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 [7], Apache Hadoop HBase[1], 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, 11, 22].

Historically, NoSQL databases only allowed atomic reads and writes of individual items. More recent systems (e.g., Google's Percolator [19] and HBase's Omid[2] [24]) introduce transaction processing [14] for complex applications that require ACID semantics while accessing multiple items. NoSQL transaction processors implement consistency models extensively studied by the database community [17, 12]. 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 [24]. 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).

	native	**trans + none**	**trans + short**	**trans + long**
Read	3.9	5.2	6.0	9.2
Write	8.3	9.5	10.3	14.0

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*).**

We would like to avoid automatically converting the atomic

*Research done while interning with Yahoo Labs, Haifa.

[1] http://hbase.apache.org/

[2] https://github.com/yahoo/omid

database operations into transactions. Unfortunately, running them side by side with transactional traffic on shared data without any coordination is error-prone. Consider, for 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 [14]. 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 formal definition of the consistency model for systems supporting transactions and native operations, and prove Mediator's compliance with it. Namely, we extend the popular *serializability* [17] and *snapshot isolation* (SI) [12] 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.

Similarly to earlier work [23, 24], 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. Rigorous model definition and correctness proofs appear in Section 6. Finally, we survey related work in Section 7, and conclude with Section 8.

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 [14] 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, whereupon 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 [24]. 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 [24]. 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

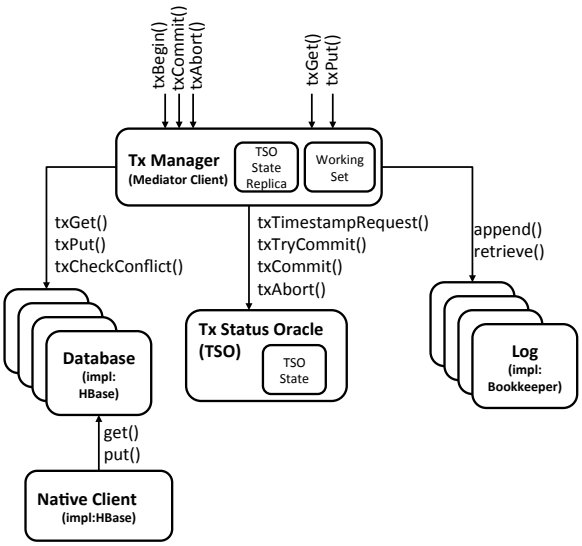


Figure 1: **Mediator architecture. The transaction manager (Mediator client) employs three backend services – the database, the log, and the transaction status oracle (TSO). The database serves native clients directly, and provides Mediator clients with a backdoor API.**

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 [19, 24] 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 [14]. 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 (Bookkeeper[3] [16]). 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.

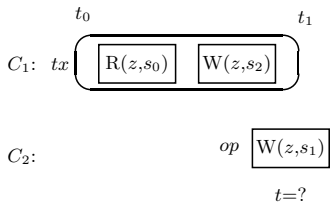
Mixed-traffic implementations need to consider how to pin native operations within this order. A straightforward semantic for mixed traffic captures native operations as single operation transactions. This surely guarantees no updates are lost and no reads are dirty. However, it may introduce inefficient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

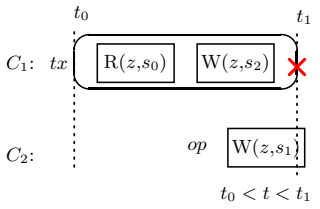
We revisit our web application example to demonstrate the main theoretical challenges of such systems. A backend transaction tx reads the status s_0 of a user, computes a refined status s_2 and tries to update the user's status record. It should succeed (commit) only if the user is not writing a new 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.

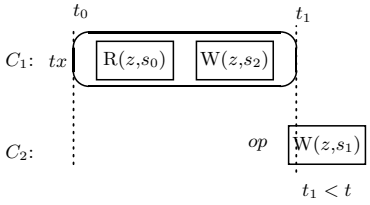
[3] `http://zookeeper.apache.org/bookkeeper/`



(a) If tx commits but op's timestamp is $t < t_1$, then op is "lost".



(b) Temporal fences (depicted as dashed lines) guard safety by aborting tx.



(c) Temporal fences guard safety by serializing op after tx.

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

Due to data partitioning, there is no single point of decision, and the timestamps assigned to op is determined by the data server accomodating 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, $t \ll t_0$ 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 operations not to abort and (3) require operations to satisfy the per-process item order. The formal extension of the classical serializability and snapshot isolation models for mixed traffic appear in Section 6.

