Reconciling Transactional and Non-Transactional Operations in Distributed Key-Value Stores

Edward Bortnikov  
Yahoo Labs  
Haifa, Israel  
ebortnik@yahoo-inc.com

Eshcar Hillel  
Yahoo Labs  
Haifa, Israel  
eshcar@yahoo-inc.com

Artyom Sharov∗  
Technion, CS  
Haifa, Israel  
sharov@cs.technion.ac.il

ABSTRACT

NoSQL databases were initially designed to provide extreme scalability and availability for Internet applications, often at the expense of data consistency. The recent generation of Web-scale databases fills this gap, by offering transaction support. However, transaction processing implies a significant performance overhead on online applications that only require atomic reads and writes. The state-of-the-art solutions are either static separation of the data accessed by transaction-enabled and native applications, or complete “transactification” of the latter, which are both inadequate.

We present a scalable transaction processor, Mediator, that enjoys the best of both worlds. It preserves the latencies of atomic reads and writes, without compromising data safety. We introduce a lightweight synchronization protocol that enables conflict resolution between transactions and native operations that share data in a distributed database. We evaluate Mediator’s implementation on top of the HBase key-value store on a large-scale testbed, and show that it substantially outperforms the traditional approach on a vast majority of mixed workloads. In particular, Mediator achieves a significantly larger throughput for all workloads in which the fraction of native operations exceeds 50%.

Categories and Subject Descriptors

H.2.4 [Database Management]: Systems—Transaction processing, Distributed databases

General Terms

Algorithms, Reliability, Performance

1. INTRODUCTION

Modern Internet applications employ data stores that scale to the entire population of online users. For example, personalized content recommendation services require maintaining profiles for hundreds of millions of unique users. Traditional SQL data management systems cannot scale up with these requirements, leading to a new generation of not-only-SQL, or NoSQL, databases—e.g., Google’s Bigtable [8], Apache Hadoop HBase1, etc. These technologies have been designed for extreme simplicity (key-value store API), scalability (data partitioning across thousands of machines), and reliability (redundant storage). In parallel, the proliferation of affordable high-end hardware (multi-core CPU’s, inexpensive RAM, SSD storage) enabled building NoSQL databases capable of serving data at memory speeds [3, 12, 23]. Historically, NoSQL databases only allowed atomic reads and writes of individual items. More recent systems (e.g., Google’s Percolator [20] and HBase’s Omid2) introduce transaction processing [15] for complex applications that require ACID semantics while accessing multiple items. NoSQL transaction processors implement consistency models extensively studied by the database community [18, 13]. They have been shown to scale well with the database size.

Transaction processing does not come for free. Every transaction incurs latency penalties at its begin and commit boundaries. In throughput-oriented applications that perform long transactions latency is not of big concern, and indeed the overhead is minor [25]. However, it is well-pronounced in online, interactive settings, which are the main focus of this work. The faster the underlying database is, the larger the toll. Table 1 exemplifies the impact of “transactifying” HBase reads and writes by Omid in a high-speed environment (fully evaluated in Section 5). Latencies start growing as every operation is framed as a transaction (column 2). They double as part of the traffic is batched in short and long transactions (columns 3 and 4, respectively).

We would like to avoid automatically converting the atomic database operations into transactions. Unfortunately, running them side by side with transactional traffic on shared data without any coordination is error-prone. Consider, for

<table>
<thead>
<tr>
<th></th>
<th>native</th>
<th>trans</th>
<th>trans</th>
<th>trans</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+ none</td>
<td>+ short</td>
<td>+ long</td>
<td></td>
</tr>
<tr>
<td>Read</td>
<td>3.9</td>
<td>5.2</td>
<td>6.0</td>
<td>9.2</td>
</tr>
<tr>
<td>Write</td>
<td>8.3</td>
<td>9.5</td>
<td>10.3</td>
<td>14.0</td>
</tr>
</tbody>
</table>

Table 1: The impact of transactification (trans) on HBase native operations latencies (ms). Transaction processing adds a surplus that grows with the length of transactions executed in the background (none, short, long).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

∗Research done while interning with Yahoo Labs, Haifa.

1http://hbase.apache.org/
2https://github.com/yahoo/omid
example, an imaginary social application, in which user statuses can be either directly posted by the users, or speculated by the system, based on a variety of signals (status history, location, time of day, sensor data, friends’ posts, etc.). In the latter case, the system updates the status unless the user has recently posted a new one. Therefore, it performs a transaction that (1) reads the user’s status, and possibly some other data, (2) does some computation, and (3) writes the status back. In contrast, a human-originated status update is blind – it must complete in the real time, and needs not be transactional. In a non-coordinated implementation, the transaction is not aware of this update, and may overwrite it with a stale speculated value.

An additional problem with the non-coordinated design is exposure to uncommitted data. The transaction processing layer prevents transactions from reading each other intermediate modifications that may eventually roll-back [15]. However, in a heterogeneous environment, native reads can retrieve transaction’s dirty writes.

Our goal is to preserve the original performance of native operations while maintaining the familiar consistency guarantees for them as well as for transactions. This challenge is amplified in distributed databases, in which the data is partitioned among multiple servers. In this context, any solution must take care not to impede the datapath scalability, by introducing minimum synchronization.

We present Mediator — Mixed Database Access Transaction Oracle — a scalable transaction processor that guarantees data consistency in the presence of native operations. To the best of our knowledge, this problem has not been addressed by the database community before.

We establish a consistency model for systems supporting transactions and native operations. Namely, we extend the popular serializability [18] and snapshot isolation (SI) [13] models to accommodate the native traffic semantics. The extension is not straightforward since native operations are not captured as transactions, and the consistency requirements are relaxed for them. The formal definition of the models and Mediator’s correctness proofs are deferred to the full version of this paper [6].

