waits for a response from our recovery manager before proceeding to actually declare the region online. When the recovery manager receives the notification from the hook, it initiates our transactional recovery procedure for the region. Once our recovery procedure is completed, we notify the region waiting on us that it may proceed to declare itself online. Delaying transaction execution until our recovery procedure is completed, ensures that transactional atomicity is not violated, since, if a region affected by a server failure is brought online before our recovery manager has supplemented the internal region recovery process, clients can potentially end up reading partially recovered write-sets.

Server Tracking. Similar to the client case, each server keeps track of up to which transaction the received write-sets have been persisted to HDFS (i.e., to the HBase write-ahead log). It also sends this information to the recovery manager via regular heartbeat messages. Persisted transactions do not need

Algorithm 3 At server s

```
1:  On startup:
2:      register(s)                                  ▷ register with recovery manager
3:      T_P ← T*_F                                   ▷ local ts threshold
4:      PQ ← synchronized priority queue             ▷ received write-sets in commit order
5:  On shutdown:                                     ▷ clean shutdown
6:      heartbeat()                                  ▷ pre-shutdown heartbeat
7:      unregister(s)                                ▷ unregister with recovery manager
8:  On heartbeat:
9:      T'_P ← T*_F                                  ▷ read latest T*_F from recovery manager
10:     while |PQ| > 0 do
11:         (T, WS(T)) ← PQ.dequeue()
12:         persist(WS(T))                           ▷ persist write-set
13:     T_P ← T'_P                                   ▷ make local progress
14:     send_heartbeat(s, T_P)                       ▷ to recovery manager
15: On receive(T, WS(T)):                            ▷ received write-set from client
16:     apply(WS(T))                                 ▷ apply updates to in-memory store
17:     PQ.queue((T, WS(T)))                         ▷ add tracker
18: On receive(T, WS(T), T_P(s')):                   ▷ received write-set from recovery client
19:     receive(T, WS(T))                            ▷ process as usual
20:     if T_P(s') < T_P then
21:         T_P ← T_P(s')                            ▷ inherit responsibility for replayed updates
22:         heartbeat()                              ▷ persist and inform recovery manager
23: On opening_region(r):                            ▷ hook after internal recovery completed...
24:                                                  ▷ but before declaring r online
25:     wait                                         ▷ until transactional recovery completed...
26:     until is_recovered(r)                        ▷ by recovery manager
```

to be replayed during recovery in case the server fails. To keep track of these transactions in a compact form, each server s maintains a threshold timestamp $T_p(s)$ that obeys the following local invariant: the write-set of every transaction with commit timestamp T smaller than or equal to $T_P(s)$ (i.e., where $T \leq T_p(s)$), and where the server is a participant, has been received in full by the server and fully persisted (that is, the part of the write-ahead log containing these write-sets has been written to HDFS).

However, it is not that simple for a server to deduce that this invariant holds. While clients know exactly which transactions are currently active, and thus, know which transactions with lower timestamps have been completely flushed, things are not as simple at the server. For instance, assume a server has received and persisted write-sets of transactions with timestamps 20, 22, and 23, but misses 21. Then, it could be that the server is not a participant in transaction 21, in which case, its $T_p(s)$ should be set to 23; but it could also be that the client executing transaction 21 has simply not yet flushed this write-set (but will do so in the future), in which case, $T_p(s)$ should be held at 20.

Algorithm 4 At recovery manager (server related)

1: **On register(s):** ▷ register server
2: S.add(s)
3: $T_P^r(s) \leftarrow T_P^*$
4: **On unregister(s):** ▷ unregister server
5: S.remove(s)
6: T_P^r.remove(s)
7: **On receive_heartbeat(s, T_P):**
8: $T_P^r(s) \leftarrow T_P$ ▷ update threshold ts
9: $T_P^* \leftarrow \forall i \in S : min(T_P^r(i))$ ▷ update global persisted ts threshold
10: **On failure(s):** ▷ notified of server failure (by key-value store)
11: $L \leftarrow$ fetch_logs($T_P^r(s)$) ▷ fetch from log txns committed after $T_P^r(s)$
12: $R \leftarrow$ affected_regions(s) ▷ fetch from master server
13: **for each** $r \in R$ **do** ▷ recover each affected region one-by-one
14: **for each** $(T, WS(T))$ **in** L **do**
15: replay(T, $WS(T)$, s, r)
16: notify_region(r) ▷ notify region so it can go online
17: **On replay(T, $WS(T)$, s, r):** ▷ replay write-set using recovery client c_R
18: $T' \leftarrow T$ ▷ new txn with same commit timestamp
19: **for each** u **in** $WS(T)$ **do** ▷ for each update in recovered write-set
20: **if** u.region $= r$ **then** ▷ if the update u falls in region r
21: $WS(T')$.add(u) ▷ else skip other updates
22: **if** $|WS(T')| > 0$ **then** ▷ if any updates were selected
23: c_R.flush(T', $WS(T')$, $T_P^r(s)$) ▷ replay recovered updates

