Epoxy: ACID Transactions Across Diverse Data Stores

Peter Kraft
Stanford University
kraftp@cs.stanford.edu

Qian Li
Stanford University
qianli@cs.stanford.edu

Xinjing Zhou
MIT
xinjing@mit.edu

Peter Bailis
Stanford University
pbailis@cs.stanford.edu

Michael Stonebraker
MIT
stonebraker@csail.mit.edu

Matei Zaharia
Stanford University
matei@cs.stanford.edu

Xiangyao Yu
University of Wisconsin
yxy@cs.wisc.edu

ABSTRACT

Developers are increasingly building applications that incorporate multiple data stores, for example to manage heterogeneous data. Often, these require transactional safety for operations across stores, but few systems support such guarantees. To solve this problem, we introduce Epoxy, a protocol for providing transactions across heterogeneous data stores. We make two contributions. First, we adapt multi-version concurrency control to a cross-data store setting, storing versioning information in record metadata and filtering reads with predicates on metadata so they only see record versions in a global transaction snapshot. Second, we show our design enables an atomic commit protocol that does not require data stores implement the participant protocol of two-phase commit, requiring only durable writes. We implement Epoxy for five data stores: Postgres, Elasticsearch, MongoDB, Google Cloud Storage, and MySQL. We evaluate it by adapting TPC-C and microservice workloads to a multi-data store environment. We find it has comparable performance to the distributed transaction protocol XA on TPC-C while providing stronger guarantees like isolation, and has overhead of <10% compared to a non-transactional baseline on read-mostly microservice workloads and 72% on write-heavy workloads.

PVLDB Reference Format:
Peter Kraft, Qian Li, Xinjing Zhou, Peter Bailis, Michael Stonebraker, Xiangyao Yu, and Matei Zaharia. Epoxy: ACID Transactions Across Diverse Data Stores. PVLDB, 16(11): 2742 - 2754, 2023.
doi:10.14778/3611479.3611484

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/DBOS-project/apiary.

1 INTRODUCTION

Developers are increasingly building large-scale applications utilizing multiple data storage systems. This is driven by two trends in application design. First, applications increasingly use multiple data stores to manage heterogeneous data. For example, an online store may process customer transactions in Postgres, but store item data in Elasticsearch for rapid search and store images in S3 for cheap storage; these specialized systems are far more efficient than an RDBMS for such tasks. Second, developers are embracing microservice architectures, where large applications are decomposed into smaller services, each managing their own data [17].

Many applications require transactional safety for operations across multiple data stores as without these, they are exposed to serious concurrency anomalies. The traditional solution to providing transactions across data stores is to use a distributed transaction protocol based on two-phase commit such as X/Open XA [34]. However, these do not provide transactional isolation (only atomicity) and additionally require data stores to implement the participant protocol of two-phase commit, which is not supported in many popular data stores such as MongoDB, CockroachDB, and Redis. Cherry Garcia [12] provides ACID transactions across heterogeneous stores, but is limited to key-value operations and not other query models. Skeena [35] provides transactions across multiple engines in the same database, but requires modifying these engines and assumes they share memory. Because of these limitations, developers in practice provide cross-data store transactions by manually managing concurrency control, for example by implementing ad hoc transactions in application code. However, such code is difficult to write and is a frequent source of bugs and errors [31].

To solve this problem, we introduce Epoxy, a protocol providing ACID transactions across heterogeneous data stores. The key challenge in Epoxy is to support distributed ACID transactions without changing the internals of these stores, which significantly constrains the design. We make two main contributions.

First, we adapt multi-version concurrency control (MVCC) to a cross-data store setting to provide transactional isolation. This requires solving two challenges. First, conventional MVCC approaches rely on a co-designed data management layer providing record versioning using custom data structures like version chains [33], but heterogeneous data stores do not support these. However, we observe that while few data stores support versioning, most provide efficient metadata filtering. Thus, Epoxy stores version information *in record metadata*, interposing on writes to version records and on reads to filter input tables, so transactions only see record versions in the transaction snapshot. Second, unlike conventional MVCC, we must ensure a transaction reads from consistent snapshot across multiple heterogeneous stores. To make this possible, we manage snapshots centrally in the transaction coordinator. To optimize this, we observe that in most applications utilizing multiple data stores, at least one is a transactional DBMS [17], so we design Epoxy to use a DBMS as a coordinator.

Second, our design enables a simple commit protocol to provide transactional atomicity and durability without requiring data stores

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 16, No. 11 ISSN 2150-8097.
doi:10.14778/3611479.3611484

implement the participant protocol of two-phase commit. In protocols based on two-phase commit, such as XA, each transaction participant must `prepare` the transaction to indicate it is ready to commit, promising it will not unilaterally abort but can still be rolled back. However, many popular data stores do not support `prepare` because it can only be implemented in a transactional store and requires the cooperation of its concurrency control system. By contrast, because we assume all participants utilize Epoxy concurrency control, we only require they make writes durable. Epoxy's concurrency control protocol ensures that a transaction cannot read durable but uncommitted writes or write to records modified by a durable but uncommitted transaction; transactional isolation is controlled by the coordinator instead of the individual participants. Thus, transactions can be rolled back at any point before they commit on the coordinator, eliminating the need for a two-phase commit protocol like XA. The coordinator commits a transaction once it is complete, validated, and made durable on all participants, guaranteeing transactions either completely succeed or completely abort and are durable once committed.

As Epoxy only requires record metadata filtering and durable writes, it supports most data stores we consider. To prove this, we implement Epoxy in *shim layers* on top of five heterogeneous stores: Postgres, MySQL, Elasticsearch, MongoDB, and Google Cloud Storage (GCS). These shims can provide ACID guarantees for transactions between any combination of these data stores. Each requires <1K lines of data store-specific code and no changes to the underlying data store. One limitation of Epoxy is that it must be the exclusive mode of accessing a table in a participating store: if one application accessing a table adopts it, all applications must adopt it for operations on that table. However, Epoxy interposes transparently on operations, so adopting it only requires redirecting queries from a store to its shim layer.

We evaluate Epoxy by adapting TPC-C and microservice workloads to a multi-data store setting. On TPC-C, Epoxy performs similarly to XA but provides stronger guarantees like transactional isolation. On microservices, Epoxy adds <10% overhead compared to a non-transactional baseline on read-mostly workloads and 72% on write-heavy workloads. In summary, our contributions are:

- We propose Epoxy, a protocol providing ACID transactions across heterogeneous data stores. Epoxy provides isolation by adapting MVCC to a cross-data store setting and atomicity and durability through a commit protocol that only requires data stores provide durable writes.
- We implement Epoxy for five diverse data stores: Postgres, MySQL, Elasticsearch, MongoDB, and Google Cloud Storage.
- We show Epoxy performs similarly to XA and adds 10-72% overhead compared to a non-transactional baseline.

2 EPOXY ARCHITECTURE

Epoxy is a protocol for providing ACID transactions across diverse data stores. We use a *primary* transactional DBMS as a transaction coordinator for transactions among it and several potentially non-transactional *secondary* data stores. Epoxy is implemented in shim layers co-located with these data stores which intercept and interpose on client requests, but do not require modifications to the stores themselves. We sketch Epoxy's architecture in Figure 1.

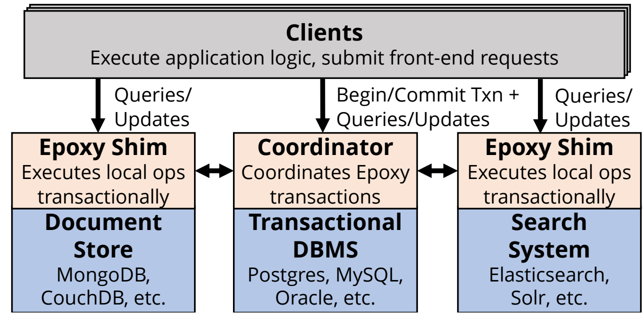

Figure 1: Architecture of Epoxy.

One of our goals in Epoxy is to support *heterogeneous* secondary stores with diverse data models, including full-text search systems like Elasticsearch, NoSQL document stores like MongoDB, and object stores like GCS. Thus, our shims do not manage data or process queries themselves. Instead, they transparently interpose on writes to add versioning metadata to records and on reads to filter data sources based on that metadata, then pass those operations on to the secondary store which can freely optimize and execute them.

2.1 Epoxy Assumptions

We require the primary database provide ACID transactions with at least snapshot isolation. Our implementation uses Postgres.

We make three secondary store assumptions for correctness:

- Single-object write operations are linearizable and durable.
- Each record has a uniquely identifiable key.
- Epoxy is the exclusive mode of accessing a secondary store table: if one application accessing a table adopts it, all applications accessing that table must adopt it for operations on that table.

As we will see, a wide variety of data stores, many of which are non-transactional, satisfy these assumptions. We implement Epoxy with four of them: Elasticsearch, MongoDB, GCS, and MySQL.

We also make another assumption needed for performance:

- Records can include metadata, and queries can be efficiently filtered based on this metadata.