Similarly to earlier work [24, 25], Mediator exploits multi-version concurrency control at the database layer to implement its consistency model. The unique challenge is installing a logical order between transactions, which are ordered by a centralized logical clock, and native operations, accessing multiple independent servers. We introduce temporal fencing – a novel protocol that loosely synchronizes the servers’ local clocks with the global clock. The algorithm trades performance optimization of native operations for an extra overhead imposed on transactions.

We implement a working prototype of Mediator on top of HBase, and extensively evaluate it on a distributed testbed. We study Mediator’s performance tradeoffs by comparing it to an Omid-powered system that automatically converts native operations into transactions. The results emphasize the performance impact incurred to native traffic by Omid. More importantly, we show that Mediator’s overall system performance is superior for a vast majority of the considered mixed traffic patterns. In particular, its throughput is higher for all workloads that contain at least 50% native operations.

The rest of this paper is structured as follows. Section 2 sketches Mediator’s system architecture. Section 3 informally presents mixed traffic semantics, and Section 4 describes the algorithms implementing two consistency models. Section 5 depicts and analyzes the evaluation results. Finally, we survey related work in Section 6, and conclude with Section 7.

2. SYSTEM OVERVIEW

Mediator operates on top of a distributed key-value store with a get/put API that provides a read/write access to data items identified by unique keys. For scalability, the data can be partitioned over multiple nodes. In this context, all accesses to a given item are served by a single node (database server). The get/put API is called native, in contrast with Mediator’s API that is transactional. The database serves both types of traffic. Native clients are unaware of concurrent transactions.

Mediator assumes multi-version concurrency control [15] in the underlying database. Namely, every update creates a new version of the data item, and multiple versions can be accessed in parallel. The transaction processor maintains a global logical clock to timestamp all transactional writes. The execution is optimistic – i.e., each transaction runs unobstructed until commit, whenupon consistency is enforced. At that point, semantic conflicts are detected through version timestamps, and the compromised transactions are aborted.

Mediator shares many design principles with Omid [25]. Similarly to Omid, Mediator employs a standalone transaction status oracle service (TSO), which maintains the clock and tracks the state required for guaranteeing the safety properties. Transactional clients use this context to read the correct data versions and to timestamp their writes. A transaction communicates with the TSO twice – upon begin, to retrieve the required state, and upon commit, to resolve the conflicts with the concurrent transactions. The key for scalability is keeping the TSO separate from the datapath.

The TSO is highly optimized, to prevent it from becoming the system’s bottleneck. A transaction starts getting tracked only once it issues a commit request. A client communicates to the oracle the set of keys accessed by the transaction. The Mediator TSO stores it in a compressed Bloom filter [5] form, hence each transaction’s footprint is fixed and small. Mediator adopts Omid’s optimization of replicating the oracle’s state to the client upon transaction begin, to enable local decision-making [25]. For clients with persistent TSO connections, this replication is incremental and efficient.

In multi-version databases, concurrent transactions are protected from reading non-committed writes by creating new versions with timestamps that are beyond the read horizon. This approach is non-applicable in our setting, since native reads that simply retrieve the latest data versions must be protected. Instead, Mediator clients buffer the updates locally, and write them back upon commit. Prior to updating the database, the client atomically appends its modification set to the write-ahead log (WAL). This is in contrast to other transaction processor implementations [20, 25] writing eagerly to the database. These implementations exploit the durability of database updates and therefore avoid managing a separate log. Mediator’s performance for the transactional part of the traffic is therefore a-priori inferior to eager-write systems. Section 4 describes the optimizations that target this gap.

Since Mediator’s writes are deferred until commit, it never needs to roll back aborted transactions. However, a failure...
of either a client or a database server in the middle of a distributed write-back can leave transactions that are committed in the log-incomplete. While the algorithm guarantees that subsequent transactions always observe a consistent database state, some of them might get blocked and eventually abort due to dependencies on incomplete transactions. To guarantee progress, the TSO helps uncompleted transactions finish their database update. It periodically initiates a helper process that locates their commit records in the log, and replays them in an idempotent way [15]. Transactions that failed to log their changes prior to the helper’s execution are (possibly spuriously) aborted.

Oracle’s failures are handled similarly. Upon recovery, the TSO replays the log, and aborts all transactions that initiated a commit before the crash but failed to log their changes. Hence, the volatile state that the TSO has maintained for them prior to the failure needs not to be restored. Before becoming operational, the TSO sets its clock sufficiently ahead of the committed transactions and the local clocks of all live database servers, to guarantee correctness. Obviously, until the recovery completes, no transactions can commit, however, native operations execute regularly. The rest of the paper focuses on non-failure scenarios.

Figure 1 depicts Mediator’s architecture, and highlights the component API’s. We implement Mediator on top of two open source products – a multi-versioned key-value store (HBase) and a shared log service (Bookkeeper3 [17]). Both scale horizontally across multiple machines.

### 3. MIXED TRAFFIC SEMANTICS

In classical (transaction-only) implementations the versions of an item are ordered according to the temporal sequence of the transactions that created these versions. Informally, in serializable systems all transactions appear to execute sequentially. The weaker model of snapshot isolation decouples the consistency of the gets and the puts of a transaction. That is, all reads within a transaction see a consistent view of the database, as though the transaction operates on a private snapshot of the database taken before its first read. In addition, concurrent transactions are not allowed to modify the same data. This ensures that among two transactions that produced a version of an item, one commits before the other starts.

Mixed-traffic implementations need to consider how to pin native operations within this order. A straightforward semantic for mixed traffic captures native operations as single operation transactions. This surely guarantees no updates are lost and no reads are dirty. However, it may introduce inefficient implementations that incur unnecessary overhead.