Therefore, we use a conservative value for $T_p(s)$ by ensuring that $T_P(s) \leq T_F^*$. For any committed transaction with timestamp $T \leq T_F^*$, we know that its write-sets have been successfully flushed to all participating servers. That is, if a server s has received and persisted a transaction $T \leq T_F^*$, then s knows that it also has received all transactions with timestamp $T' < T$. Therefore, periodically, we can persist all write-sets received up until T_F^* and then advance $T_P(s)$ to T_F^*. For that purpose, each server has to receive the latest value of T_F^* from the recovery manager on a regular basis.

For each server s, the recovery manager keeps track of the threshold timestamp $T_P^r(s)$ it has received through the last heartbeat message received from s. $T_P^r(s)$ is a conservative threshold of what s has persisted so far. No transactions with timestamp $T < T_P^r(s)$ have to be replayed in case s fails.

Furthermore, we saw earlier that $T_P^r(s) < T_F^*$. Based on this information, the recovery manager can declare that $T_P^* = \forall s : min(T_P^r(s))$, i.e., the lowest $T_P^r(s)$ among all servers, is a system-wide threshold that upholds the following global invariant: all transactions that were committed up until time T_P^* have been flushed to, received in full, and safely persisted by their participant servers. Therefore, T_P^* also represents a global checkpoint for the purposes of commit

logging and failure recovery. That is, transactions with timestamp $T < T_P^*$ may be truncated from the recovery log since they have been safely persisted.

Server Recovery. In the event of a server failure, we lose the state of the in-memory store on that server. Since we persist these updates and the HBase write-ahead log asynchronously, lost updates must be recovered from the transaction management log to ensure their durability.

When the recovery manager detects that a server s has failed, it will first wait for the key-value store to perform its standard recovery process to recover, one-by-one, the set $R(s)$ of regions affected by this failure. For each affected region r, once its in-memory store has been reconstructed by HBase, our recovery manager will take over. It will replay from the transaction manager's recovery log those write-sets that were committed *after* time $T_P^r(s)$, which is the piggybacked value received by the recovery manager with the most recent heartbeat from server s (before it failed). These are the write-sets that have potentially not yet been persisted. The recovery manager replays them via its local client c_R. Once our recovery process for a region r has completed, the region is brought back online. Server recovery is complete once all affected regions have been recovered.

There are three important ways in which the recovery client c_R differs from a regular client. First, it replays the recovered updates using the commit timestamp of the original (recovered) transaction, rather than requesting a fresh commit timestamp from the transaction manager. This applies also to client recovery. The other two modifications apply only to server recovery. During server recovery, when replaying a write-set to an affected region r, the recovery client checks each update in the write-set to see if it falls within r, replaying it if it does, and skipping it otherwise. This means that we only replay those updates that were left un-persisted due to this specific server failure. Secondly, the recovery client piggybacks $T_P^r(s)$ on every replayed update when performing the recovery procedure for a failed server s. To understand why this is necessary, consider the following scenario. During recovery, we replay an update u, which belongs to the write-set of some transaction T, where $T_P^r(s) < T$, and which falls under region r, one of the affected regions in $R(s)$. A live server s', which now hosts r, receives u, applies it to its in-memory store, queues it for persistence, and then returns to the client. At this point, if s' fails, we can end up losing u under the following condition: $T_P^r(s) < T \leq T_P^r(s')$, since the recovery procedure for s' will only recover write-sets of transactions committed later than $T_P^r(s')$. In order to avoid this situation, once we add u to the in-memory store of s', we modify $T_P(s')$ to $T_P^r(s)$, before returning to the client. This ensures that s' correctly inherits the responsibility for the recovered updates of s.