Many data stores satisfy this assumption by supporting complex record types and efficient indexed filtering. For example, we can store metadata in record fields in MongoDB and Elasticsearch and additional columns in MySQL, then index it for efficient filtering using B-trees in MongoDB and MySQL and numeric indexes in Elasticsearch. If data stores do not support efficient metadata filtering, we must store versioning information in the primary database, which entails additional communication per query. We manage metadata in this way for Google Cloud Storage (GCS), finding overhead is low because GCS's latency is naturally high. However, we do not expect this to be practical for low-latency secondary stores.

2.2 Epoxy Interface

Users perform Epoxy transactions through a small client library, shown in Figure 2, which interfaces with Epoxy shims. The primary database shim acts as the transaction coordinator and serves client-initiated requests to begin, commit, and abort cross-data store

Client-Transaction Coordinator Interface

beginTransaction()	➢ Begin Epoxy transaction, create a global snapshot.
commitTransaction()	➢ Commit an Epoxy transaction.
abortTransaction()	➢ Abort and rollback an Epoxy transaction.

Client-Secondary Store Shim Interface

query(Query, List[Arg]) → Result	➢ Execute a query, filtering its input to only see record versions in the transaction snapshot.
update(Key, Record)	➢ Update a record: create a new version then mark the previous version as not visible to future txns.
delete(Key)	➢ Delete a record by marking it as not visible to future transactions.

Figure 2: The Epoxy client library.

```
1   def reserve(hotelId, customerData):
2     ctxt = epoxy.beginTransaction()
3     # Check room availability in Postgres.
4     res = pg.query("SELECT avail FROM Hotels WHERE
       hotel=hotelId") # Epoxy does not interpose
      on primary database operations.
5     if res == 0:
6       epoxy.commitTransaction(ctxt)
7       return false # No room available.
8     # Update availability in Postgres.
9     pg.update("UPDATE Hotels SET avail=res-1 WHERE
       hotel=hotelId")
10    # Make a reservation in MongoDB.
11    epoxy.update(context=ctxt, secondary=mongo,
      key=hotelId, record=customerData)
12    epoxy.commitTransaction(ctxt)
13    return true
```

Figure 3: In an application storing hotel information in Postgres and customer data in MongoDB, we use Epoxy to transactionally validate room availability, then make a reservation.

transactions. Users can also query and update the primary database directly without going through the shim. Each secondary store shim transparently interposes on client-issued data operations on that secondary store, tagging writes with version information and filtering reads to only see appropriate record versions.

We show an example of an Epoxy transaction in Figure 3. Suppose a hotel reservation application stores hotel information in Postgres and customer information in MongoDB. Without cross-data store transactions, the application cannot atomically reserve a room then record customer information, so it may incur anomalies such as booking a room but not storing customer data for the booking. Additionally, because there is no transactional isolation, it is possible for concurrent operations to find customer information in MongoDB that does not correspond to any room reservation in Postgres (or vice versa), potentially violating constraints and causing errors. Using Epoxy, the application performs both operations in a single transaction, providing both atomicity and isolation.

3 EPOXY PROTOCOL

In this section, we discuss Epoxy's data structures, core algorithm, and concurrency control and failure recovery mechanisms.

3.1 Epoxy Data Structures

To provide transactional isolation, Epoxy utilizes two data structures: a snapshot representation and a record versioning scheme.

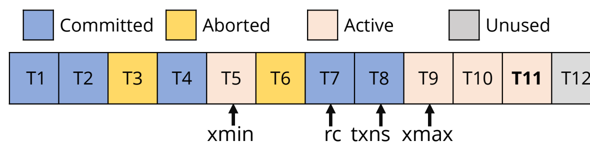

Figure 4: The Epoxy snapshot for transaction T11. `xmin` is the smallest active transaction T5, `xmax` is transaction T9 (one past the largest committed transaction T8), and `rc_txns` contains recently committed transactions T7 and T8.

Legend: Visible / Not Visible

Item (Record Key)	**Price**	**beginTxn**	**endTxn**
TV	$500	T1	∞
Microwave	$50	T4	T8
Microwave	$60	T8	∞
Fork	$1	T2	T9
Fork	$2	T9	∞

Figure 5: Record versions visible to transaction T11 from Figure 4. The TV is visible at $500. The microwave is visible at $60 because T8 committed before the snapshot was taken. The fork is visible at $1 because T9 was still active when the snapshot was taken.

Snapshots. Each Epoxy transaction is associated with a snapshot, the set of all past transactions which are visible to it. Because this set can be large, managing it directly is impractical, so we need a compact snapshot representation. We assume transaction IDs increase monotonically. Borrowing notation from Postgres, we represent a snapshot using two transaction IDs, `xmin` and `xmax`, and a list of recently committed transactions `rc_txns`. We diagram this in Figure 4. At the time the snapshot is taken, `xmin` is defined as the smallest active transaction ID, `xmax` is defined as one past the largest committed transaction ID, and `rc_txns` is defined as the set of committed transactions with ID greater than `xmin`. We assume that aborted transactions are considered active until they are fully rolled back; in §5 we discuss how to enforce this. Thus, a transaction with ID x is in a snapshot if $(x < \texttt{xmin}) \vee (x \in \texttt{rc_txns})$. It is important to note that snapshots are *monotonic*: if T_1 is in the snapshot of T_2 and T_2 in that of T_3, then T_1 is in the snapshot of T_3.

Record Metadata. Epoxy secondary store shims tag record versions with metadata so that read operations can easily identify which versions are in the transaction snapshot. Specifically, shims tag all record versions with two values: `beginTxn` and `endTxn`. `beginTxn` is the ID of the transaction that created the record version; `endTxn` is the ID of the transaction that superseded it with a new version or deleted the record entirely. A record version is visible to a transaction if and only if `beginTxn` is in its snapshot but `endTxn` is not. We diagram this in Figure 5.

3.2 Epoxy Transactions

An Epoxy transaction executes in four phases, sketched in Figure 6 with pseudocode shown in Algorithm 1. Each phase is initiated

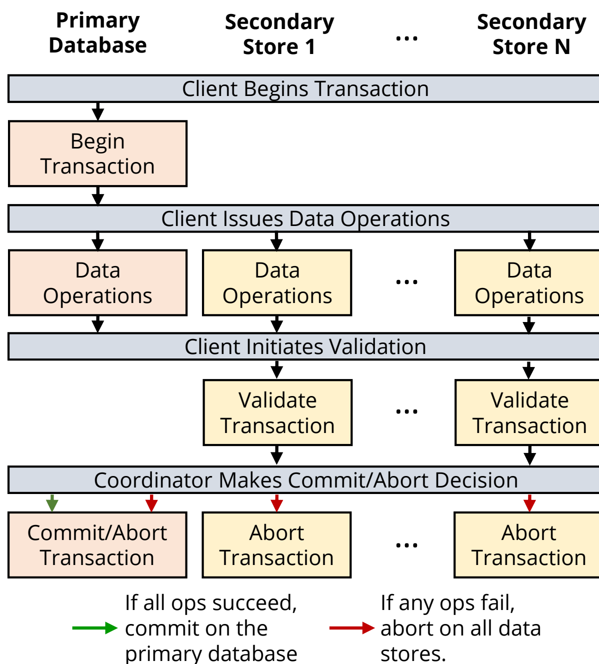

Figure 6: Steps taken by an Epoxy transaction. First, a client begins a transaction on the primary database. Then, the client issues read and write operations, which are interposed on to enforce transactional isolation. Next, the client instructs secondary stores to validate the transaction. Finally, the coordinator decides whether to commit or abort.

by the client. First, the client begins a transaction, instructing the transaction coordinator to begin a transaction on the primary database and create a snapshot of all committed transactions. Then, the client executes the transaction's business logic and issues read and write operations, which secondary store shims interpose on to enforce transactional isolation. Next, the client instructs each secondary store shim to validate that the transaction does not conflict with concurrent committed transactions. Finally, the client communicates these results to the coordinator, which decides to commit (or abort) the transaction.

Begin Transaction. The coordinator initiates an Epoxy transaction by beginning a transaction in the primary database (lines 3-7 in Algorithm 1). If the transaction is between secondary stores, this is an empty transaction in the primary database. After initiating a transaction, the coordinator creates a snapshot of all committed transactions. The snapshot is represented in a summary format (§3.1) and may be computed from active transaction metadata maintained in memory in the coordinator or from snapshot information provided by the primary database. We describe how we compute summary information in Postgres in §5.

Data Operations. After a transaction is initiated, it executes its business logic. Transactions can perform arbitrary read and write operations in the primary database and secondary stores. Reads and writes to the primary database are not interposed on.