The detour we suggest from converting native operations to transactions is twofold. Similar to traditional NoSQL, (i) native operations cannot abort, and (ii) no guarantees are provided on the order of native gets in the serialization (with respect to each other). That is, a process that sequentially retrieves two different items being updated in parallel by some transaction might observe an older version in the second read.

We revisit our web application example to demonstrate the main theoretical challenges of such systems. A backend transaction \( tx \) reads the status \( s_0 \) of a user, computes a refined status \( s_2 \) and tries to update the user’s status record. It should succeed (commit) only if the user is not writing a new status \( s_1 \) at the same time. Furthermore, should the user write a new status before \( tx \) commits, no friend of the user (applying a transaction or native operations) should see status \( s_2 \).

Figure 2a depicts an execution of this scenario. Assume the initial timestamps of all items are 0. Transaction \( tx \) starts (at \( t_0 \)), reads item \( z \) (the user status) and returns \( s_0 \), writes \( s_2 \) to \( z \), and finally commits. The user entry is updated with \( s_2 \) only after the transaction is guaranteed to commit (otherwise it might be visible to the user’s friends). While \( tx \) is committing, after it is logged as committed and just before it writes the new value to \( z \) (with timestamp \( t_1 \)), a concurrent native put, \( op \), by the user writes a new status \( s_1 \) to \( z \).

Due to data partitioning, there is no single point of decision, and the timestamps assigned to \( op \) is determined by the data server accommodating the item (the user record). A possible naïve approach is to associate \( op \) with the item’s previous timestamp plus some increment. This is, however, insufficient for correctness. For example, if \( op \) is assigned with an arbitrary (positive) timestamp \( t \), \( t < t_1 \), a friend of the user reading the status after the transaction completes sees status \( s_2 \) and \( s_1 \) is “lost”.

It is also desirable to avoid trivial solutions “separating” native operations and transactions in time. That is, to assign \( op \) with a small timestamp \( t \), \( t < t_0 \) such that it is “lost in the past”, or a very big timestamp \( t \gg t_1 \) such that it is “lost in the future” and never read by any transaction. To enforce this restriction, the semantics require the serialization of all accesses of transactional and non-transactional operations to the same item by the same process to be in the same order as executed by the process. This property is denoted per-process item order.

To conclude, mixed traffic semantics (1) require transactions to satisfy some consistency model, be it serializability, snapshot isolation or any other model, (2) require native op-

---

Figure 1: Mediator architecture. The transaction manager (Mediator client) employs three backend services - the database, the log, and the transaction status oracle (TSO). The database serves native clients directly, and provides Mediator clients with a backdoor API.
versus a native put

read-only operations, a write-only (its write set is empty), and a

Figure 2: Execution example: transaction \( tz \) (process \( C_1 \)) versus a native put \( op \) (process \( C_2 \)).

4. MEDIANOR ALGORITHM

We present the algorithm focusing on SI consistency; Section 4.5 elaborates on the adjustments required to support serializability. In what follows, \textit{write set} refers to data items written by a transaction and \textit{read set} to items it reads. A \textit{read-write} transaction performs both get and put operations, a \textit{read-only} transaction performs only get operations (its write set is empty), and a \textit{write-only} transaction performs only put operations (its read set is empty). A \textit{put} transaction is a write-only transaction writing to a single item.

4.1 Temporal Fences

To accommodate mixed traffic, Mediator embeds a standard centralized transaction manager, with an original mechanism to pin the versions produced by native puts in the total order without compromising the transactions safety. Native writes are stamped by each data server independently, whereas transactional writes inherit the timestamps issued centrally by the TSO. Combining the two mechanisms—one centralized and the other distributed—requires care. Each database server maintains a local clock. This clock is used to stamp native puts, which in turn increase it in increments of \( \delta \). Each transaction is associated with two values of the global TSO clock. Transaction’s reads are associated with its start timestamp, and writes with its commit timestamp. Upon each database access, the transaction synchronizes the local clock with one of these values—i.e., the server’s clock is promoted to be (at least) this value. This value then serves as a fence—no subsequent native operation to this server is assigned with timestamp lower than the fence. Specifically, the \( \delta \) increment defer the timing of subsequent native put operations to this server beyond the fence value. The global clock assigning transactional timestamps grows by \( \Delta \) upon each timestamp request. To avoid the trivial time-separation solution we set \( \Delta \gg \delta \). This ensures native puts that are executed within the epoch marked by two temporal fences, are serialized within this epoch.

In the execution example, when \( tz \) accesses the data server for the first time, it assigns its local clock to \( t_0 \) (first fence), thereby guaranteeing that \( op \)'s timestamp is greater than \( t_0 \), and it is not lost in the past. Prior to writing a new value to \( z \), \( tz \) verifies no concurrent put (specifically, a native one) has written to \( z \). Upon this conflict testing, \( tz \) promotes the local clock to \( t_1 \), \( t_1 \geq t_0 + \Delta \) (second fence). Therefore, the write conflict validation is safe; the set of native writes between \( t_0 \) and \( t_1 \) is sealed. In the scenario depicted in Figure 2b, \( op \) is assigned with timestamp \( t \), \( t_0 + \delta = t < t_1 \), \( tz \) identifies the conflict and aborts. In the scenario depicted in Figure 2c, \( op \)'s timestamp is \( t_1 + \delta = t \), hence there is no conflict and \( tz \) commits. As \( \delta \ll \Delta \) \( op \) is not lost in the future, and is visible to subsequent transactions.

With this in mind, Mediator’s algorithm is simple and intuitive. The begin timestamp of a transaction is a temporal fence. A get returns the latest version of the item prior to this timestamp. Write accesses to the data server by transactions are deferred to commit, and a put simply privately records the key-value pair. Upon commit of read-only transactions, no further action is required since the snapshot property holds: a \textit{no-commit} optimization. A put transaction commits locally, by applying a native put thereby avoiding the commit overhead: a \textit{local-commit} optimization. Other transactions apply a \textit{two-phase commit}. The first phase (conflict testing) consists of a centralized part and a distributed part. The former generates a commit timestamp and tests for inter-transaction conflicts. The latter installs the commit timestamp as a temporal fence at the data servers accommodating the write set, and performs the \(\delta\) optimization. The second phase (write-back) logs the new values for durability, and ultimately stores them in the database, making the changes visible to other transactions and native operations.