Mediator's way to satisfy these semantics—*temporal fences*, effectively partitions the logical timing of native puts into epochs framed by transaction begin and commit operations.

4. MEDIATOR ALGORITHM

We present the algorithm focusing on SI consistency; Section 4.5 elaborates on the adjustments required to support serializability. In what follows, *write set* refers to data items written by a transaction and *read set* to items it reads. A *read-write* transaction performs both get and put operations, a *read-only* transaction performs only get operations (its write set is empty), and a *write-only* transaction performs only put operations (its read set is empty). A *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 δ. 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 δ increment defer the timing of subsequent native put operations to this server beyond the fence value. The global clock assigning transactional timestamps grows by Δ 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 tx accesses the data server for the first time, it assigns its local clock to be 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, tx verifies no concurrent put (specifically, a native one) has written to z. Upon this conflict testing, tx 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, $t_0 + \delta = t < t_1$, tx 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 tx 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 *no-commit* optimization. A put transaction commits locally, by applying a native put thereby avoiding the commit overhead: a *local-commit* optimization.

Other transactions apply a *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 transaction-vs-native conflict test. 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[4].

A transaction is represented by a descriptor, which holds the start and commit timestamps, as well as the write set (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 [24], 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.

Other transactions invoke txTryCommit at the TSO and txCheckConflict at the database, to verify transaction-vs-transaction and transaction-vs-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]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:     for RETRY_GET times do
 4:         ⟨val, ts_val⟩ ← DB.txGet(key,ts)      ▷ sync DB with ts
 5:         if ts_val ≥ latestCommitted(key,ts) then
            ▷ latest timestamp before ts in the TSO state replica
 6:             tx.addToReadSet(key,val,ts_val)
 7:             return val
 8:     txAbort(ts)                          ▷ unable to read latest value

 9: function txCommit(Timestamp ts)
10:     tx ← getTx(ts)
11:     if readOnly(tx) then
12:         return                           ▷ committed successfully
13:     if singleWriteOnly(tx) then
14:         ⟨key, val, ts⟩ ← tx.writeSet.first()
15:         DB.put(key, val)                                ▷ local write
16:         return                           ▷ committed successfully
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_c ← TSO.txTryCommit(tx,type)
23:     if tx.ts_c = ⊥ then
24:         txAbort(ts)
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_c)
31:     if !ok then
32:         txAbort(ts)
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_c)                     ▷ batch write
39:     TSO.txCommit(tx)
```

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 commit or abort the status is updated. 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.

Algorithm 2 $\mathcal{TSO}$ methods (snapshot isolation)

```
struct TxEntry {
  Timestamp ts_c
  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_s 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_c ← getNextTimestamp()         ▷ add Δ
64:     for current = tail → head do
65:         for item ∈ confSet do
                                  ▷ check conflict with concurrent txs
66:             if current.ts_c < 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.ts_c                       ▷ serialization succeeded
```

TXTRYCOMMIT detects inter-transaction conflicts. A transaction acquires a commit timestamp, and traverses the ring 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 $\Delta \gg \delta$ (e.g., $\Delta = 2^{20}$ and $\delta = 1$ 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.

Algorithm 3 $\mathcal{DB}$ methods

```
81: function GET(Key key)                     ▷ atomic
82:     return lastVersion(key)

83: function PUT(Key key, Val val)            ▷ atomic
84:     clock ← clock + δ
85:     put(key, val, clock)

86: function TXGET(Key key, Timestamp ts)
87:     sync(ts)
88:     return lastVersionBefore(key, ts)

89: function TXCHECKCONFLICT(Set⟨KeyValVersion⟩ items,
    Timestamp ts)
90:     sync(ts)
91:     for all item ∈ items do
92:         ⟨val, ts_val⟩ ← lastVersionBefore(item.key,ts)
93:         if item.ts < ts_val then
94:             return FALSE                  ▷ conflict - not ok
95:     return TRUE                           ▷ no conflict - ok

96: function SYNC(Timestamp ts)               ▷ atomic
97:     clock ← max{ts, clock}                ▷ temporal fence
```

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 tuple 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. [6] to adapt our SI algorithm to serializability. Similar to it, we exploit the observations from [12], 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 [12].

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

Algorithm 4 Adaptation for serializability support

```
101: function TXGET(Timestamp ts, Key key)
102:     tx ← getTx(ts)
103:     ⟨val, ts_val⟩ ← DB.get(key)
104:     tx.addToReadSet(key,val,ts_val)
105:     return val