Algorithm 1 Epoxy Functions

```
1:  Connection primary                              ▷ Connection to primary database.
2:  List[Connection] secondaries                    ▷ Connections to all secondary stores.
3:  function BEGINTRANSACTION()
4:      primary.beginTransaction()
5:      TxnContext txn                              ▷ Initialize transaction context.
6:      txn.txID, txn.xmin, txn.xmax, txn.rc_txns ← primary.snapshot()
7:      return txn
8:  function UPDATE(TxnContext txn, Secondary s, Key k, Record r)
9:      r.set("key", k)
10:     r.set("beginTxn", txn.txID)
11:     r.set("endTxn", ∞)
12:     s.writeLock(k)                              ▷ Abort if lock already held.
13:     txn.s.modifiedKeys.add(k)
14:     s.find(key=k ∧ endTxn=∞).set("endTxn", txn.txID)
15:     s.insertRecord(k, r)
16: function DELETE(TxnContext txn, Secondary s, Key k)
17:     s.writeLock(k)                              ▷ Abort if lock already held.
18:     s.find(key=k ∧ endTxn=∞).set("endTxn", txn.txID)
19: function QUERY(TxnContext txn, Secondary s, Query q)
20:     q.addPredicate(((beginTxn < txn.xmin) ∨ (beginTxn ∈ txn.rc_txns)
        ∨ (beginTxn = txn.txID)) ∧ (endTxn ≥ txn.xmin) ∧ (endTxn ∉
        txn.rc_txns) ∧ (endTxn ≠ txn.txID))
21:     return s.query(q)
22: function VALIDATE(TxnContext txn)
23:     valid ← true
24:     for s ∈ secondaries do                      ▷ Validate secondary stores.
25:         s.validateLock.lock()
26:         for cTxn ∈ s.validTxns do
27:             if cTxn.txID >= txn.xmin ∧ cTxn.txID ∉ txn.rc_txns then
28:                 for k ∈ txn.s.modifiedKeys do
29:                     if k ∈ cTxn.s.modifiedKeys then
30:                         valid ← false
31:             if valid then s.validTxns.add(txn)
32:         s.validateLock.unlock()
33:     return valid
34: function COMMIT(TxnContext txn)
35:     primary.commitTransaction()
36:     for s ∈ secondaries do
37:         s.releaseWriteLocks(txn.s.modifiedKeys)
38: function ABORT(TxnContext txn)
39:     for s ∈ secondaries do                      ▷ Roll back secondary stores.
40:         for k ∈ txn.s.modifiedKeys do
41:             s.find(key=k ∧ beginTxn=txn.txID).delete()
42:             s.find(key=k ∧ endTxn=txn.txID).set(endTxn, ∞)
43:     primary.abortTransaction()
44:     for s ∈ secondaries do
45:         s.validTxns.remove(txn)
46:         s.releaseWriteLocks(txn.s.modifiedKeys)
47: function GARBAGECOLLECT(List[TxnContext] activeTxn)
48:     if activeTxn.isEmpty() then
49:         globalXmin ← primary.snapshot().xmin
50:     else
51:         globalXmin ← min(txn.xmin ∀ txn ∈ activeTxn)
52:     for s ∈ secondaries do                      ▷ Garbage collect secondary stores.
53:         s.findAll(endTxn < globalXmin).delete()
```

Secondary store shims interpose on write operations to version records (lines 8-15 in Algorithm 1). Let us say a transaction with ID

x writes a record with key k to a secondary store. The secondary store's shim first takes an exclusive write lock on k (we discuss concurrency control in §3.3). It then creates a new version of the updated record with beginTxn set to x and endTxn set to infinity. Next, it checks if an older version of the record exists and, if one does, it sets the endTxn field of the most recent older record version to x. This check implicitly enforces the uniqueness of k. Because these operations are not visible to other transactions until the transaction commits, they need not be atomic and so can be implemented on any data store that supports record metadata. If a transaction performs a delete, the shim sets the endTxn field of the most recent version of the deleted record to x; once the transaction commits, the record will no longer be visible (lines 16-18 in Algorithm 1).

Secondary store shims interpose on read operations so they only see record versions in the transaction snapshot (lines 19-21 in Algorithm 1). Specifically, transactions see all record versions that were created by transactions in the snapshot but have not been superseded or deleted by transactions in the snapshot. This is equivalent to saying transactions can only see record versions whose beginTxn field is in the snapshot and whose endTxn field is not. Additionally, transactions can read their own writes. We can express this condition as a filter, so a read in a transaction with ID X can only see records that satisfy:

$$((\texttt{beginTxn} < \texttt{xmin}) \lor (\texttt{beginTxn} \in \texttt{rc_txns}) \lor (\texttt{beginTxn} = x))$$
$$\land(\texttt{endTxn} \geq \texttt{xmin}) \land (\texttt{endTxn} \notin \texttt{rc_txns}) \land (\texttt{endTxn} \neq x)$$

Lemma 1. *For some record r, at most one version of r can satisfy this statement for a given transaction T with ID x, and this version reflects the most recent transaction in the snapshot of T to modify r.*

Define r_n as the most recent version of r that satifies this statement (if one exists); it was created by transaction T_n with ID x_n. T_n must be in the snapshot of T as otherwise the beginTxn field of r_n (set to x_n) would not satisfy the statement. All prior transactions modifying r must be in the snapshot of T_n (and also of T, as snapshots are monotonic) because if they were not, T_n would not have passed validation (§3.3). Thus, the most recent previous version r_{n-1} was created by a transaction in T_n's snapshot, so it either was visible to T_n or was deleted by an intervening transaction in T_n's snapshot. In either case, r_{n-1} is not visible to T: in the former, because T_n would set its endTxn field to x_n, in the latter because the deleting transaction (due to montonicity) is also in T's snapshot. A similar argument applies to r_{n-2} and all previous versions of r, so at most one version of r is visible to T.

Now, define T_k as the most recent transaction in the snapshot of T to modify r. If it created a new version of r, that version is visible to T as its beginTxn field is in T's snapshot and its endTxn field (if not ∞) is set by a transaction more recent than T_k, which is by definition not in T's snapshot. If it deleted r, then no version of r is visible to T because at most one version was visible prior to the deletion, and the deletion makes that version not visible. Either way, the version of r (or lack thereof) visible to T reflects the most recent transaction in the snapshot of T to modify r.

Validation. Our implementation of Epoxy uses optimistic concurrency control, requiring a validation step after data operations complete to ensure they do not conflict with concurrent committed transactions on any secondary store. We show this step in lines 22-33 of Algorithm 1 and discuss it in detail in §3.3.

Commit or Abort Transaction. A transaction is ready to commit once it validates on all secondary stores. Transactions commit by committing on the primary database (lines 34-37 in Algorithm 1). This atomically makes the transaction visible to future transactions on all data stores as it will appear in their snapshots. Thus, the set of committed transactions in Epoxy is equivalent to the set of transactions committed in the primary database; we make use of this property to perform failure recovery (§3.4) and enforce transactional atomicity and durability (§4.2).

If a transaction fails validation or encounters any error in any data store, it aborts. To prevent transactions from hanging indefinitely on client failure, the coordinator also aborts a transaction if its connection with the client times out. The transaction aborts in the primary database and rolls back all changes made in secondary stores by deleting newly added record versions and reverting record endTxn fields (lines 38-46 in Algorithm 1). A transaction can safely abort at any point before it commits because its uncommitted changes are never visible to other transactions.

Optimizing Read-Only Transactions. While the previously-described procedure is necessary for any transaction that may modify data, we can optimize read-only transactions on secondary stores to bypass the transaction coordinator. All a read-only transaction requires from the coordinator is a snapshot from which to read; it makes no changes to validate or commit. A secondary store shim can cache snapshot information in memory and use it to run new read-only transactions instead of going through the coordinator. Such transactions are guaranteed to reflect all data committed as of when the snapshot was taken. Importantly, this optimization allows read-only transactions on secondary stores to proceed in the event of a transaction coordinator failure, improving availability.

3.3 Optimistic Concurrency Control

To enforce transactional isolation (specifically snapshot isolation), we adapt the multi-version optimistic concurrency control (OCC) protocol of Larson et al. [18], originally designed for a single-node main-memory database, to a multi-data store setting. We use OCC and provide snapshot isolation because this naturally fits our lightweight shim model, requiring us only to check for write-write conflicts. To use a pessimistic locking scheme or to provide serializable isolation, we would have to efficiently detect read-write conflicts, which requires knowledge of query semantics and thus must be implemented in a data store-specific manner on each shim.

Assume secondary store S is executing transaction T. Before S writes a record, it acquires an exclusive lock on that record's key. Each secondary store shim contains a lock manager for records in that store, with one exclusive write lock for each record. This lock prevents transactions from concurrently modifying the endTxn field of the previous version of that record. If S fails to acquire a lock, it is guaranteed to conflict with the lock holder, so it aborts T. After S finishes T, it validates it. S takes an exclusive (but local to S) validation lock, then verifies that no key written to by T was also written to by a committed transaction not in T's snapshot (lines 22-33 in Algorithm 1). If T passes this validation, S provisionally

marks it as committed, releases the lock, then votes to commit. If the coordinator later decides to abort T, S unmarks it (we take this approach to minimize the time the validation lock is held; it can cause unnecessary validation failures if T aborts, but not incorrect validations). If S fails and restarts, it recovers the list of committed transactions from the coordinator (§3.4). A transaction only commits if all secondary stores successfully validate; otherwise it aborts and rolls back. Secondary stores release write locks after learning of a commit or completing a rollback.