Next, we describe the implementation details, including the pseudo-code. The code is structured in a way that can be easily adapted to support serializability.

4.2 Transaction Manager Implementation

The key parts of Mediator’s API implementation appear in Algorithm 1. For simplicity; we assume that (1) a transaction reads and writes any item at most once, and (2) if a transaction writes an item, it does not read it afterwards.

A transaction is represented by a descriptor, which holds the start and commit timestamps, as well as the write set

\footnote{These assumptions are not restrictive – modeling these (redundant) operations is possible but obscures the presentation.}
Algorithm 1 Mediator API implementation - get and commit (snapshot isolation)

1: function txGet(Timestamp ts, Key key)
2:   tx ← getTx(ts)
3:   if retryGet times do
4:     (val, ts_val) ← DB.txGet(key, ts)  ⊲ sync DB with ts
5:     if ts_val ≥ latestCommited(key, ts) then
6:       return val  ⊲ unable to read latest value
7:   end if
8:   return 13:
9: function txCommit(Timestamp ts)
10:  tx ← getTx(ts)
11:  if readOnly(tx) then
12:    return 11:
13:  if singleWriteOnly(tx) then
14:    (key, val, ts) ← tx.writeValueSet.first()
15:    DB.put(key, val)  ⊲ local write
16:    return 11:
17:  ok ← tryCommit(ts, ww)
18:  if ok then
19:    writeVals(tx)
20: function tryCommit(Timestamp ts, ConflictType type)
21:  tx ← getTx(ts)  ⊲ check txs conflicts against TSO
22:  tx.ts, ts ← TSO.txTryCommit(tx, type)
23:  if tx.ts, ts = ∅ then
24:    txAbort(tx)
25:  return FALSE  ⊲ check natives conflicts against DB
26:  if type = ww then
27:    confSet ← tx.writeSet  ⊲ ww conflicts
28:  if type = rw then
29:    confSet ← tx.readSet  ⊲ rw conflicts
30:  ok ← DB.txCheckConflict(confSet, tx.ts)
31:  if ok then
32:    txAbort(tx)
33:    TSO.txAbort(tx)
34:  return FALSE
35: return TRUE
36: function writeVals(Transaction tx)
37: LOG.append(tx.writeSet)  ⊲ write ahead logging
38: DB.txPut(tx.writeSet, ts)
39: TSO.txCommit(tx)

(set of key-value-timestamp tuples). The latter is used to identify conflicts upon commit. The implementation supporting serializability also needs to maintain the read set.

A transaction begins by retrieving a unique start timestamp from the TSO. Similar to Omid [25], the incremental changes of the status oracle's state are piggybacked on the response to facilitate local decisions.

A transactional get (txGet) reads from the database the latest value preceding the start timestamp. The timestamp of the expected version is registered in the local replica of the TSO's state, however the value itself might not be stored in the data server yet. A failure to retrieve the correct version triggers a sequence of attempts to re-read this version, and ultimately an abort in case they all fail.

A transactional put adds the key-value pair to the write set stamped with the start timestamp. This timestamp is used to check for conflicts upon commit.

Upon commit (txCommit) of a read-only transaction no further action is required. A put transaction commits locally, by performing the respective native put operation.

Algorithm 2 TSO methods (snapshot isolation)

struct TxEntry {
  Timestamp ts,
  Status status // {ACTIVE, COMMITTED, ABORTED}
  BloomFilter writeBF
}
class Ring {
  Timestamp ts_min // earliest timestamp
  TxEntry head
  TxEntry tail
}

51: function txTryCommit(Transaction tx, ConflictType type)
52:  if ts_min > tx.ts then
53:    return ⊥  ⊲ too long a tx - abort
54:    writeKeys ← getKeys(tx.writeSet)
55:    new ← initTxEntry(writeKeys)
56:  if type = ww then
57:    confSet ← tx.writeSet  ⊲ ww conflicts
58:  if type = rw then
59:    confSet ← tx.readSet  ⊲ rw conflicts
60:  ts ← checkAndAppend(confSet, new)
61:  return ts  ⊲ timestamp or ⊥
62: function checkAndAppend(Set(KeyValVersion) confSet, TxEntry new)  ⊲ atomic
63:  new.ts ← getNextTimestamp()  ⊲ add Δ
64:  for current = tail → head do
65:    for item ∈ confSet do
66:      if current.status < item.ts then break
67:      if current.status ≠ aborted then
68:        if isMember(item, current.writeBF) then
69:          return ⊥  ⊲ conflict - abort
70:        append(new, tail)  ⊲ append to tail
71:        return new.tail  ⊲ serialization succeeded

Other transactions invoke txTryCommit at the TSO and txCheckConflict at the database, to verify transaction-transaction and transaction-native conflicts, respectively (first phase of the two-phase commit). Note that the SI implementation checks for write-write (ww) conflicts. Then (second phase), the transaction dumps its write set to the log and to the database.

4.3 Transaction Status Oracle Implementation

The TSO maintains an ordered circular buffer (ring) of transaction entries. The ring's entries describe only transactions that invoked txTryCommit, saving space and redundant processing. A transaction commits by enqueuing an entry into the ring, therefore its position in the ring is its explicit serialization with respect to other transactions.