106: function TXCOMMIT(Timestamp ts)
107:     tx ← getTx(ts)
108:     if singleWriteOnly(tx) then
109:         ⟨key, val, ts⟩ ← tx.writeSet.first()
110:         DB.put(key, val)                    ▷ local writing
111:         return                              ▷ committed successfully
112:     ok ← tryCommit(ts, RW)
113:     if ok then
114:         if readOnly(tx) then return
115:         else writeVals(tx)
```

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.

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 expected to be low. We experiment with multiple workloads that feature different traffic mixes (varying proportions of gets 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 transactified, 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 datababse (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 employs LSM trees [22] 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 [9], a popular load generator. The YCSB clients exercise the Mediator, Omid, and HBase API's, depending on the configuration.

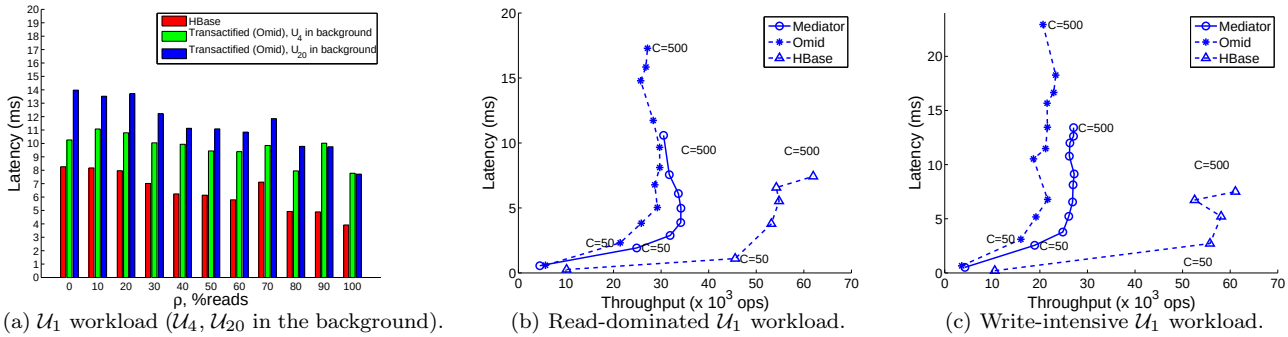
In a single experiment, each YCSB instance drives the same traffic pattern, therefore the cumulative workload remains steady over time. In other words, each client generates the same load bandwidth, which splits independently among (1) reads and writes, and (2) transactional and native accesses. We denote the fraction of reads by ρ, and the fraction of native operations by ν. 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 $\mathcal{U}_n$. The larger n is, the wider the spectrum of the exercised transaction sizes is – from a single access to a bulk of operations. $\mathcal{U}_1$ is a non-realistic workload (singleton transactions workload is meaningless) that we use to demonstrate the local-commit optimization. Table 2 summarizes the notation.

Mediator log is implemented through Bookkeeper [16]. Finally, the TSO employs 1024-bits-wide Bloom filters.

5.2 Numerical Results – Snapshot Isolation

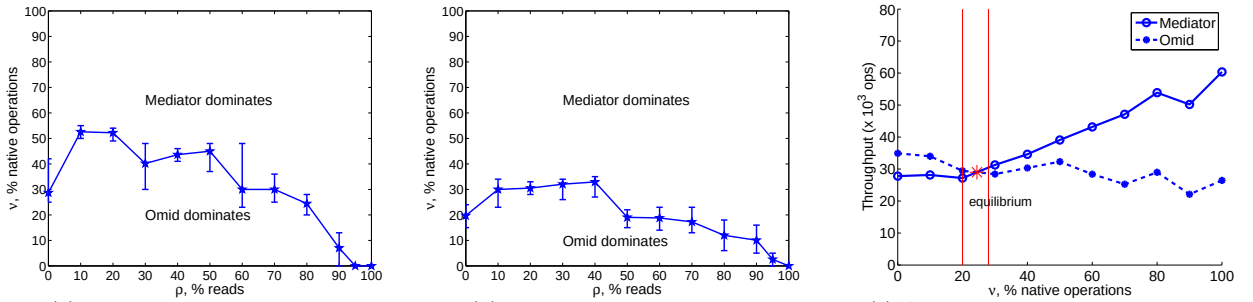
This section studies Mediator's performance under the snapshot isolation consistency model.

Latency Overhead on Single Operations. We start by motivating the advantage of serving native traffic directly.



(a) $\mathcal{U}_1$ workload ($\mathcal{U}_4, \mathcal{U}_{20}$ in the background). (b) Read-dominated $\mathcal{U}_1$ workload. (c) Write-intensive $\mathcal{U}_1$ workload.

Figure 3: **Transaction processing's performance overhead on gets and puts. (a) Latency perspective. HBase compared to Omid's single-operation ($\mathcal{U}_1$) transactions, with larger transactions in the background. Fixed number of clients ($C = 200$), running read ratio ($0 \leq \rho \leq 1$). (b,c) Throughput-Latency perspective. HBase compared to Omid's and Mediator's $\mathcal{U}_1$ transactions. Running number of clients ($5 \leq C \leq 500$).**



(a) Equilibrium curve for $\mathcal{U}_4$. (b) Equilibrium curve for $\mathcal{U}_{20}$. (c) An equilibrium point, $\mathcal{U}_4$, $\rho = 0.8$.

Figure 4: **Comfort zones of Mediator and Omid, for a variety of read-write mixes ($0 \leq \rho \leq 1$) and transactional-native mixes ($0 \leq \nu \leq 1$). The workload is generated by 200 clients.**

Notation	Description	Values
C	number of concurrent clients	$50, 100, \ldots, 1200$
ν	ratio of native operations	$0, 0.1, \ldots, 1$
ρ	ratio of get accesses	$0, 0.1, \ldots, 1$
$\mathcal{U}_n$	transaction size distribution (uniform over $[1, 2, \ldots n]$)	$\mathcal{U}_1$ (singletons), $\mathcal{U}_4, \mathcal{U}_{20}$

Table 2: **Workload parameter notation.**

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 ($\mathcal{U}_1$) transactions with background $\mathcal{U}_4$ traffic, and (3) the same with background $\mathcal{U}_{20}$ 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 $\mathcal{U}_{20}$ 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 ($\mathcal{U}_1$), 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[5]. These results emphasize the potential of consolidating transactional and non-transactional traffic within the same framework, to avoid the overhead of transactifying the latter.

Total Throughput. We now turn to our main goal – contrasting Mediator with Omid on a wide variety of mixed workloads. We study the $\mathcal{U}_1$, $\mathcal{U}_4$ and $\mathcal{U}_{20}$ distributions.

The $\mathcal{U}_1$ case is an exception – Mediator is faster than Omid in all configurations (Figure 3(b) and Figure 3(c) demon-

[5]The same HBase configuration scores much higher throughputs for bulk I/O. This setting is not the focus of our experiment.