One scenario that we must also consider is that a server failure will interrupt any incoming client flushes. A client c in this situation will retry, multiple times, to flush the remaining part of the write-set to the target regions. As soon as the affected regions are recovered and brought back online, the client will be able to proceed again normally and complete any interrupted flushes. However, if a client flush eventually runs out of retries or times out, $T_F(c)$ can be permanently

blocked from advancing, even after the affected regions comes back online a little later. Even though other concurrent flushes of the same client c may have been unaffected by the failure, we cannot advance $T_F(c)$, since it must advance in step with the local commit order at c. This will then block the progress of our global timestamp thresholds, since T_F^* is bound to the lowest $T_F(c)$ among all clients, and T_P^* is bound to T_F^*. Therefore, we work around this by removing the retry and timeout limits so that the client keeps retrying until it succeeds. During this time, the client can at least continue to execute read-only transactions on older snapshots of the data. Alternatively, we could terminate the client to induce the recovery manager to attempt a recovery of the interrupted write-set.

If a region remains unavailable forever (i.e., it cannot be recovered for some reason), then a system administrator must intervene and manually recover the region. In order to detect such incidents, each client/server monitors the size of its flush/persist queue (which reflects the transactions flushed/persisted, but not yet reflected in the corresponding threshold timestamps) and alerts the recovery manager if the size exceeds a configurable threshold. Once the problematic region is recovered, the blocked timestamp thresholds are able to advance until they become current again.

3.3 Recovery Manager Failure

As a final note, the failure of the recovery manager also has to be considered. The only data the recovery manager maintains are the threshold timestamps. These timestamps can also be written to the recovery log periodically or stored in a highly reliable service such as ZooKeeper[5]. Our implementation uses ZooKeeper for coordination between the recovery manager and clients/servers (i.e., heartbeat messages are exchanged via ZooKeeper). Upon failure, the recovery manager is restarted and contacts ZooKeeper to catch up with the system's progress. Transaction processing can continue while the recovery manager is down.

4 Performance Evaluation

In this section, we present a preliminary performance evaluation of our integrated implementation. We first look at the performance benefits of asynchronous versus synchronous persistence. Next, we show that the overhead for providing a reliable transaction processing framework for HBase, using our failure recovery scheme, is small. Finally, we look at the effects of a server failure on runtime performance. Our experiments measure transaction throughput and response time.

4.1 Benchmark and Setup

We use YCSB[6] to evaluate our implementation. We extended YCSB to support true transactional workloads and implemented a simple type of update

[5] http://zookeeper.apache.org/
[6] http://github.com/brianfrankcooper/YCSB

transaction that executes 10 random row operations, with a 50/50 ratio of reads/updates. We loaded our test table with half a million rows.

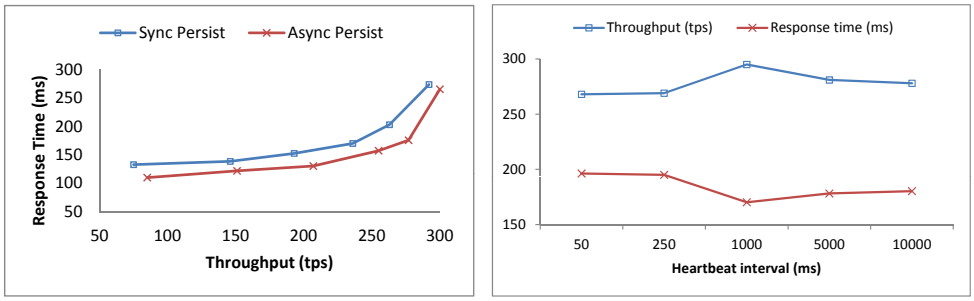
We ran our experiments on virtual machines hosted on a cluster of Dell R310 quad-core servers. Each VM was allocated two processor cores and 2 GB of main memory. The machines were connected over a 100 Mbps Ethernet switch. We ran our experiments using one client node and two server nodes. On each server node, we ran an HBase region server co-located with an HDFS datanode. We allocated two thirds of the region server's available memory for the block cache (for reads) and the remaining one third for the memstore (for updates). The size of our test dataset was chosen such that it could fit completely into the cumulative block cache of a single region server, so that we could compensate for the failure of one of the servers. We used a data replication factor of two (instead of the default of three) in HDFS. We populated a fresh dataset and warmed up the block cache before the start of each experiment.

The transaction management and recovery management components were co-hosted on one VM. The transaction management component provides an efficient concurrency control mechanism based on snapshot isolation. Its internal structure is highly scalable and fully reliable. The overall architecture of the transaction management component will soon be submitted for publication in an independent manuscript, and thus, is not further detailed here. The logging sub-component supports group commit, has access to its own high performance stable storage, and can be distributed across several nodes should one logging node not be sufficient. It offers the interface methods for the recovery manager to retrieve the necessary logs at the time of recovery.