The TSO's data structures appear in Algorithm 2. A ring entry holds the transaction's commit timestamp, its status, and the write set's keys encoded as Bloom filters [5] for compactness. The status is initially ACTIVE, indicating that the transaction is serialized with respect to other transactions, but still has to check conflicts with native puts and write to the database. Upon a commit or abort notification, the status is updated accordingly. Bloom filters help to efficiently test set membership – in particular, compute the intersection between transaction write sets to detect conflicts. The flip side of using them is manifested in false intersections, which yield spurious aborts.

txTryCommit detects inter-transaction conflicts. A transaction acquires a commit timestamp, and traverses the ring
Algorithm 3 DB methods

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>81:</td>
<td>function <code>GET(Key key)</code> ▷ atomic</td>
</tr>
<tr>
<td>82:</td>
<td>return <code>lastVersion(key)</code></td>
</tr>
<tr>
<td>83:</td>
<td>function <code>PUT(Key key, Val val)</code> ▷ atomic</td>
</tr>
<tr>
<td>84:</td>
<td><code>clock ← clock + δ</code></td>
</tr>
<tr>
<td>85:</td>
<td><code>put(key, val, clock)</code></td>
</tr>
<tr>
<td>86:</td>
<td>function <code>TXGET(Key key, Timestamp ts)</code></td>
</tr>
<tr>
<td>87:</td>
<td><code>sync(ts)</code></td>
</tr>
<tr>
<td>88:</td>
<td>return <code>lastVersionBefore(key, ts)</code></td>
</tr>
<tr>
<td>89:</td>
<td>function <code>TXCHECKCONFLICT(Set(KeyValVersion) items, Timestamp ts)</code></td>
</tr>
<tr>
<td>90:</td>
<td><code>sync(ts)</code></td>
</tr>
<tr>
<td>91:</td>
<td>for all <code>item ∈ items</code> do</td>
</tr>
<tr>
<td>92:</td>
<td><code>(val, ts_min) ← lastVersionBefore(item.key, ts)</code></td>
</tr>
<tr>
<td>93:</td>
<td>if <code>item.ts &lt; ts_min</code> then</td>
</tr>
<tr>
<td>94:</td>
<td>return <code>FALSE</code> ▷ conflict - not ok</td>
</tr>
<tr>
<td>95:</td>
<td>return <code>TRUE</code> ▷ no conflict - ok</td>
</tr>
<tr>
<td>96:</td>
<td>function <code>SYNC(Timestamp ts)</code> ▷ atomic</td>
</tr>
<tr>
<td>97:</td>
<td><code>clock ← max{ts, clock}</code> ▷ temporal fence</td>
</tr>
</tbody>
</table>

from tail to head validating the write set with the preceding non-aborted transactions. Finally, a new entry is appended to the ring, and the commit timestamp is returned. Long-running transactions that started before `ts_min`–the minimum timestamp from which the ring maintains transactional history–abort.

The ring is periodically garbage-collected – complete transaction entries that do not overlap with active transactions are deleted. Transactions for which no completion notification has been received remain in the ring until their final status is discovered by the helper process that runs periodically in the background (discussed in Section 2).

The algorithm’s correctness depends on the assumption that native put timestamps never exceed the next transaction timestamp. For all practical purposes, this is achieved by setting `Δ ≫ δ` (e.g., `Δ = 2^20` and `δ = 1` for a 64-bit value clock). To maintain the invariant, even when no transactional traffic arrives for a very long time, the TSO periodically increments the global clock by `Δ`.

4.4 Database Support

The database code adjustments for Mediator are modest. They summarize to a new policy for managing the server’s local clock. Local clocks synchronize with the global clock upon transactional accesses, and incremented upon native puts. Algorithm 3 depicts the implementation.

A native get simply retrieves the latest version of the data item. A native put atomically increments the clock and writes the new timestamped version to the database. A transactional get synchronizes the clock with the transaction’s timestamp, which becomes a temporal fence, and returns the latest version prior to this timestamp. A transactional put simply invokes the timestamp-based put API. The `TXCHECKCONFLICT` method (invoked upon commit) tests whether a set of timestamped key-value tuples has been modified by native puts prior to timestamp `ts`. The server’s clock is atomically synchronized with `ts`, which becomes a temporal fence.

4.5 Supporting Serializability

We follow the work by Cahill et al. [7] to adapt our SI algorithm to serializability. Similar to it, we exploit the observations from [13], which identify distinctive conflict patterns (dangerous structures) in every non-serializable execution.

In this context, a serialization graph is one in which nodes represent transactions, and edges represent conflicts between them. With mixed traffic, an edge can connect a transaction with a conflicting native operation. A read-write edge implies that a put overrides the value read by the other transaction. The serialization graph of any non-serializable SI execution contains a cycle with two adjacent read-write edges, each connecting two concurrent transactions [13]. Mediator’s adapted protocol (Algorithm 4) eliminates read-write edges in the graph by aborting the conflicting transaction. This is sufficient for removing “dangerous” structures, although spurious aborts might happen. To minimize the number of aborts, `TXGET` returns the most up-to-date item version, instead of reading the latest version written before the transaction started as in the SI implementation.

Two transactions (or operations) writing to the same item but not having read-write conflicts, can be serialized by the order of their commit timestamps. Therefore, instead of checking write-write conflicts, a transaction checks for read-write conflicts with other transactions and native operations. It verifies that no put operation has written a value to an item the transaction read. That is, upon `TRYCOMMIT` the TSO compares the write sets of transactions in the ring with the read set of the processed transaction, instead of comparing with its write set as in the SI implementation. Similarly, the database-level conflict test verifies the intersection of the transaction’s read set with native puts.

We do not assume any a-priori knowledge on the data set of a transaction. Specifically, read-only transaction are not defined as such in advance. Therefore, `TXGET` operations in read-only transactions also returns the most up-to-date item version. To this end, read-only transactions cannot employ the no-commit optimization since they need to detect read-write conflicts. The local-commit path for put transactions still holds.