strate this for two specific cases). This happens by the virtue of its no-commit optimization (which serves single-get transactions) and its local-update optimization (which serves single-put transactions). Either way, Mediator client does not communicate with the TSO upon commit skipping consistency checks and logging, hence, the oracle's state remains void. Upon transaction begin, the state replicated to the client is minimal (transaction timestamp), and therefore Mediator's overhead is smaller. Obviously, if Mediator is faster for all-transactional $\mathcal{U}_1$ traffic, the same holds if part of the workload is native.

The comparison becomes interesting for truly mixed workloads, in which native operations run side by side with transactions 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 cumulative throughput (in terms of both transactional and native operations) is a convenient metric for evaluating the overall 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 exercises all combinations of read ratios ($0 \leq \rho \leq 1$) and native access ratios ($0 \leq \nu \leq 1$). 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 throughput. The collection of equilibrium points for a given workload type 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 $\mathcal{U}_4$ and $\mathcal{U}_{20}$, 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 $\nu \geq 50\%$. It is consistently more advantageous for $\mathcal{U}_{20}$ versus $\mathcal{U}_4$, due 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 $\rho = 1$, the no-commit optimization applies to all transactions, thus reducing the equilibrium point to zero; for $\rho = 0$, 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 ($\mathcal{U}_4$ and $\mathcal{U}_{20}$), 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 statistically significant dominance.

Abort Ratio. We explore the *abort ratios* incurred by Mediator to user applications – i.e., the fraction of transactios that get aborted due to (possibly falsely) detected conflicts. The probability of colliding with concurrent operations grows with the fraction of updates and with the transaction size. The computation of the write-set intersections is approximate, due to the use of Bloom filters. Therefore, for write-intensive workloads, the fraction of spurious aborts grows as well.

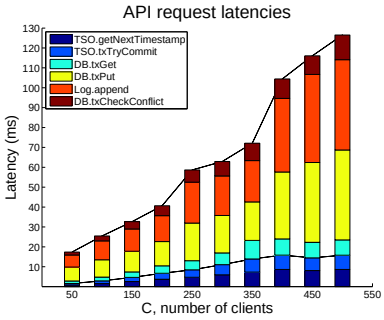


Figure 5: **Breakdown of internal API request latencies, for the $\mathcal{U}_4$ workload ($\rho = 0.5, \nu = 0$), with a varying number of clients. The database and the logger layers are the execution bottlenecks.**

For short $\mathcal{U}_4$ transactions, the abort ratios are totally negligible (below 0.03% under all workloads). For $\mathcal{U}_{20}$ distribution, the ratio remains below 0.1% 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 $\mathcal{U}_4$ 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 experi-

ment 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, 6].

6. MODEL AND CORRECTNESS

We refine the classical definitions of serializability and snapshot isolation (SI) to apply for mixed traffic executions. Then we outline the safety proofs for our algorithms.

6.1 Consistency Models

A *transactional processing system* employs *transactions* to execute pieces of sequential code, namely *operations*, by concurrent asynchrounous processes. An *implementation* of the system provides routines to execute these operations. When a process calls a routine we say it *invokes* an operation, when the execution of the routine is completed a *response* is returned.

An *execution* is a sequence of invocation and responses of native and transactional operations starting from the initial configuration of the database. If the last response of a transaction is a commit or abort indication, then the transaction is *completed*, and is said to be *committed* or *aborted*, respectively; if a transaction invoked a commit but not yet received a response then it is *commit-pending*; otherwise, it is *active*.

In a *serial execution*, native operations and transactions are executed to completion in isolation one after the other.

A transaction T is *legal* in a serial execution if every read invocation of x in T returns a value that was written to x in T before the read, or if there are no invocations of write to x in T before the read, then the read returns a value that was last written to x before T by a native put or a committed transaction, or the initial value of x if no such write to x exists before T. A native read *op* is *legal* in a serial execution if it returns a value that was last written to x before *op* by a native put or a committed transaction, or the initial value of x if no such write to x exists before *op*. A serial execution is *legal* if all its committed transactions and native reads are legal.