3.4 Availability and Failure Recovery

Epoxy builds on the availability and durability guarantees of participating stores. We do not aim to provide higher availability than participating stores provide natively, but do guarantee that we can recover to a consistent state from failures of any combination of participating stores. If any store becomes unavailable (we currently only consider crash failures), we assume it restarts and recovers all durable data, then our fault tolerance protocol restores it to a state consistent with all other stores. We rely on stores' clients to tell us when they are unavailable and when they have recovered.

Secondary Store Failures. In the event of a failure of secondary store S (or of its shim), Epoxy guarantees that transactions not involving S proceed without disruption and that S recovers to a state reflecting all committed transactions. When S fails, the transaction coordinator aborts all active transactions involving S and disallows any new transactions involving S. During the period of failure, transactions not involving S proceed normally. After S restarts, it queries the coordinator for the list of committed transactions involving S. The coordinator ensures that all active transactions involving S are aborted before sending this list. S is guaranteed to contain all record versions created by these committed transactions (excepting garbage-collected outdated versions, see §3.5) as the Epoxy commit protocol does not commit a transaction until all secondary stores have validated and persisted its changes. S then undoes the effects of any aborted transactions (because the coordinator aborted all active transactions involving S before sending the list, all uncommitted transactions involving S are aborted): it deletes any records with the `beginTxn` of an aborted transaction and resets to infinity any `endTxn` values set to an aborted transaction ID. We can recover S without any Epoxy-specific logging because Epoxy relies on the coordinator as a source of truth for which transactions have committed and assumes S persists all transaction data. If S fails while recovering, it simply restarts, re-requests the list (which does not change), and reruns the undo process. S is now recovered to a state reflecting all committed transactions and no uncommitted transactions, so new transactions can proceed normally.

Primary Database or Coordinator Failures. In the event of a failure of the primary database P or the transaction coordinator, Epoxy guarantees that read-only transactions on secondary stores proceed without disruption and that all stores recover to a state reflecting only the transactions committed on P. Upon detecting a failure of P or the coordinator, each secondary store shim aborts and rolls back any active transactions. During the period of failure, no transactions that perform writes may execute, but read-only transactions on secondary stores can proceed normally following the procedure described in §3.2, bypassing the coordinator to read from a cached snapshot. We assume that the coordinator restarts and P recovers to a consistent state reflecting all committed transactions. Because (as discussed in §3.2) the set of committed Epoxy transactions is equivalent to the set of transactions committed to P, and because we assume P transactions are ACID, this recovery does not require Epoxy-specific logging, instead leveraging P's native recovery mechanism. The coordinator then instructs each secondary store to recover following the procedure in the previous paragraph. If any secondary stores fail during this process, the coordinator waits for them to restart, then recovers them again. This procedure recovers each secondary store to a state where it reflects all committed transactions but no uncommitted transactions. Once all secondary stores are recovered, new transactions proceed normally.

3.5 Garbage Collection

Because writes create new record versions instead of updating existing records, it is important to clean up old record versions. Record versions can be safely deleted if they are no longer visible to any transactions, meaning their `endTxn` is in the snapshot of all active transactions. Periodically, the transaction coordinator runs a garbage collector which deletes all record versions satisfying this condition (lines 47-53 of Algorithm 1). The garbage collector scans all active transactions and finds the transaction with the smallest `xmin` (`xmin` increases monotonically, so this is the oldest active transaction). It then instructs secondary store shims to delete all record versions whose `endTxn` is less than this smallest active `xmin`.

4 CORRECTNESS AND DISCUSSION

In this section, we prove the correctness of Epoxy's isolation, atomicity, and durability guarantees, then discuss limitations.

4.1 Isolation Correctness

We first prove Epoxy provides the two properties of snapshot isolation, as defined by Adya [2] for a transaction T (SI1-2).

SI1: T always reads data from a snapshot of committed information valid as of the time T started. We always take snapshots at the beginning of a transaction, so this follows from Lemma 1 (§3.2): for a given record r, T can only read at most a single version of r and that version reflects the most recent transaction in the snapshot of T to modify r.

SI2: T can only commit if, at commit time, no committed transaction not in the snapshot has written data that T intends to write. This is enforced by our validation protocol (§3.3). T only validates if no key written to by T was also written to by a committed transaction not in T's snapshot.

4.2 Atomicity and Durability Correctness

To prove the atomicity and durability of Epoxy, we follow the structure in [5] and show it has the five properties of an atomic commit protocol (AC1-5). This proof builds on the Epoxy recovery protocol (§3.4). We say that a secondary store has voted to commit if it signals the transaction is locally complete, persisted, and validated. We say the coordinator has made a decision to commit if the transaction is committed on the primary database.

AC1: All processes that reach a decision reach the same one. Specifically, all stores reflect the decision made by the coordinator. The coordinator can only commit if each secondary store votes to commit. If it decides to abort, it rolls back the transaction on all secondary stores. If a secondary store fails, it is recovered (§3.4) to a state reflecting only the transactions committed by the coordinator. If the primary database or coordinator fail, they restart and recover (§3.4) themselves and all secondary stores to a state reflecting only the transactions committed by the coordinator prior to the failure, implicitly aborting all active transactions.

AC2: A process cannot reverse its decision after it has reached one. We have already shown that all stores reflect the decision of the coordinator. The coordinator makes decisions by committing or aborting on the primary database, which is required (§2.1) to provide ACID transactions, so its decisions are irreversible.

AC3&4: The Commit decision can only be reached if all processes voted Yes. If there are no failures and all processes voted Yes, then the decision will be to Commit. Both these properties are clearly enforced by our commit protocol (§3.2).

AC5: Consider any execution containing only failures that the algorithm is designed to tolerate (i.e., crash failures). At any point in this execution, if all existing failures are repaired and no new failures occur for sufficiently long, then all processes will eventually reach a decision. Specifically, the coordinator always reaches a decision; we have already shown all stores reflect the decision of the coordinator. If a secondary store fails before voting, the coordinator aborts. If a secondary store fails after voting, the coordinator makes a decision following the commit protocol. If the primary database or coordinator fail, they recover (§3.4) to a state reflecting only transactions committed prior to the failure, implicitly aborting all active transactions.

4.3 Limitations

One limitation of Epoxy is that it must be the exclusive mode of accessing a secondary store table. If a client writes to a secondary store table without using it, the write will lack the version information needed to be visible to reads. If a client reads from a secondary store table without using it, the read may see multiple potentially conflicting versions of the same record. Thus, if one application accessing a secondary store table adopts Epoxy, all other applications accessing that table must also adopt it for operations on that table.

Another limitation of Epoxy is that while it enforces primary key constraints (requiring each record to have a uniquely identifiable key), it does not currently support other constraints. Secondary store shims store different record versions as separate records in the secondary store, so Epoxy clashes with native constraint enforcement and may cause erroneous constraint violations. This can be solved by enforcing constraints in the shim itself (in the same way we already enforce key uniqueness and thus primary key constraints), but we leave that to future work.

5 IMPLEMENTATION

In this section, we discuss how we implement Epoxy primary database and secondary store shims on a range of systems. We implement one transaction coordinator, on Postgres, and four secondary store shims, on Elasticsearch, MongoDB, Google Cloud Storage, and MySQL. We implement each shim in <1K lines of Java code.

5.1 Postgres

We use Postgres as a primary database in our experiments, setting its isolation level to repeatable read (which Postgres implements as snapshot isolation). Because Postgres uses MVCC, we can optimize snapshot creation using Postgres system tables. Postgres represents a transaction snapshot using `xmin`, `xmax` (defined as in Epoxy), and a list of active transactions at the time of the snapshot `xip_list` [25]. This is slightly different than our snapshot definition, so we modify the expression (§ 3.2) used to determine whether a secondary store record version is visible to a transaction with ID x:

$$(((\texttt{beginTxn} < \texttt{xmax}) \lor (\texttt{beginTxn} = x)) \land (\texttt{beginTxn} \notin \texttt{xip_list}))$$
$$\land((\texttt{endTxn} \geq \texttt{xmax}) \lor (\texttt{endTxn} \in \texttt{xip_list})) \land (\texttt{endTxn} \neq x)$$

A major challenge in this expression is handling aborted transactions. We do not want record versions created by an aborted transaction to ever be visible, so we need aborted transactions to be considered active until they are rolled back in the primary database and all secondary stores. Thus, the transaction coordinator keeps track of all transactions that are active or currently being rolled back. When starting a new transaction and creating its snapshot, the coordinator scans these transactions (except those in `xip_list`) and adds them to `xip_list` if Postgres reports they have aborted.

5.2 Elasticsearch

We implement a secondary store shim for the popular full-text search system Elasticsearch. Elasticsearch stores data as documents, indexing them for fast search. To implement an Elasticsearch shim, we add numeric `beginTxn` and `endTxn` fields to all documents and manage them during writes as described in §3.2. All Elasticsearch queries are searches that find and rank documents based on a set of conditions, so our shim adds to queries a filter similar to that described in §3.2. Elasticsearch natively supports fast indexed range queries on numeric fields, so this additional predicate is efficient.