4.2 Benefits of Asynchronous Persistence

We evaluated the advantages of persisting updates asynchronously to the key-value store. We used two region servers. Figure 2(a) shows a performance comparison between synchronous and asynchronous persistence. The graph shows response time (in milliseconds) against throughput (in transactions per second (tps)). We achieve lower response times with asynchronous persistence. These results reflect our original premise that asynchronous persistence offers a performance advantage because it eliminates the latency associated with flushing and persisting updates to the key-value store from end-to-end response times.

Note that our transactions are quite short, in which case, transaction management related tasks make up a considerable part of the execution time (timestamp management, logging, etc.). With longer transactions that perform more read and write operations one can expect a larger performance gain. Also, we can expect the gap between the two curves to be greater if the HDFS data nodes are not physically co-located with the HBase servers, as that worsens end-to-end latency under synchronous persistence.



(a) Asynchronous persistence benefits (b) Transaction tracking overheads

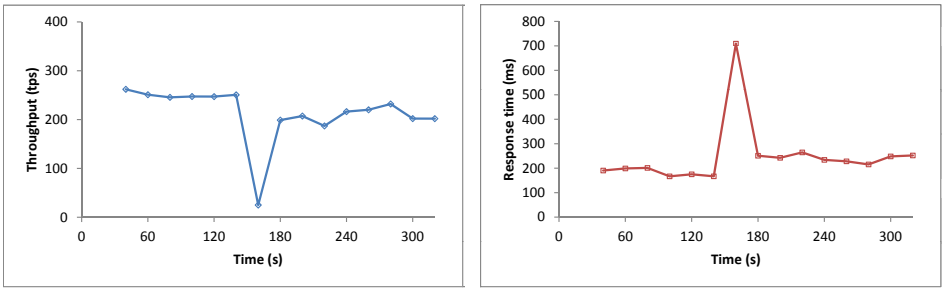
Fig. 2. Performance improvements and overheads

4.3 Overhead of Providing Reliability

We evaluated the overhead associated with providing reliability under the asynchronous persistence approach. Each client and server component performs some light-weight tracking of their local transaction progress. This tracking involves the use of synchronized data structures. Periodically, just before sending its heartbeat, each component updates its local tracking information, which the recovery manager then uses to update its global trackers. The system throughput and the length of the heartbeat interval together determine the amount of processing performed with each heartbeat. The shorter the interval, the more frequently we update the tracking information and the less the information processed per heartbeat. On the other hand, our tracking data structures need to be synchronized, since they are accessed concurrently by multiple threads. Thus, updating the tracking information too frequently can potentially reduce performance due to added contention. We use trial-and-error to determine a good heartbeat interval. In our experiments, we varied this interval from 50 ms to 10 seconds for 50 client threads and two region servers. As Figure 2(b) shows, both throughput and response time vary as a function of the heartbeat interval, and we are able to find a good interval value for our setup.

4.4 Evaluating Failure Recovery

We evaluated the effects of a component failure on transaction processing. We measured the runtime performance of 50 client threads with two region servers. We simulated a workload of 250 tps, which in our setup is near the peak capacity for a single region server serving 50 client threads. We set the heartbeat intervals to one second. In Figure 3, runtime throughput and response time readings are plotted on the vertical axis against wall-clock time on the horizontal axis. We manually induced a server failure during the experiment. The server crash causes a sharp drop in throughput and a corresponding peak in response time. The performance returns to nearly pre-failure readings over the next 30 seconds. In



(a) Throughput (b) Response time

Fig. 3. Failure detection and recovery

our tests, we observed that the actual recovery process takes only a few seconds, whereas the longer delay in returning to pre-failure performance levels is due to the region server cache taking a while to warm up to the recovered regions' data. Note that transaction processing is not interrupted (i.e., transactions are not lost) by the failure.

5 Related work

There has been considerable work in adding transaction functionality to key-value stores [7, 8, 17, 19, 20], some of which discusses logging and recovery to some extend. CloudTPS [20] supports scalable transactions by distributing the transaction management among a set of local transaction managers and partitioning the data using the key-value store for persistence. Similar to our model, the updates are not written to the storage during commit but buffered in-memory for performance reasons and only sent to the key-value store periodically. Updates are replicated across several local transaction managers (LTMs) to guarantee availability in the advent of failures. This requires coordination among the replicas. If there is a failure, data is recovered from a LTM replica. Since some data items in the replica may have been written to storage, to avoid repeating writes, they keep track and replicate in some LTMs the latest checkpointed timestamp for each data item. Again timestamps are replicated in batches. Compared to our approach, the checkpointing overhead is considerably larger as it is on a per-item basis vs. per-transaction basis. Furthermore, their approach assumes that once data is written to the data store it is persistent, while we also handle asynchronous writes at the data store.