An execution α' preserves the *per-process transaction order* of execution α if it preserves the order of any pair of begin operations executed by the same process in α. An execution α' preserves the *per-process item order* of execution α if it preserves the order of any pair of native and transactional operations accessing the same item and executed by the same process in α.

Let T be a committed transaction in an execution α. Let $T|read$ ($T|write$, $T|all$) be the longest subsequence of T in α consisting only of T's read (write, read and write, respectively) invocations and their corresponding responses. The transaction T_r is defined as follows: if $T|read$ is empty then T_r is empty, otherwise T_r is the sequence of a begin invocation and response appended by $T|read$, a commit invocation and a committed response; T_w and T_{rw} are defined in a similar way, w.r.t. $T|write$ and $T|all$.

A *serialization* of an execution α is the sequence σ_α that includes a *serialization point* for every native operation in α, $*_{op}$, for every committed transaction, and for some of the commit-pending transactions in α, $*_T$. The *serial execution* α' corresponding to σ_α is defined by replacing each $*_T$ with T_{rw} and each $*_{op}$ with *op*'s invocation and response in α.

DEFINITION 1 (SERIALIZABILITY). *An implementation supports* serializability *if any execution α has a serialization, σ_α, such that the serial execution α' corresponding to σ_α is legal and preserves the per-process transaction order and per-process item order of α.*

A *snapshot serialization* of an execution α, is the sequence σ_α that includes a serialization point for every native operation in α, $*_{op}$, and for every committed transaction and for some of the commit-pending transactions in α it includes a *read serialization point* $*_{T_r}$ and a *write serialization point* $*_{T_w}$ such that (i) $*_{T_r}$ preceeds $*_{T_w}$, (ii) both $*_{T_r}$ and $*_{T_w}$ are inserted after the invocation of T's begin and before the response of T's commit in α. The *snapshot serial execution* α' corresponding to σ_α is defined by replacing each $*_{T_r}$ with T_r and each $*_{T_w}$ with T_w and each $*_{op}$ with *op*'s invocation and response in α.

For any execution α, consider a snapshot serialization σ_α of α. The *interval* of a native put operation *op* is $*_{op}$. The *interval* of a write-only transactions T is the point $*_{T_w}$. The *interval* of a read-only transactions T is the point $*_{T_r}$. The *interval* of a read-write transaction T is the interval between $*_{T_r}$ and $*_{T_w}$ in σ_α. Two transactions or a transaction and a native put operation *overlap* in σ_α if their intervals overlap.

DEFINITION 2 (SNAPSHOT ISOLATION). *An implementation supports* snapshot isolation *if any execution α has a snapshot serialization σ_α such that:*

Snapshot property *the snapshot serial execution α' corresponding to σ_α is legal and preserves the per-process item order in α.*

Disjoint writes property *The write sets of any pair of overlapping transactions, and any pair of overlapping transaction and native put operation in σ_α, are disjoint.*

The snapshot property implies that native get operations cannot read an uncommitted value; for serializability, the same trivially follows from the total order. The deviation of these definitions from a straightforward extension of classical definitions –conceiving native operations as transactions—is twofold: (1) native operations cannot abort (while transactions can), (2) similar to traditional NoSQL, both models might flip the order of two native gets in the serialization (whereas two read transactions by the same process cannot flip their order). For example, a process that sequentially reads two items might observe an older version in the second read (e.g., since the items are updated in parallel by some transaction).

6.2 Snapshot Isolation Proof

Consider the SI implementation depicted in Algorithms 1, 2, and 3.

We first show that the TSO implies an execution satisfying conditions that are necessary for the disjoint writes and snapshot properties to hold.

LEMMA 1. *Consider a committed not write-only transaction* tx *with start timestamp s and commit timestamp c.*

1. *If* tx *reads a version of item k with timestamp $t \leq s$, then no transaction with commit timestamp q, $t < q \leq s$ writes to k.*
2. *If* tx *writes to item k, then no other transaction with a commit timestamp q, $s \leq q \leq c$ writes to k.*