5.3 MongoDB

We implement a secondary store shim for the popular NoSQL document database MongoDB. MongoDB provides a schemaless document-oriented data format backed up by indexes. Like our Elasticsearch shim, our MongoDB shim adds `beginTxn` and `endTxn` fields to all documents and manages them during writes as described in §3.2. Our shim interposes on queries by inserting operators that filter all input collections using the conditions described in §3.2 to ensure the query only sees record versions in the transaction snapshot. To improve performance, our shim indexes `beginTxn` and `endTxn` using B-trees, which MongoDB supports natively.

5.4 Google Cloud Storage

We implement a secondary store shim for the cloud object store Google Cloud Storage (GCS); we believe similar principles could be used for other cloud object stores such as AWS S3 or Azure Blob Storage. GCS provides a key-value interface for durably storing large blobs, where each blob is associated with a unique key. While

GCS satisfies the three correctness assumptions we make in §2.1, it lacks metadata filtering. However, because GCS provides only key-value lookup, insert, and update operations, we can still implement an efficient shim by storing all metadata in the primary database. The shim interposes on all GCS write operations to create a primary database record containing the key, `beginTxn`, and `endTxn`. It then stores the key and value in GCS, appending `beginTxn` to the key. With this metadata, the shim interposes on all GCS read operations to first check the primary database to find the appropriate key version, then access the key. Because each read is to only one key and thus requires only one primary database lookup and because the latency of GCS is high compared to that of the primary database, these metadata operations are efficient.

5.5 MySQL

We implement a secondary shim for the relational DBMS MySQL to demonstrate that Epoxy can efficiently support distributed transactions not only for non-transactional stores but also for relational DBMSs. Our MySQL secondary shim adds two integer columns, `beginTxn` and `endTxn`, to all client-defined tables and interposes on write operations to update these columns as described in §3.2. Epoxy requires every record to have a uniquely identifiable key, so we expect every MySQL table to have a specified column with unique values. However, as discussed in §4.3, we do not currently support enforcing other key constraints in MySQL. To interpose on read operations, we substitute every table in a query with a subquery filtering the table using the predicates described in §3.2.

To improve MySQL performance, we apply two optimizations to our MySQL shim. First, we create indexes on the `beginTxn` and `endTxn` columns to speed up operations on record versions. Second, we leverage the native transactional capabilities of MySQL to reduce the number of disk writes required by Epoxy transactions. We use the default MySQL transaction isolation level (repeatable read). When an Epoxy transaction accesses MySQL, it begins a MySQL transaction internally to perform all queries and updates. After the transaction finishes validation, it commits on MySQL. Thus, a committed Epoxy transaction must only perform a durable write to disk once. If an Epoxy transaction must abort before the MySQL transaction commits, it simply rolls back the MySQL transaction. Otherwise, it executes a new MySQL transaction undoing the previous one, following the procedure in Algorithm 1. This undo transaction is guaranteed to succeed because changes committed in MySQL but not Epoxy are protected by Epoxy concurrency control, so concurrent transactions may not see them or conflict with them. We note that this optimization is general and can apply to other data stores providing interactive transactions.

6 EVALUATION

We evaluate Epoxy with TPC-C and microservice workloads by adapting them to a multi-data store setting. We compare Epoxy to a baseline that provides no transactional guarantees (and exhibits anomalies) and to an XA-based transaction manager that provides transactional atomicity but not isolation. We also analyze Epoxy performance with microbenchmarks. We show that:

(1) Epoxy provides comparable performance to but stronger transactional guarantees, including isolation, than an XA-based transaction manager on multi-DBMS TPC-C.
(2) Epoxy provides transactional guarantees for multi-data store microservices and eliminates concurrency anomalies with overhead compared to a no-transactions baseline of <10% on read-mostly workloads and 72% on write-heavy workloads.
(3) On microbenchmarked point operations, Epoxy provides transactional guarantees while adding overhead of <20% for reads, <76% for inserts, and <249% for updates compared to a no-transactions baseline across all data stores.

6.1 Experimental Setup

We implement each Epoxy shim in <1K lines of Java code. For our experiments, we use Postgres 14.2, MongoDB Community Server 5.0.9, Elasticsearch 8.2.0, and MySQL Community Server 8.0.30.

Where not otherwise noted, we run on Google Cloud using c2-standard-8 VM instances with 8 vCPUs, 32GB DRAM, and a SCSI HDD. In experiments involving multiple data stores, we run each data store in single-node mode on its own server (except GCS, which is accessed through its cloud API).

6.2 Baselines

No Transactions. To measure the absolute overhead of Epoxy, we use a baseline that executes our benchmark workloads with no cross-data store transactional guarantees. We run this baseline in a setup identical to that of Epoxy, but do not store additional versioning metadata or index structures, and execute operations separately on each data store without coordinating them.

XA-Based Transaction Manager. For cross-DBMS transactions between Postgres and MySQL, we use the XA-based transaction manager Bitronix [6] as a baseline. This baseline provides transactional atomicity, but not isolation. We run it in a setup identical to the no-transactions baseline, but coordinate cross-DBMS transactions with the Bitronix transaction manager implementing the role of coordinator in the standard two-phase commit protocol as specified in the Java Transaction API (JTA 1.1) [22]. The coordinator is colocated with the client and writes commit and abort decisions to disk for durability.

6.3 Experimental Workloads

We benchmark Epoxy using multi-data store TPC-C and microservice workloads. Each workload exhibits anomalies if run without cross-data store transactional guarantees. Epoxy provides cross-data store transactions that eliminate these anomalies.

Multi-DBMS TPC-C. For our first benchmark, we adapt the NewOrder and Payment transactions from TPC-C [11] to a multi-DBMS setting. We choose these two transactions because they comprise 90% of the TPC-C workload. The NewOrder transaction models customers placing orders on their local district of a warehouse. The Payment transaction models making payments on orders. Both transactions may access items in multiple warehouses. To simulate a scenario in which warehouse data is stored in different geographic locations, we partition the database containing the 40 warehouses so that half the warehouses are stored in MySQL and the other

half in Postgres. TPC-C tables have composite primary keys, so in Epoxy experiments we add a unique string column to each table containing concatenated key values and use its values as Epoxy keys. We execute a workload of 50% NewOrder and 50% Payment, running both as multi-DBMS transactions across the warehouse information in Postgres and MySQL.

Hotel. Our second benchmark simulates a hotel reservation workload. It consists of a room availability service that stores data in Postgres and a customer reservation service that stores data in MongoDB, similar to the example in Figure 3. Our workload consists of 80% searches for available rooms, performing a read in Postgres and a geospatial search in MongoDB, and 20% room reservations, performing a read and update in Postgres and an insert in MongoDB. Without Epoxy, these operations do not occur atomically and are not isolated, causing anomalies as discussed in §2.2. We initialize the benchmark with 100 hotels.

Cart. Our third benchmark is an e-commerce service simulating an online marketplace. The service stores customer shopping carts and an item catalog in Postgres and replicates the catalog to Elasticsearch for rapid search. We run a workload of 90% searching and adding items, performing a search for an item in Elasticsearch and a read, insert, and update to add the item to a cart in Postgres; 8% checkouts, performing a read, delete, and two inserts to move items from a cart to an order table in Postgres; 1% catalog inserts, inserting a new item in Postgres and Elasticsearch; and 1% catalog updates, updating an item in Postgres and Elasticsearch. Without Epoxy, this service exhibits what Laigner et al. [17] term "feral ordering" for concurrent catalog and cart operations. For example, if a search and add for an item below a certain price happens concurrently with a catalog update that increases items' prices, an item may be added to a customer's shopping cart despite it being more expensive than what the customer searched for. We initialize the benchmark with 10M items.

Profile. Our fourth benchmark simulates a social network where user profiles are stored in Postgres but user profile images are stored in GCS. We run a workload of 90% profile reads, consisting of reads in Postgres and GCS; 5% profile inserts, adding a profile to Postgres and uploading an image to GCS; and 5% profile updates; updating a profile in Postgres and replacing its image in GCS. Without Epoxy, this exhibits fractured reads, a common problem in production systems [17], where if a read and update to a profile occur simultaneously, the read may see the new profile image but not its accompanying profile change, or vice versa. We initialize the benchmark with 10K profiles. Images are on average 1MB in size.

Many-Data Store Benchmark. All previous benchmarks perform transactions across two systems, so to show Epoxy can support more we implement a synthetic benchmark performing transactions across Elasticsearch, MongoDB, and Postgres. This benchmark stores information on items, managing item inventory in Postgres, item pricing in MongoDB, and item descriptions in Elasticsearch. We run a workload of 99% reads, reading an item's properties in each system, and 1% updates, updating an item's properties in each system. Without Epoxy, this exhibits fractured reads, where a read may reflect a concurrent update in some systems but not others. We initalize the benchmark with 10M items in each system.