Calvin [19] is a fault-tolerant replicated transactional system that provides transactions across multiple data partitions. In contrast to our approach, their transaction management and data store are tightly coupled and build a holistic system, while our approach keeps the data store backend nearly unchanged. Calvin logs the history of transactional input (that is, logical logging instead of physical logging). If there is a failure, the input can be replayed during re-

covery. Different checkpointing techniques are implemented to limit the number of transactions that are re-executed. Checkpointing, however, has considerable performance implications during normal processing.

Sinfonia [1] provides serializable *minitransactions* for accessing the data that is distributed on a set of memory nodes. Minitransactions are executed in the first phase of the two-phase commit protocol (2PC) at the memory nodes. The coordinator is the client and participants are the memory nodes. In contrast to traditional 2PC, the coordinator does not log any information and failures of the coordinator do not block the system. However, a crashed participant can block progress. Additionally, a dedicated recovery coordinator deals with the recovery of transactions that are in-doubt based on participant votes. Memory nodes can be replicated to avoid the blocking behavior, using primary-copy replication during the first phase of 2PC. Our approach targets regular transactions. It does not resort to 2PC, thereby avoiding blocking situations. Recovery only uses the information in the log and does not contact data nodes to decide on the outcome of transactions. Data is kept in HBase/HDFS, which eventually provides data durability without adding extra latency to transaction execution.

Omid is similar to our system in that it implements transactions on top of HBase [11, 21]. According to [11], Omid updates the data in HBase as part of the transaction execution, whereas our proposal is based on the deferred-update model (changes are only applied to HBase after the transaction commits). Omid uses a distributed logging service, BookKeeper[7], for write-ahead logging. Recovery is not described.

G-Store [8] provides transactions over partitioned groups of keys on top of a key-value store. Groups are dynamically created. One of the keys in a group is the leader, which grants read/write access to the group's keys. G-Store uses write-ahead logging and flushes changes asynchronously to the data store. All the information related to groups is also logged, and recovery deals also with in-progress creation and deletion of groups at the time the failure occurred. However, recovery itself is not discussed. No checkpointing mechanism is described that would show how to limit recovery costs.

ElasTraS [7] provides an elastic key-value data store where transactions execute within a single data partition (static partitions), and partitions can be migrated online from one server to another. ElasTraS uses write-ahead logging for durability and stores the log in HDFS. Once data is persisted in the key-value store, the log is updated in order to enable truncation. However, no further analysis is provided regarding the logging and recovery processes.

In [4], a database is built on top of Amazon S3, analyzing how various Amazon services can be used for database purposes. The approach presents a global solution where transaction management is tightly integrated with the other components. Amazon's Simple Queuing System is used to store log records. However, failure and recovery are not described or analyzed in detail.

Deuteronomy [14] supports transactions over arbitrary data. It decouples transaction processing from data storage, as already done in [15, 16]. Just as in

[7] http://zookeeper.apache.org/bookkeeper/

our approach logging is done at the transaction manager, which has to coordinate with the data stores. In the case of Deuteronomy, the transaction manager tells the data store when to persist data items. In contrast, we let the clients and servers tell the recovery manager what has been flushed and persisted, respectively. In Deuteronomy, if the transaction manager fails, a new transaction manager is initialized and performs recovery using the log. Recovery may also need to undo updates, which is not necessary in our approach since we only flush after commit.

In recent years, geo-replicated transactions have received attention in order to achieve consistency across geographically distributed data stores [6, 12, 18]. The idea is to remain available through wide-area replication even if individual data centers go down. The main focus is coordination through 2PC and Paxos [13]. In such a context, recovery costs are less important because availability is maintained through replication, and the costs of persistence play a lesser role as wide-area coordination is the main factor.

Hyder [2] is a log-structured multi-version key-value database shared by many servers where the log not only guarantees durability but also is used to update the actual server state. The idea is that server caches are a (partial) copy of the database. Transactions write their changes to the log, and servers run a meld algorithm [3] that traverses the log to keep the cache copy up-to-date while at the same time performing concurrency control.