Algorithm 4 Adaptation for serializability support

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>101:</td>
<td>function <code>txGET(Timestamp ts, Key key)</code></td>
</tr>
<tr>
<td>102:</td>
<td><code>tx ← getTx(ts)</code></td>
</tr>
<tr>
<td>103:</td>
<td><code>(val, ts_min) ← DB.get(key)</code></td>
</tr>
<tr>
<td>104:</td>
<td><code>tx.addToReadSet(key, val, ts_min)</code></td>
</tr>
<tr>
<td>105:</td>
<td>return <code>val</code></td>
</tr>
<tr>
<td>106:</td>
<td>function <code>txCOMMIT(Timestamp ts)</code></td>
</tr>
<tr>
<td>107:</td>
<td><code>tx ← getTx(ts)</code></td>
</tr>
<tr>
<td>108:</td>
<td>if <code>singleWriteOnly(tx)</code> then</td>
</tr>
<tr>
<td>109:</td>
<td><code>(key, val, ts) ← tx.writeSet.first()</code></td>
</tr>
<tr>
<td>110:</td>
<td><code>DB.put(key, val)</code> ▷ local writing</td>
</tr>
<tr>
<td>111:</td>
<td>return ▷ committed successfully</td>
</tr>
<tr>
<td>112:</td>
<td><code>ok ← tryCommit(ts, rw)</code></td>
</tr>
<tr>
<td>113:</td>
<td>if <code>ok</code> then</td>
</tr>
<tr>
<td>114:</td>
<td>if <code>readOnly(tx)</code> then return</td>
</tr>
<tr>
<td>115:</td>
<td>else <code>writeVal(tx)</code></td>
</tr>
</tbody>
</table>

5. EVALUATION

We evaluate Mediator on a distributed testbed, and assess the system performance (throughput and latency) metrics, as well as the ratio of aborted transactions. The latter is an upper bound on the ratio of false aborts, which captures system’s negative impact on client applications, and is ex-
pected to be low. We experiment with multiple workloads that feature different traffic mixes (varying proportions of get versus puts, transactional versus native operations) and different distributions of transaction size (single-access versus bulk transactions). Mediator’s behavior is explored in the context of snapshot isolation and serializability models.

We compare a system in which transactions are served by Mediator and native traffic is handled by HBase with a system in which native operations are transacted, and all the traffic is served by Omid. For brevity, we call the first system Mediator and the second system Omid.

We start by analyzing the overhead transaction processing imposes on native operations. Following this, we study Mediator’s impact on the overall system throughput. Namely, we explore Mediator’s and Omid’s comfort zones – the workload patterns for which one platform performs significantly better than its counterpart.

5.1 Environment

We utilize a cluster of machines equipped with a 4-core 2.50 GHz Xeon(R) L5420 CPU and 16 GB RAM. All services are implemented in Java, with the JVM using 4 GB heap. Separate nodes are allocated to the database (10), the log (10), the TSO (1), and the clients (20).

The key-value store is HBase, deployed on top of Hadoop’s distributed filesystem, HDFS, with a replication factor of 3. The HBase database (region) servers are co-located with HDFS storage servers (datanodes) for efficiency. The dataset under test holds 200 million records. Each record is 1 KB long, with a 12 bytes long key. That is, the database’s size is approximately 200 GB, each server controlling 10% thereof. The block cache at each server defaults to 40% of the heap.

We are interested in latency-oriented applications and therefore focus on configurations that serve individual operations with low latency. We address workloads with reasonable locality of gets – the keys are drawn from a Zipfian distribution that generates approximately a 90% cache hit rate. The put latencies are insensitive to key distribution since HBase servers employ LSM trees [23] that absorb multiple writes into a memory buffer. To keep the changes to the database layer minimal, we do not try to optimize the HBase overhead by switching off the database’s internal write-ahead logging (which is redundant with Mediator’s log for transactional traffic).

We perform a large set of experiments on a variety of workloads. A single experiment performs 500,000 gets and puts. Each client node concurrently drives the system’s workload through up to 40 concurrent processes of YCSB [10], a popular load generator. The YCSB clients exercise the Mediator, Omid, and HBase API’s, depending on the configuration. In a single experiment, each YCSB instance drives the same traffic pattern, therefore the cumulative workload remains steady over time. In other words, each client generates the same load bandwidth, which splits independently among (1) reads and writes, and (2) transactional and native accesses. We denote the fraction of reads by $\rho$, and the fraction of native operations by $\nu$. In the Omid setting, YCSB transactifies all the original native operations. In both settings, transactional accesses are clustered in transactions of varying sizes, picked uniformly at random from a range $[1, 2, \ldots, n]$, where $n$ is specified by the experiment. We denote this distribution $U_n$. The larger $n$ is, the wider the spectrum of the exercised transaction sizes is – from a single access to a bulk of operations. $U_n$ is a non-realistic workload (singleton transactions workload is meaningless) that we use to demonstrate the local-commit optimization. Table 2 summarizes the notation.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U$</td>
<td>number of concurrent clients</td>
<td>50, 100, . . . , 1200</td>
</tr>
<tr>
<td>$\nu$</td>
<td>ratio of native operations</td>
<td>0, 0.1, . . . , 1</td>
</tr>
<tr>
<td>$\rho$</td>
<td>ratio of get accesses</td>
<td>0, 0.1, . . . , 1</td>
</tr>
<tr>
<td>$U_n$</td>
<td>transaction size distribution</td>
<td>$U_1$ (singleton), (uniform over $[1, 2, \ldots, n]$)</td>
</tr>
</tbody>
</table>

Table 2: Workload parameter notation.

5.2 Numerical Results – Snapshot Isolation

This section studies Mediator’s performance under the snapshot isolation consistency model.