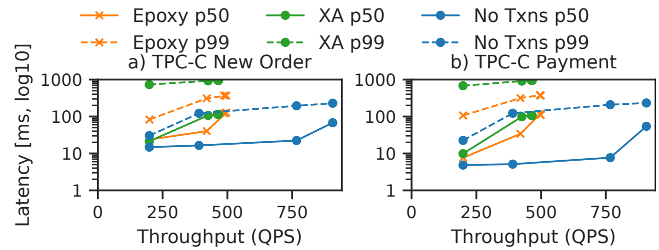

Figure 7: Throughput versus p50 and p99 latency of Epoxy, XA, and a no-transactions baseline on TPC-C.

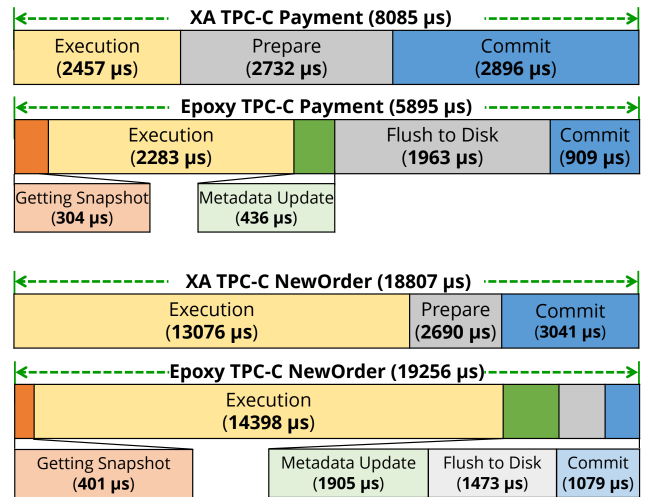

Figure 8: Latency breakdowns of the TPC-C Payment and NewOrder transactions for XA and Epoxy.

6.4 Multi-DBMS TPC-C

We first evaluate the performance of Epoxy on the multi-DBMS TPC-C workload described in §6.3. We run a 1:1 mixture of NewOrder and Payment transactions, observing p50 and p99 latency as we vary offered load. We compare with both XA, which provides atomicity but not isolation, and a no-transactions baseline, which provides neither atomicity nor isolation. We show results in Figure 7.

Epoxy provides 7% higher throughput than XA with comparable latency despite offering stronger guarantees such as isolation. However, Epoxy imposes 53% space overhead from maintaining indexed version columns in each table. Both Epoxy and XA add substantial (82–95%) overhead compared to a no-transactions baseline.

To further investigate the performance of XA and Epoxy, we break down the latency of TPC-C transactions in Figure 8. Epoxy spends 11-24% more time than XA executing transaction business logic because it must maintain versioning metadata and index structures. However, XA has expensive prepare and commit phases that require multiple rounds of communication following the participant protocol of two-phase commit, while Epoxy can simply commit on MySQL then commit on Postgres. XA prepare takes 39-83% longer than a MySQL commit (which Epoxy uses to make MySQL data durable before committing) and XA commit takes 2.8-3.2× longer

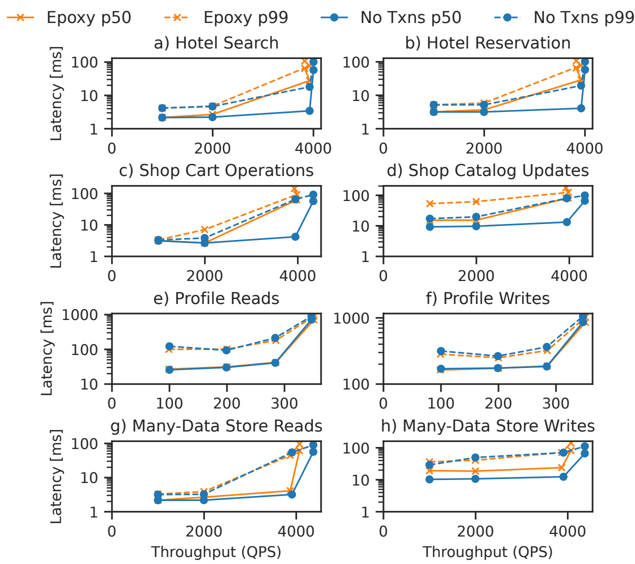

Figure 9: Throughput versus p50 and p99 latency of Epoxy and a no-transactions baseline on end-to-end microservice workloads and a many-data store synthetic benchmark.

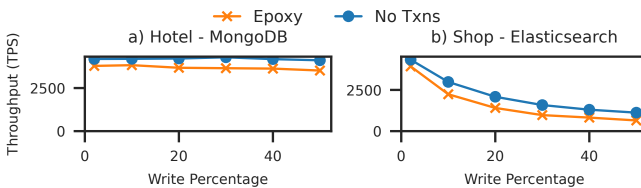

Figure 10: Write fraction versus maximum throughput of Epoxy and a no-transactions baseline.

than an Epoxy commit. Epoxy also incurs overhead constructing a transaction snapshot, but this is <5% of runtime.

6.5 End-to-end Microservice Benchmarks

We next evaluate Epoxy on the microservice workloads and many-data store synthetic (MDSS) benchmark described in §6.3. We use only the no-transactions baseline as none of the data stores in these experiments support XA. We observe p50 and p99 latency while varying offered load, showing results in Figure 9.

We find that the throughput and read latency overhead of Epoxy compared to the no-transactions baseline is 5% for Hotel, 10% for Shop, not statistically significant for Profile, and 7% for MDSS. For write latency, overhead is 4-90%. Epoxy performs better relative to a no-transactions baseline on these workloads than on multi-DBMS TPC-C because they are read-heavy (as the secondary stores we evaluate are designed for read-heavy workloads) and the overhead of Epoxy is lower for reads than for writes, as we show in §6.6.

To analyze the performance of Epoxy for write-heavy workloads, we vary the write fraction of Hotel and Shop and measure the impact on maximum throughput for Epoxy and the baseline, showing results in Figure 10. We find that for both workloads, overhead increases with write fraction. For the Hotel workload, where all writes are inserts, overhead increases from 10% at 2% writes to 17% at 50% writes. For the Shop workload, where writes are an equal mixture of inserts and updates, overhead increases from 10% at 2% writes to 72% at 50% writes. Thus, we see that Epoxy overhead is higher for write-heavy workloads than read-mostly workloads, and is higher for updates than inserts. We examine this more in §6.6.

6.6 Microbenchmark Analysis

To break down the performance of Epoxy, we analyze microbenchmarks on each secondary store. We insert 1M records into each store (10K in GCS), then evaluate the performance of point reads, inserts, and updates with and without Epoxy, observing p50 and p99 latency as we vary offered load. We compare with the no-transactions baseline. In each microbenchmark, we perform a point operation in the secondary store but no operation in the primary database. When using Epoxy, we coordinate this point operation using an empty transaction in the primary database (except for reads where we apply the optimization from §3.2). We show results in Figure 11.

We find overhead is lowest for GCS (5% for reads, not significant for inserts or updates) because the cost of GCS data operations is far higher than the overhead of the metadata operations used in transactions. For the other three systems, overhead is 3-21% for point reads, 25-76% for point inserts, and 120-249% for point updates. Overhead is higher for updates than for reads and inserts because performing an update in Epoxy requires not only creating a new record version but also updating the `endTxn` field in the most recent older record version. Overhead is higher for microbenchmarks than end-to-end benchmarks because end-to-end benchmarks perform more complex operations, amortizing Epoxy overhead.

We further investigate Epoxy overhead in Figure 12, breaking down the performance of inserts and updates in MongoDB. We observe similar trends for Elasticsearch and MySQL. We find that insert overhead comes largely from checking that no record already exists with the key to be inserted. Update overhead comes, as expected, from updating the `endTxn` field of the most recent older record version. There is also a small amount of overhead (not shown) from coordination in the primary database.

Write Conflicts. We next use a microbenchmark to analyze the impact of write conflicts on performance. We store a varying number of key-value pairs in Postgres and MongoDB. We then run a workload of 50% reads and 50% updates, where reads read and updates update the same uniformly random key in both systems. We run this workload in Epoxy and the no-transactions baseline, varying the size of the key space (and thus the frequency of write conflicts) and observing maximum achievable throughput. We show results in Figure 13. Performance of both Epoxy and the baseline decreases as the size of the key space decreases and the frequency of write conflicts increases. With 100K keys, conflicts are near zero and Epoxy is 1.5× slower than the baseline. With 100 keys, 78.8% of Epoxy transactions and 10.7% of baseline Postgres transactions abort due to a write conflict with a concurrent transaction on the same key and Epoxy is 1.9× slower than the baseline. With 10 keys, 98.8% of Epoxy transactions and 57.4% of baseline Postgres transactions abort and Epoxy is 3× slower than the baseline.