PROOF. *Claim 1.* When *tx* reads a version of item k with timestamp t, $t \leq s$ (line 4), it verifies with the local replica that no other (committed or commit-pending) transaction writes to k with timestamp q, $t < q \leq s$ (line 5). Otherwise, *tx* aborts (line 8). It is left to show that the local replica includes all entries of transactions with commit timestamp q, $q \leq s$ that have not yet written to the database.

A transaction acquires a commit timestamp only after it has invoked the *txTryCommit* method (line 22) during which it is appended to the ring (line 60), or commits locally by writing to the database (line 15). Therefore, all transactions with commit timestamp that have not yet written to the database appear in the ring. Furthermore, the atomicity of line 60 implies that whene an entry is appended to the ring no other transaction acquires a start timestamp.

When *tx* begins, it acquires its start timestamp s, and the client gets a fresh replica of the ring. This replica includes entries of all transactions with commit timestamp q, $q < s$ that have not yet written to the database, and the claim holds.

Claim 2. Upon commit, the transaction (atomically) applies *checkAndAppend*: first, it acquires a commit timestamp (line 63), then going through the TSO ring from tail to head (line 64) *tx* verifies it has no conflicts with other concurrent transactions that are either active or committed (lines 67, 68), and finally it appends the transaction entry to the ring (line 70). The atomicity implies that the ring entries are ordered by their commit timestamp, and that no other transaction that acquires a commit timestamp can append an entry that violates the claim. □

Both *put* and *sync* are executed atomically, and both are the only means to update the data server's clock. Since *put* increases the clock's value (line 84), and *sync* sets it to the maximum of the current and new values (line 97), the clock in all data servers is monotonically increasing. We use this observation in the following lemmas.

A *timestamp-based snapshot serialization* of an execution E is a snapshot serialization of E that is based on the execution timestamps. The serialization is as follows: for every committed transaction and commit-pending transaction that executed line 15 or line 37 (i.e., is durable), the read serialization point is mapped to the transaction start timestamp, while the write serialization point is mapped to its commit timestamp (or the native put timestamp in case of the local commit optimization). A native put is mapped to its timestamp. A native get is mapped to the timestamp of the version it reads. If a get operation is mapped to the same point as a put operation, the put is serialized ahead of the get.

LEMMA 2. *For any execution E, consider its timestamp-based snapshot serialization σ_E.*

1. *Consider a transaction* tx *serialized in σ_E with start timestamp s. If* tx *reads a version of item k with timestamp t, $t \leq s$, then no put operation in σ_E writes to k between timestamps t and s.*
2. *Consider a transaction* tx *serialized in σ_E with start timestamp s and commit timestamp c. If* tx *writes to item k then no put operation in σ_E writes to k between timestamps s and c.*

PROOF. Let R be the server accommodating item k.

Claim 1. By Claim 1 in Lemma 1 no other transaction with commit timestamp q, $t < q \leq s$ writes to k.

Consider the *lastVersionBefore* method (line 88) applied at R in the last invocation of the *txGet* method during the repeated attempts of the client to read the value (line 4). Prior to returning the value, the transaction updates R's local clock with the start timestamp of *tx* (line 87). The clock's value is monotonically increasing, hence, after this assertion, no native put writes to k with timestamp q, $t < q \leq s$. Therefore no put operation writing to k is mapped to a point between t and s in σ_E.

Claim 2. Since *tx* has a read and write serialization points, it is not write-only. By Claim 2 of Lemma 1, no other transaction with commit timestamp q, $s < q \leq c$ writes to k.

Prior to becoming durable (line 37), the transaction verifies it has no conflicts with native operations or transactions that commit locally by applying a native put. Specifically, *tx* checks conflicts in server R (line 30). First, it updates R's local clock with the commit timestamp of *tx* (line 90). Then, the transaction verifies that no put operation have written to k with timestamp q, $s \leq q \leq c$ (line 93), otherwise *tx* aborts. The clock's value is monotonically increasing hence, after this assertion, no native put writes to k with timestamp q, $s < q \leq c$. Therefore no put operation writing to k is mapped to a point between s and c in σ_E. □

LEMMA 3. *Consider a server R accommodating item k. For any timestamp t, at most one put operation writes to k with t.*