Latency Overhead on Single Operations. We start by motivating the advantage of serving native traffic directly. Our first experiment demonstrates the surplus to the median latency of native HBase operations when the latter are transactified with Omid. We consider three configurations: (1) 100% native traffic, (2) single-operation ($U_1$) transactions with background $U_4$ traffic, and (3) the same with background $U_5$ traffic. The workload is driven by 200 clients. We explore a variety of read ratios ($0 \leq \rho \leq 1$).

Figure 3(a) shows the results. The penalty grows with the fraction of writes and the background transactions’ bulkiness. This is explained as follows. Transactions introduce a fixed communication overhead (round trip upon begin and commit), and the TSO state replication overhead. The TSO state depends on the number of keys updated by individual transactions, i.e., for the same read/write ratio the larger transactions populate a larger state, which translates to a larger replication overhead, and eventually to larger latencies incurred to short transactions. For example, in write-only workloads in which transactified puts run in parallel with $U_5$ transactions, the put latency becomes almost twice as large as that of the native HBase operation.

The next example provides a different perspective on the same phenomenon. We compare the median operation latency and the system throughput for the traffic of 100% single operations ($U_1$), in the following scenarios: (1) native operations, (2) the same, transactified with Omid, and (3) the same, transactified with Mediator. We observe the performance for varying numbers of clients, and draw the throughput-latency curves. All implementations are considered in two settings – a read-dominated workload ($\rho = 0.9$, Figure 3(b)), and a write-intensive workload ($\rho = 0.5$, Figure 3(c)). We see that even without any bulky transactions in the background, Omid and Mediator are inferior to bare-metal HBase. For example, Mediator scales to approximately 35K operations per second (ops) in the read-dominated scenario, and to 28K in the write-intensive one, whereas HBase achieves above 55K ops. These results emphasize the potential of consolidating transactional and non-transactional traffic within the same framework, to avoid the overhead of transactifying the latter.

*The same HBase configuration scores much higher throughputs for bulk 1/O. This setting is not the focus of our experiment.*
loads, in which native operations run side by side with trans-
os. The collection of equilibrium points for a given work-
ltype defines an equilibrium curve. This curve separates Mediator’s and Omid’s comfort zones. The area above it is
more advantageous for U_{\text{mid}} transactions (only part of the workload).

The comparison becomes interesting for truly mixed work-
lloads, in which native operations run side by side with trans-
s of different sizes. Both Omid and Mediator have their strong points. The former is superior for transactional
traffic, since it avoids the WAL overhead (Section 2). The latter is faster for native traffic. In this context, the cumu-
ative throughput (in terms of both transactional and native operations) is a convenient metric for evaluating the over-
all system performance. (Note that in an environment in
which get and put operations are clustered in transactions, the latency of individual operations is not well-defined.)

The following experiment employs 200 clients, and exer-
cises all combinations of read ratios (0 ≤ ρ ≤ 1) and native
access ratios (0 ≤ ν ≤ 1). In this context, for a given read
ratio ρ, an equilibrium point is the smallest ratio of native
operations ν for which Mediator achieves a larger through-
put. The collection of equilibrium points for a given work-
ltype defines an equilibrium curve. This curve separates Mediator’s and Omid’s comfort zones. The area above it is
the fraction of configurations in which Mediator is superior.

Figure 4(a) and Figure 4(b) portray the equilibrium curves
for U_{\text{mid}} and U_{\text{trans}}, respectively. Every data point is depicted with a 10% confidence interval. Mediator outperforms Omid in a vast majority of configurations – in particular, in any read-write mix with ν ≥ 50%. It is consistently more advantageous for U_{\text{trans}} versus U_{\text{mid}} due to a better manifestation of write batching in bulk transactions. For both workloads, Mediator’s dominance is more pronounced for the extreme values of ρ. For example, for ρ = 1, the no-commit optimization applies to all transactions, thus reducing the equilibrium point to zero; for ρ = 0, the local-commit optimization applies to singleton transactions (only part of the workload).

Figure 4(c) zooms in on how a single equilibrium point is
computed. For a given read ratio ρ, Mediator’s throughput
monotonically increases with the fraction of native traffic ν
(which is natural, since the latter has no overhead). On the
contrary, Omid’s throughput does not grow with ν. This
happens because the system wraps every individual access
as a transaction. For non-singleton transactions (U_{\text{mid}} and
U_{\text{trans}}), the overhead grows disproportionately with ν, thus
reducing the total throughput. The crossing of the two curves is an equilibrium point; the vertical bars mark the areas of

**Total Throughput.** We now turn to our main goal –
contrasting Mediator with Omid on a wide variety of mixed
workloads. We study the U_{\text{mid}}, U_{\text{trans}} and U_{\text{mid}} distributions.

The U_{\text{mid}} case is an exception – Mediator is faster than
Omid in all configurations (Figure 3(b) and Figure 3(c) de-

\[ \text{Latency (ms) vs. } \rho, \% \text{ reads} \]

\[ \text{Throughput (x } 10^3 \text{ ops) vs. } \rho, \% \text{ reads} \]

\[ \text{Total Throughput vs. } \rho, \% \text{ native operations} \]

\[ \text{Equilibrium curve for } U_{\text{mid}}. \]

\[ \text{Equilibrium curve for } U_{\text{trans}}. \]

\[ \text{An equilibrium point, } U_{\text{mid}}, \rho = 0.8. \]

\[ \text{Read-dominated } \rho_{\text{mid}} \text{ workload.} \]

\[ \text{Write-intensive } \rho_{\text{mid}} \text{ workload.} \]
but hits a high 1\% in most configurations, but hits a high 1.78% in write-intensive settings. This fraction of aborts can be reduced 10-fold by applying wider Bloom filters, but this entails a slight performance penalty.