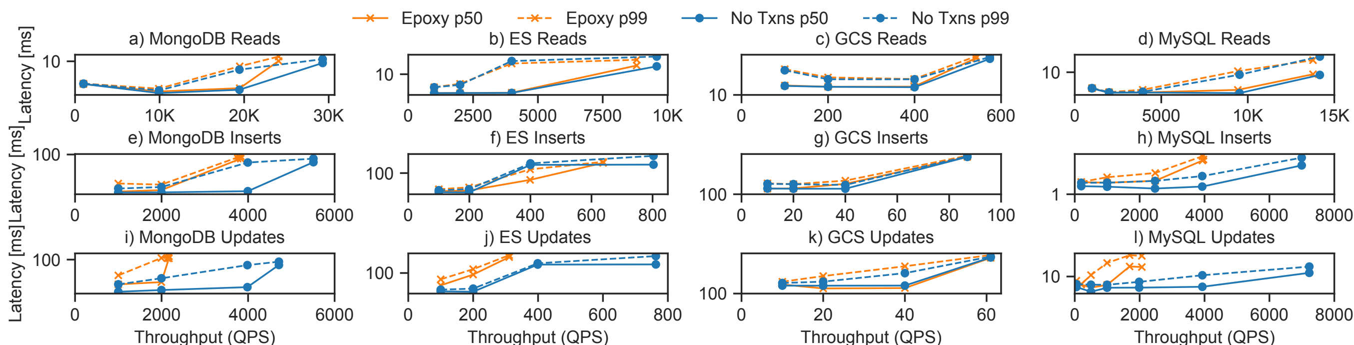

Figure 11: Throughput versus p50 and p99 latency of Epoxy and a no-transactions baseline on microbenchmarks.

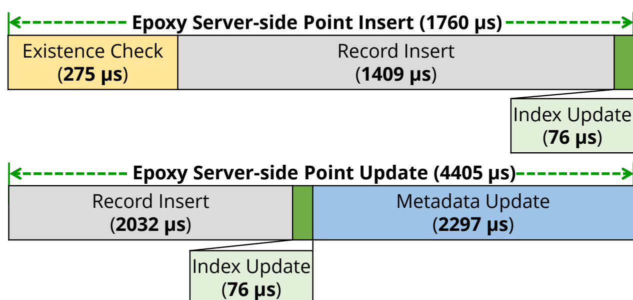

Figure 12: Performance breakdowns of Epoxy point inserts and point updates in MongoDB.

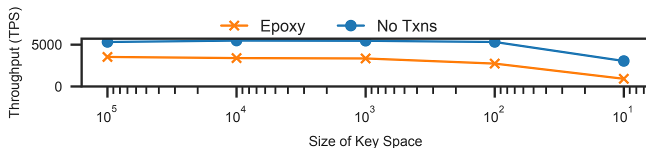

Figure 13: For a workload of 50% reads and 50% updates on Postgres and MongoDB, size of key space versus throughput for Epoxy and a no-transactions baseline.

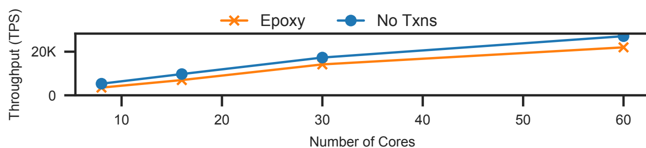

Figure 14: For a workload of 50% reads and 50% updates on Postgres and MongoDB, number of cores on data store servers versus throughput for Epoxy and a no-transactions baseline.

Scalability. We also use a microbenchmark to assess Epoxy's scalability. We store 1M key-value pairs in Postgres and MongoDB, then run the same workload of 50% reads and 50% updates as in the write conflicts benchmark while scaling both data store servers from 8 to 60 cores. We show results in Figure 14. We find Epoxy scales similarly to the no-transactions baseline. With 8 cores on each database server, Epoxy achieves 3.5K TPS and the baseline achieves 5.3K TPS, a difference of 1.5×. With 60 cores, Epoxy achieves 22K TPS and the baseline achieves 27K TPS, a difference of 1.2×.

Table 1: Compared to existing cross-data store transactions protocols, Epoxy provides ACID guarantees while supporting heterogeneous and potentially non-transactional stores.

	A	C	I	D	Supported Data Stores
XA [34]	✓	✓	X	✓	Transactional DBs.
WS-TX [21]	✓	✓	X	✓	Web services.
Cherry Garcia [12]	✓	✓	✓	✓	KV Stores.
Skeena [35]	✓	✓	✓	✓	Engines inside the same DB.
Epoxy	✓	✓	✓	✓	Any store that provides metadata filtering and durable writes (§2.1).

Storage Overhead. Finally, we measure Epoxy storage overhead. Using Epoxy, we create in MongoDB a collection of 1M documents, each containing ten integers, one randomized ten-character string, and a unique string key. Without Epoxy, this collection consumes 118 MB on disk, on average 118 bytes per document. With Epoxy, this collection consumes 136 MB on disk, on average 136 bytes per document. Thus, Epoxy adds storage overhead of ~18 bytes per document, which is reasonable as Epoxy adds to each document two `long` fields (`beginTxn` and `endTxn`) and creates an index on `beginTxn`. We observe similar results in MySQL and Elasticsearch.

7 RELATED WORK

Cross-Data Store Transaction Protocols. We summarize the differences between Epoxy and existing cross-data store transaction protocols in Table 1. Conventionally, cross-data store transactions are implemented through a distributed transaction protocol such as X/Open XA [34] or WS-TX [21]. Such protocols use two-phase commit, requiring participating systems to implement its participant protocol. They provide atomicity but not isolation; for example, it is possible for a committed transaction to be visible to an active transaction in one system but not another. Supporting a distributed transaction protocol requires modifying database internals to support two-phase commit and is thus only possible in a transactional database. By contrast, Epoxy does not require modifying systems and supports diverse non-transactional data stores.

The recent Cherry Garcia protocol [12] provides ACID transactions across heterogeneous key-value stores. Like Epoxy, Cherry Garcia supports non-transactional data stores and does not require modification of participating systems. However, unlike Epoxy, Cherry Garcia supports only key-value operations (reads from and writes to specific keys) but not other operations such as searches or aggregations. This is because Cherry Garcia requires the transaction manager to also do data management, storing uncommitted writes in a local key-value cache, redirecting read queries to read from the cache, then merging the cache into the key-value store at commit time. By contrast, Epoxy shims manage data in the secondary store and are transparent to its data model, interposing on writes only to version records and interposing on reads only to filter which record versions they see. Omid [15] is a similar protocol that, like Cherry Garcia, supports only key-value operations.

Skeena [35] provides transactions across multiple engines in the same database. It provides isolation by ensuring sub-transactions of different cross-engine transactions follow the same start order in each engine. Thus, unlike Epoxy, it requires engines natively support snapshot isolation and modifies engines to consult Skeena's snapshot registry to choose the appropriate snapshot.

Many-Database Systems. Epoxy builds on a rich literature on many-database systems, including federated databases [27] and polystores [30]. Supporting transactions across in these systems is considered an important research challenge [29].

Federated databases [27] such as MYRIAD [16] or super-databases [26] require participating data stores to implement the participant protocol of two-phase commit for atomicity (unlike Epoxy) and cannot provide transactional isolation without imposing strong requirements on participating data stores (for example, the ticket method [14] requires all participating stores to provide serializable transactions). Breitbart et al. [7] propose alternatives to two-phase commit for achieving atomicity, such as redoing aborted transactions either by retrying them until they succeed or by installing their writes directly; however, the former is not guaranteed to succeed while the latter violates isolation. Polypheny-DB [32] implements transactions using strict two-phase locking, so, unlike Epoxy, it can only provide transactions if the underlying data stores offer transaction support. Recently, Faria et al. [13] proposed a transactional polystore protocol based on MVCC where changes to a table are stored in local caches until they are visible to all queries, then are merged into the table. However, unlike Epoxy, this protocol requires data stores to provide complex operators such as a left anti-join to integrate cached data into query results; it also assumes all writes are done atomically at commit time and does not allow transactions to read their own writes.

Bolt-on Transactions. Database researchers have developed many protocols for providing strong guarantees as bolt-on properties for existing data stores. The decision to architecturally separate transactional safety from data management in Epoxy was influenced by bolt-on causal consistency [4], which uses a shim layer to provide causal consistency to an eventually consistent data store.

One system related to Epoxy is Percolator [24], which bolts ACID transactions onto Bigtable [10] using MVCC with two-phase locking. Like Epoxy, Percolator stores MVCC metadata with data to facilitate transactions, but while Epoxy provides cross-database transactions and supports many diverse data stores, Percolator is designed exclusively for search indexing on Bigtable. Percolator trades off latency for scale; its lazy lock management can add tens of seconds of latency to transactions but eliminates the need for a central lock manager. Epoxy by contrast uses a transactional DBMS as a central transaction coordinator.

Deuteronomy [19] proposes decomposing a database into separate data and transaction components, a separation of transaction and data management concerns analogous to Epoxy's architecture. The Deuteronomy transaction component provides serializable ACID transactions using MVCC with timestamp ordering. However, it is co-designed with the Deuteronomy data component and its dedicated version manager; by contrast, Epoxy shims can bolt on to diverse and potentially non-transactional data stores.