PROOF. Read-write transactions that do not apply the local-commit optimization acquire commit timestamps from the TSO. These are guaranteed to be unique, thus no two read-write transactions write to k with the same timestamp.

Consider two native put operations (either due to a native put by the client or a put transaction applying the local-commit optimization). The emphput method is executed atomically: R's local clock is increased (line 84) prior to writing the value (line 85). The clock's value is monotonically increasing. Therefore, the second native put writes to k with a timestamp that is strictly bigger than the timestamp of the first put.

It is left to show that no native put and transactional put pair can write to item k with the same timestamp. Consider the *txPut* method, invoked by a commit operation of a transaction writing to item k. Prior to writing the values (line 19) and invoking a *txPut* at R (line 38), the transaction checks if it can commit (line 17), and specifically checks for conflicts by invoking *txCheckConflict* (line 30). The transaction updates R's local clock with its commit timestamp c, unless the former is already greater than c (line 90).

We assume that the gap Δ between two consecutive timestamps generated by the TSO is sufficiently large, such that no put applied to R prior to this clock synchronization writes with timestamp c. All puts writing to k after the synchronization are associated with timestamps greater than c (line 84). □

LEMMA 4. *For any execution E, consider the timestamp-based snapshot serialization σ_E and the snapshot serial execution S corresponding to σ_E. It holds that:*

1. *S is legal,*
2. *S preserves the per-process item order in E,*
3. *The transactions and native operations in σ_E satisfy the disjoint writes property.*

PROOF. *Claim 1.* Lemma 3 implies that for any timestamp t, at most a single put operation is serialized at t. By construction of the timestamp-based snapshot serialization, a native read is serialized at the timestamp of the version it reads (after the put operation that writes it, if the version is not the initial value). Therefore, all native reads in S are legal.

For simplicity, we assume that if a transaction writes to an item it does not read this item afterwards. Hence, it is left to prove that the reads of each transaction T_r in S return either the last value written to the item before T_r, or the initial value if no such write exists. By Claim 1 in Lemma 2, if a get in T_r reads item k and returns a value with timestamp t, then no put writes to k in S between t and the read serialization point. Therefore, all transactions in S are legal, and the claim holds.

Claim 2. For brevity, we omit the proof of this claim, which is purely technical – analyzing case by case of any pair of transactional and native operations by the same process accessing k. The proof follows the flow of the algorithm, and relies on the $\Delta \gg \delta$ assumption.

Claim 3. We first note that if two intervals overlap in σ_E, then at least one of them is of a read-write transaction. Consider a read-write transaction tx in E, with a start timestamp s and a commit timestamp c. Claim 2 in Lemma 2 implies that no other transaction or native put writes to any item in tx's write set with timestamp between c and s. In addition, it implies that no other read-write transaction in E, with start timestamp s' and commit timestamp c' writes to any item in tx's write set such that $s' \leq s$ and $c' \geq c$. As these timestamp define the intervals of the transactions and native operations, the claim holds. □

For any execution, Claims 1 and 2 of Lemma 4 imply that it satisfies the snapshot property, and Claim 3 implies it satisfies the disjoint writes property. Therefore, we conclude the following:

THEOREM 5. *A transaction processing implementation that is based on Algorithms 1, 2, and 3 supports snapshot isolation.*

6.3 Proof Sketch for Serializability

We follow the lines of the correctness proof of [6]. By [12, Theorem 2.1], which shows that in any non-serializable execution of a SI implementation there is a "dangerous" structure (a cycle including two consecutive read-write edges in the serialization graph), we only need to prove that whenever an execution contains such cycles, then one of the transactions is aborted.

Consider the SI implementation with the refinements of Algorithm 4. Whenever a read-write conflict is detected, either in the TSO ring (line 22) where the read set is compared against the write keys of all committed transactions, or at a database server (line 30) where the read set of the transaction is compared against native put operations, the committing transaction aborts (line 34). Therefore, we conclude the following.