Scalability. Finally, we study Mediator’s bottlenecks, to get an insight about its scalability limits. We take a closer look at the $U_t$ traffic pattern ($\rho = 0.5, \nu = 0$), for the number of clients $C$ ranging from 50 to 500. Figure 5 depicts the latency breakdown by the time spent on significant internal API’s. In this context, the datapath calls that happen upon commit (the native conflict check, the WAL, and the database write) account for over 70% of transaction latency, whereas TSO’s API’s consume less than 20%. For very large numbers of clients, this fraction drops below 10%, which demonstrates that the TSO scales better than the database. The begin timestamp retrieval is a non-negligible component. This happens due to the oracle’s state replication that is piggybacked on this request.

The overhead of write-ahead logging might be reduced by employing state-of-the-art shared log services (e.g., Corfu [3] uses specialized hardware, and boasts sub-millisecond latencies for loads similar to those exercised in our experiment). The potential upside of this optimization is approximately 25% reduction of transaction latency.

5.3 Numerical Results – Serializability

We conclude our experimentation by evaluating the overhead required to support transaction serializability. In this setting, the algorithm incurs additional overhead (sending the transaction’s read set to the TSO, in conjunction with the write set), and tests for read-write conflicts instead of write-write conflicts (Section 4.5).

We compare the serializability implementation’s performance with the one for snapshot isolation, by repeating the experiment in Section 5.2, which evaluates Mediator with transaction-only traffic ($\nu = 0$). For read-dominated workloads ($\rho = 0.9$), communication and processing for serializability support incurs a significant overhead – up to 30\% less throughput in similar operating points. For write-intensive traffic ($\rho = 0.5$), the performance gap is negligible. These results resonate well with other performance studies in the database literature [1, 7].

6. RELATED WORK

Transaction processing is a textbook area in database research [24, 15]. It appears in the ANSI SQL standards, as well as in modern NoSQL technologies that took databases to an unprecedented scale (e.g., [20]). The literature defines a wealth of transaction consistency models that capture different perceptions of concurrency control (e.g., [18, 4]). Traditionally, client applications sharing a single database instance agree on a single consistency model (or multiple levels thereof that subsume each other), and pay the required performance toll. We posit that this approach is not necessarily required, i.e., it is possible to accommodate within the same database two incompatible semantics: multi-operation transactional consistency, and atomic read-write consistency appropriate for simple key-value store applications.

The database community has been reasoning about transaction semantics since the late 70’s [18]. Serializability has been widely adopted. The seminal paper by Berenson et al. [4] introduced the snapshot isolation model. The latter is particularly attractive due to its implementations that improve concurrency.

In databases that support range queries, the literature distinguishes between repeatable read (RR) and serializability isolation levels, which subsumes RR, and extends it with a requirement of avoiding phantom reads (returning two different tuple sets for the same key range to two queries running under the same transaction [15]). Should Mediator be extended to support predicate queries, it can use the same transformation technique by Fekete et al. [13] to get a phantom-free serializable implementation.

Early NoSQL databases, e.g., Google Bigtable [8], Yahoo! PNUTS [9] and Apache HBase sacrifice strong consistency for extreme scalability. Their safety guarantee is single-key atomic reads and writes [2]. Google Percolator technology [20] supports multi-operation transactions in Bigtable for incremental maintenance of its search index. Percolator implements snapshot isolation through database locks. Omid, a transaction processor for HBase [25], supports snapshot isolation with a lock-free protocol. Omid also implements serializability [14]. Mediator’s design bears similarity with Omid, however, its algorithm is profoundly different, to capture the new safety properties.

Google Spanner [11] provides distributed transactions across datacenter with a blend of SI (for read-only transactions) and serializability guarantees. Spanner implements lock-free read-only transactions and lock-based read-write transactions. Calvin [22] also addresses globally distributed transactions, albeit in a different way. It replaces locking with deterministic scheduling that orders transactions through a global consensus service. SCOR[e] [19] is a serializable partial replication protocol that guarantees read operations always access a consistent snapshot. It applies a timestamp management scheme to synchronize the nodes handling the
transaction. Neither one of the above provide any consistency guarantees to hybrid workloads targeted by Mediator.

Transactional memory (TM) [16] is a popular approach for alleviating the difficulty of programming concurrent applications for multi-core and multiprocessor systems. TM allows concurrent processes synchronize via in-memory transactions. Our TSO implementation is inspired by RingSTM [21] – an implementation that allows accessing the same items from inside and outside a transaction. RingSTM is not geared for distributed environments, hence our challenges are different.

7. CONCLUSIONS

We presented Mediator – a transaction processing system for Web-scale NoSQL databases. Mediator mitigates the consistency gaps that arise when transactional and native operations are allowed to share the same data in a straightforward way. Mediator protects the safety invariants of both API’s – namely, (1) atomic reads and writes for the native traffic, and (2) snapshot isolation or serializability for the transactional traffic.

Mediator provides weak synchronization between two types of logical clocks: the global clock maintained by the transaction processing service, and the local clocks of multiple independent database servers. This temporal fencing mechanism installs a logical order between native and transactional accesses, despite the fact that the native accesses completely bypass Mediator’s infrastructure. The protocol is well-founded, and also extremely lightweight compared to physical clock synchronization.

Mediator preserves the original performance of native traffic, while incurring minor impact on transactional operations. A large-scale evaluation shows that this design choice strikes a favorable tradeoff. Namely, it demonstrates that performance-wise, Mediator’s approach is superior to automatic transactification of native operations, for a vast majority of our tested workloads. We also show that spurious aborts – the price paid for preserving the best of both worlds – are very infrequent.

8. ACKNOWLEDGMENTS

We thank Daniel Gomez-Ferro for his relentless help with explaining Omid’s design. We thank Flavio Junqueira, Ronny Lempel and Mark Shovman for stimulating discussions.

9. REFERENCES