Middleware systems automatically distribute data and queries across multiple data stores, providing fault-tolerant replication [8, 23] with guarantees like snapshot isolation [20]. However, these systems provide distributed capabilities across multiple single-node data stores of the same type (e.g., multiple MySQL instances [9]), not transactions across heterogeneous data stores like Epoxy.

Atomic Commit Protocols. Atomic commit protocols guarantee that the participants in a transaction either all commit or all abort. The most popular atomic commit protocol is two-phase commit, discussed earlier. The Epoxy commit protocol is related to one-phase commit protocols, which make strong assumptions about how participating stores manage data so they can validate a store is ready without an explicit preparation phase [1]. The coordinator log protocol [28] requires each participating database send its log records to the coordinator so it can recover them in case of failure, but this is not practical for heterogeneous stores that cannot manage each other's log records. The implicit yes-vote protocol [3] assumes every database employs strict two-phase locking for concurrency control and forces redo log writes after every operation so that they are guaranteed to be ready to commit after all data operations are complete, but many data stores do not meet these assumptions. The Epoxy commit protocol can determine if a participant is ready without making these assumptions because it instead assumes all participants provide durable writes and use Epoxy concurrency control: a participant is ready if all its operations are complete and durable and have passed validation.

8 CONCLUSION

In this paper, we have described Epoxy, a protocol for providing ACID transactions across heterogeneous data stores. Epoxy makes two contributions: an adaptation of MVCC to a cross-data store setting to provide isolation and a commit protocol providing atomicity without requiring data stores to implement the participant protocol of two-phase commit. Epoxy can be implemented by any data store satisfying two basic requirements: it must support record metadata filtering and provide durable writes.

REFERENCES

[1] Maha Abdallah, Rachid Guerraoui, and Philippe Pucheral. 1998. One-phase commit: does it make sense?. In *Proceedings 1998 International Conference on Parallel and Distributed Systems (Cat. No. 98TB100250)*. IEEE, 182–192.

[2] Atul Adya. 1999. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions.* Ph. D. Dissertation. Massachusetts Institute

of Technology, Dept. of Electrical Engineering and ....

[3] Y Al-Houmaily and Panos K Chrysanthis. 1996. The implicit-yes vote commit protocol with delegation of commitment. In *Proc. of 9th Intl. Conf. on Parallel and Distributed Computing Systems*. Citeseer.

[4] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 761–772. https://doi.org/10.1145/2463676.2465279

[5] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency control and recovery in database systems*. Vol. 370. Addison-wesley Reading.

[6] bitronix authors. 2022. https://github.com/bitronix/btm

[7] Yuri Breitbart, Hector Garcia-Molina, and Avi Silberschatz. 1992. Overview of Multidatabase Transaction Management. In *VLDB Journal 1*. 181–239.

[8] Emmanuel Cecchet, George Candea, and Anastasia Ailamaki. 2008. Middleware-Based Database Replication: The Gaps between Theory and Practice. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 739–752. https://doi.org/10.1145/1376616.1376691

[9] Emmanuel Cecchet, Marguerite Julie, and Willy Zwaenepoel. 2004. C-JDBC: Flexible Database Clustering Middleware. In *USENIX Annual Technical Conference*.

[10] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2, Article 4 (jun 2008), 26 pages. https://doi.org/10.1145/1365815.1365816

[11] Transaction Processing Performance Council. 2005. Transaction processing performance council. *Web Site, http://www. tpc. org* (2005).

[12] Akon Dey, Alan Fekete, and Uwe Röhm. 2015. Scalable distributed transactions across heterogeneous stores. In *2015 IEEE 31st International Conference on Data Engineering*. 125–136. https://doi.org/10.1109/ICDE.2015.7113278

[13] Nuno Faria, José Pereira, Ana Nunes Alonso, and Ricardo Vilaça. 2021. Towards Generic Fine-Grained Transaction Isolation in Polystores. In *Heterogeneous Data Management, Polystores, and Analytics for Healthcare*. Springer International Publishing, Cham, 29–42.

[14] D. Georgakopoulos, M. Rusinkiewicz, and A. Sheth. 1991. On serializability of multidatabase transactions through forced local conflicts. In *[1991] Proceedings. Seventh International Conference on Data Engineering*. 314–323. https://doi.org/10.1109/ICDE.1991.131479

[15] Daniel Gómez Ferro, Flavio Junqueira, Ivan Kelly, Benjamin Reed, and Maysam Yabandeh. 2014. Omid: Lock-free transactional support for distributed data stores. In *2014 IEEE 30th International Conference on Data Engineering*. 676–687. https://doi.org/10.1109/ICDE.2014.6816691

[16] S.-Y. Hwang, E.-P. Lim, H.-R. Yang, S. Musukula, K. Mediratta, M. Ganesh, D. Clements, J. Stenoien, and J. Srivastava. 1994. The MYRIAD Federated Database Prototype. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data* (Minneapolis, Minnesota, USA) *(SIGMOD '94)*. Association for Computing Machinery, New York, NY, USA, 518. https://doi.org/10.1145/191839.191986

[17] Rodrigo Laigner, Yongluan Zhou, Marcos Antonio Vaz Salles, Yijian Liu, and Marcos Kalinowski. 2021. Data Management in Microservices: State of the Practice, Challenges, and Research Directions. *Proc. VLDB Endow.* 14, 13 (sep 2021), 3348–3361. https://doi.org/10.14778/3484224.3484232

[18] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *Proc. VLDB Endow.* 5, 4 (dec 2011), 298–309. https://doi.org/10.14778/2095686.2095689

[19] Justin Levandoski, David Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. 2015. High Performance Transactions in Deuteronomy. In *Conference on Innovative Data Systems Research (CIDR 2015)*. https://www.microsoft.com/en-us/research/publication/high-performance-transactions-in-deuteronomy/

[20] Yi Lin, Bettina Kemme, Marta Patiño Martínez, and Ricardo Jiménez-Peris. 2005. Middleware Based Data Replication Providing Snapshot Isolation. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data* (Baltimore, Maryland) *(SIGMOD '05)*. Association for Computing Machinery, New York, NY, USA, 419–430. https://doi.org/10.1145/1066157.1066205

[21] Oasis. 2009. Web Services Atomic Transaction (WS-AtomicTransaction). https://docs.oasis-open.org/ws-tx/wsat/2006/06

[22] Oracle. 2022. Java Transaction API (JTA). https://www.oracle.com/java/technologies/jta.html

[23] Marta Patiño Martinez, Ricardo Jiménez-Peris, Bettina Kemme, and Gustavo Alonso. 2005. MIDDLE-R: Consistent Database Replication at the Middleware Level. *ACM Trans. Comput. Syst.* 23, 4 (nov 2005), 375–423. https://doi.org/10.1145/1113574.1113576

[24] Daniel Peng and Frank Dabek. 2010. Large-scale incremental processing using distributed transactions and notifications. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*.

[25] Postgres. 2022. System Information Functions and Operators. https://www.postgresql.org/docs/current/functions-info.html

[26] C. Pu. 1988. Superdatabases for composition of heterogeneous databases. In *Proceedings. Fourth International Conference on Data Engineering*. 548–555. https://doi.org/10.1109/ICDE.1988.105502

[27] Amit P. Sheth and James A. Larson. 1990. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Comput. Surv.* 22, 3 (sep 1990), 183–236. https://doi.org/10.1145/96602.96604

[28] James W Stamos and Flaviu Cristian. 1993. Coordinator log transaction execution protocol. *Distributed and Parallel Databases* 1, 4 (1993), 383–408.

[29] Michael Stonebraker. 2015. The Case for Polystores. https://wp.sigmod.org/?p=1629

[30] Ran Tan, Rada Chirkova, Vijay Gadepally, and Timothy G. Mattson. 2017. Enabling query processing across heterogeneous data models: A survey. In *2017 IEEE International Conference on Big Data (Big Data)*. 3211–3220. https://doi.org/10.1109/BigData.2017.8258302

[31] Chuzhe Tang, Zhaoguo Wang, Xiaodong Zhang, Qianmian Yu, Binyu Zang, Haibing Guan, and Haibo Chen. 2022. Ad Hoc Transactions in Web Applications: The Good, the Bad, and the Ugly. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 4–18. https://doi.org/10.1145/3514221.3526120

[32] Marco Vogt, Nils Hansen, Jan Schönholz, David Lengweiler, Isabel Geissmann, Sebastian Philipp, Alexander Stiemer, and Heiko Schuldt. 2021. Polypheny-DB: Towards Bridging the Gap Between Polystores and HTAP Systems. In *Heterogeneous Data Management, Polystores, and Analytics for Healthcare*. Springer International Publishing, Cham, 25–36.

[33] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *Proc. VLDB Endow.* 10, 7 (mar 2017), 781–792. https://doi.org/10.14778/3067421.3067427

[34] X/Open. 1991. Distributed Transaction Processing: The XA Specification.

[35] Jianqiu Zhang, Kaisong Huang, Tianzheng Wang, and King Lv. 2022. Skeena: Efficient and Consistent Cross-Engine Transactions. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 34–48. https://doi.org/10.1145/3514221.3526171