6 Conclusion

In this paper, we present a logging and recovery infrastructure where a modular transaction manager is combined with a distributed key-value store. Transaction write-sets are persisted to the transaction manager's recovery log at commit time and then flushed asynchronously to the key-value store and then eventually persisted to the distributed filesystem. Transaction progress is tracked at the key-value clients and servers. Light-weight checkpointing is performed in order to limit the amount of recovery that has to be performed at recovery time.

7 Acknowledgments

This research has been partially funded by the Ministère de l'Enseignement supérieur, de la Recherche, de la Science et de la Technologie of Quebec and by the European Commission under project CumuloNimbo (FP7-257993), the Madrid Regional Council (CAM), FSE and FEDER under project CLOUDS (S2009TIC-1692), and the Spanish Research Agency MICINN under project CloudStorm (TIN2010-19077).

References

1. Aguilera, M.K., Merchant, A., Shah, M.A., Veitch, A.C., Karamanolis, C.T.: Sinfonia: A new paradigm for building scalable distributed systems. ACM Trans. Comput. Syst. **27**(3) (2009)

2. Bernstein, P.A., Reid, C.W., Das, S.: Hyder - a transactional record manager for shared flash. In: CIDR. (2011) 9–20
3. Bernstein, P.A., Reid, C.W., Wu, M., Yuan, X.: Optimistic concurrency control by melding trees. PVLDB **4**(11) (2011) 944–955
4. Brantner, M., Florescu, D., Graf, D.A., Kossmann, D., Kraska, T.: Building a database on s3. In: SIGMOD Conference. (2008) 251–264
5. Chang, F. et al.: Bigtable: a distributed storage system for structured data. In: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7. OSDI '06, Berkeley, CA, USA, USENIX Association (2006) 15–15
6. Corbett, J.C. et al.: Spanner: Google's globally-distributed database. In: Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation. OSDI'12, Berkeley, CA, USA, USENIX Association (2012) 251–264
7. Das, S., Agrawal, D., El Abbadi, A.: Elastras: an elastic transactional data store in the cloud. In: Proceedings of the 2009 conference on Hot topics in cloud computing. HotCloud'09, Berkeley, CA, USA, USENIX Association (2009)
8. Das, S., Agrawal, D., El Abbadi, A.: G-store: a scalable data store for transactional multi key access in the cloud. In: Proceedings of the 1st ACM symposium on Cloud computing. SoCC '10, New York, NY, USA, ACM (2010) 163–174
9. Ghemawat, S., Gobioff, H., Leung, S.T.: The google file system. In: SOSP. (2003) 29–43
10. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. 1st edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1992)
11. Junqueira, F., Reed, B., Yabandeh, M.: Lock-free transactional support for large-scale storage systems. In: DSN Workshops. (2011) 176–181
12. Kraska, T., Pang, G., Franklin, M.J., Madden, S., Fekete, A.: Mdcc: multi-data center consistency. In: EuroSys. (2013) 113–126
13. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2) (1998) 133–169
14. Levandoski, J.J., Lomet, D.B., Mokbel, M.F., Zhao, K.: Deuteronomy: Transaction support for cloud data. In: CIDR. (2011) 123–133
15. Lomet, D.B., Fekete, A., Weikum, G., Zwilling, M.J.: Unbundling transaction services in the cloud. In: CIDR. (2009)
16. Lomet, D.B., Mokbel, M.F.: Locking key ranges with unbundled transaction services. PVLDB **2**(1) (2009) 265–276
17. Peng, D., Dabek, F.: Large-scale incremental processing using distributed transactions and notifications. In: OSDI. (2010) 251–264
18. Sovran, Y., Power, R., Aguilera, M.K., Li, J.: Transactional storage for geo-replicated systems. In: SOSP. (2011) 385–400
19. Thomson, A., Diamond, T., Weng, S.C., Ren, K., Shao, P., Abadi, D.J.: Calvin: fast distributed transactions for partitioned database systems. In: SIGMOD Conference. (2012) 1–12
20. Wei, Z., Pierre, G., Chi, C.H.: Cloudtps: Scalable transactions for web applications in the cloud. IEEE T. Services Computing **5**(4) (2012) 525–539
21. Yabandeh, M., Gómez Ferro, D.: A critique of snapshot isolation. In: Proceedings of the 7th ACM european conference on Computer Systems. EuroSys '12, New York, NY, USA, ACM (2012) 155–168