THEOREM 6. *A transaction processing implementation that is based on Algorithms 1, 2, and 3 with the refinements of Algorithm 4 supports serializability.*

7. RELATED WORK

Transaction processing is a textbook area in database research [23, 14]. It appears in the ANSI SQL standards, as well as in modern NoSQL technologies that took databases to an unprecedented scale (e.g., [19]). The literature defines a wealth of transaction consistency models that capture different perceptions of concurrency control (e.g., [17, 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 [17]. 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 [14]). Should Mediator be extended to support predicate queries, it can use the same transaformation technique by Fekete et al. [12] to get a phantom-free serializable implementation.

Early NoSQL databases, e.g., Google Bigtable [7], Yahoo! PNUTS [8] and Apache HBase sacrifice strong consistency for extreme scalability. Their safety guarantee is single-key atomic reads and writes [2]. Google Percolator technology [19] 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 [24], supports snapshot isolation with a lock-free protocol. Omid also implements serializability [13]. Mediator's design bears similarity with Omid, however, its algorithm is profoundly different, to capture the new safety properties.

Google Spanner [10] 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 [21] also addresses globally distributed transactions, albeit in a different way. It replaces locking with deterministic scheduling that orders transactions through a global consensus service. SCORe [18] 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) [15] is a popular approach for alleviating the difficulty of programming concurrent applications for multi-core and multiprocessing systems. TM allows concurrent processes synchronize via in-memory transactions. Our TSO implementation is inspired by RingSTM [20] – 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.

8. 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.

9. 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.

10. REFERENCES

[1] M. Alomari, M. Cahill, A. Fekete, and U. Rohm. The cost of serializability on platforms that use snapshot isolation. In *ICDE*, 2008.

[2] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics.* Wiley Series on Parallel and Distributed Computing, 2004.

[3] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, and M. Wei. Corfu: A shared log design for flash clusters. In *NSDI*, 2012.

[4] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *SIGMOD '95*, pages 1–10, 1995.

[5] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[6] M. J. Cahill, U. Rohm, and A. Fekete. Serializable isolation for snapshot databases. In *SIGMOD*, pages 729–738, 2008.

[7] F. Chang et al. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[8] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!s hosted data serving platform. In *VLDB*, 2008.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *ACM SoCC 2010*, pages 143–154, 2010.

[10] J. C. Corbett et al. Spanner: Google's globally-distributed database. In *OSDI*, pages 251–264, 2012.

[11] J. O. et. al. The case for ram cloud. *Commun. ACM*, 54:121–130, July 2011.

[12] A. Fekete, D. Liarokapis, E. O'Neil, P. O. Neil, and D. Shasha. Making snapshot isolation serializable. *ACM TODS*, 30:492–528, 2005.

[13] D. Gomez-Ferro and M. Yabandeh. A critique of snapshot isolation. In *EuroSys*, pages 155 – 168, 2012.

[14] J. Gray and A. Reuters. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[15] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[16] F. P. Junqueira, I. Kelly, and B. Reed. Durability with BookKeeper. *OS Review*, 47:9–15, 2013.

[17] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, Oct. 1979.

[18] S. Peluso, P. Romano, and F. Quaglia. SCORe: A scalable one-copy serializable partial replication protocol. In *Proceedings of the 13th International Middleware Conference*, Middleware '12, pages 456–475, 2012.

[19] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, 2010.

[20] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: scalable transactions with a single atomic instruction. In *SPAA '08*, pages 275–284, 2008.

[21] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD*, 2012.

[22] R. Thonangi, S. Babu, and J. Yang. A practical concurrent index for solid-state drives. In *CIKM*, pages 1332–1341, 2012.

[23] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2001.

[24] M. Yabandeh, D. Gomez-Ferro, I. Kelly, F. Junqueira, and B. Reed. Lock-free transactional support for distributed data stores. Accepted to ICDE '14.