Efficiently Making Cross-Engine Transactions Consistent

Jianqiu Zhang[1]* Kaisong Huang[1] Tianzheng Wang[1] King Lv[2]
[1]Simon Fraser University [2]Huawei Cloud Database Innovation Lab
{jianqiuz, kha85, tzwang}@sfu.ca {lvjinquan}@huawei.com

ABSTRACT

Database systems are becoming increasingly multi-engine. In particular, a main-memory engine may coexist with a traditional storage-centric engine in a system to support various applications. It is desirable to allow applications to access data in both engines using *cross-engine* transactions. But existing systems are either only designed for single-engine accesses, or impose many restrictions by limiting cross-engine transactions to certain isolation levels and operations. The result is inadequate cross-engine support in terms of correctness, performance and programmability.

This paper describes Skeena, a holistic approach to cross-engine transactions. We propose a lightweight snapshot tracking structure and an atomic commit protocol to efficiently ensure correctness and support various isolation levels. Evaluation results show that Skeena maintains high performance for single-engine transactions and enables cross-engine transactions which can improve throughput by up to 30× by judiciously placing tables in different engines.

1 Introduction

Traditional database engines are storage-centric: they assume data is storage-resident and optimize for storage accesses. Modern database servers often feature large DRAM that fits the working set or entire databases, enabling memory-optimized database engines [10, 13, 14, 22] that perform drastically better with lightweight concurrency control, indexing and durability designs.

Now suppose you are a database systems architect, and inspired by recent advances, built a new memory-optimized engine. But soon you found it was difficult to attract users: some do not need such fast speed; some say "I want it only for some tables or part of my application." A common solution is to integrate the new engine into an existing system that initially uses a traditional engine. The result is a *multi-engine* database system (Figure 1). The application can judiciously use tables in both engines. Although engines share certain components (e.g., SQL parser), each engine is autonomous with its own indexes, concurrency control, etc. Some systems [2, 9, 18] already take this approach for easier migration and compatibility.

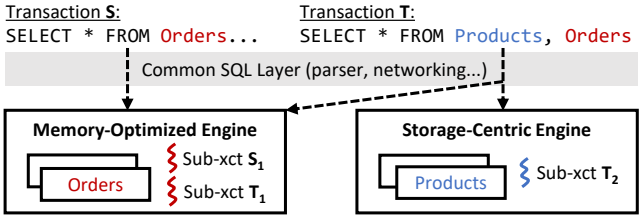

Figure 1: Multi-engine database system. Data accesses are routed to the corresponding storage engines.

1.1 Cross-Engine Transactions: Necessary but Poorly-Supported

As an experienced architect—perhaps even before users did—you realized it was necessary to support cross-engine transactions. For example, a financial application may use a memory table for fast trading and keep other data in the traditional engine for low cost; yet the user may need to access both engines for recent and historical trading data in one ACID transaction [8]. The application may use a unified SQL interface to access all engines, but since each engine is an autonomous "package," the system has to use each engine's own transaction abstractions; we refer to them as *sub-transactions*. A *transaction* consists of at least one sub-transaction. In Figure 1, S is a single-engine transaction that consists of S_1, while T is cross-engine that consists of T_1 (memory-optimized) and T_2 (storage-centric).

Cross-engine transactions can be very useful, but existing support is inadequate. First, although simply starting and committing sub-transactions suffice to support single-engine transactions, doing so does not ensure correct cross-engine execution. A transaction over two engines that both use snapshot isolation (SI) [3] can still see inconsistent snapshots. Even if both engines ensure serializability, the overall execution is not necessarily serializable. Atomicity will also be at risk if a sub-transaction fails to commit. Second, prior solutions were not designed for modern multi-engine systems, which are *fast-slow* where a (much faster) memory-optimized engine and a (much slower) storage-centric engine coexist in

*Currently with ByteDance.

a single node. It is then vital for the cross-engine solution to impose low (if any) overhead, especially on the faster engine to retain its high performance. Prior solutions [4, 6, 20, 21] ignored this hidden requirement by assuming storage-centric engines. Finally, past solutions are often at odds with (1) keeping engine autonomy for maintainability as engines are typically developed by different teams (but of the same vendor), and (2) easing application development. They often require non-trivial application changes and limit functionality, by forcing users to pre-declare whether a transaction is cross-engine or to use certain isolation levels [9]; both can be complex and affect performance.

1.2 Skeena

This paper presents Skeena, a holistic approach to efficient and consistent cross-engine transactions for multi-versioned, fast-slow systems. We make three key observations. (1) As prior work [4] noted, inconsistent snapshots can be avoided by carefully selecting a snapshot in each engine. This requires efficiently tracking snapshots that can be safely used by later transactions. (2) In addition to using correct snapshots and enforcing sub-transactions commit orders, for serializability it suffices to require each engine use commit ordering, i.e., forbid schedules where commit and dependency orders mismatch [1, 20]. Many concurrency control protocols exhibit this property, including the widely-used 2PL and optimistic concurrency control (OCC) [15]. (3) Engines are developed and/or well understood by the same vendor, potentially allowing non-intrusive changes to engines for more optimizations.

Based on these observations, we design Skeena to consist of (1) a cross-engine snapshot registry (CSR) for correct and efficient snapshot selection and (2) an extended pipelined commit protocol for atomicity and durability. Skeena can be easily plugged into an existing system.

Conceptually, CSR maintains mappings between commit timestamps (therefore snapshots) in one engine and those in another. A transaction may start by accessing any engine using the latest snapshot s. Upon accessing another engine E, it queries CSR using s to select a snapshot in E to avoid incorrect executions. With CSR, one only needs to set each engine to use a serializable protocol that exhibits commit ordering to guarantee serializability. Later, we discuss techniques that make CSR lightweight and easy to maintain.

Leveraging the fact that engines can communicate via fast shared memory (e.g., in the same address space), Skeena extends pipelined commit [12] to ensure atomicity and durability. Upon commit, the worker thread detaches the transaction and places it on a commit queue, before continuing to work on the next request. Meanwhile, a background thread monitors the queue and durable log sequence numbers in both engines to dequeue transactions whose sub-transactions' log records have been persisted. This way, Skeena ensures cross-engine transactions are not committed (i.e., with results made visible to the application) until all of its sub-transactions are committed, while avoiding expensive 2PC.

We adopted Skeena in MySQL to enable cross-engine transactions across its storage-centric InnoDB and ERMIA [14], an open-source memory-optimized OLTP engine. This required 83 LoC out of over 200k LoC in MySQL codebase. Evaluation on a 40-core server shows that Skeena retains the memory-optimized engine's high performance, and incurs very low overhead for cross-engine transactions. By judiciously placing tables in both engines, Skeena can help improve the throughput of realistic workloads by up to 30×. Skeena is open-sourced at https://github.com/sfu-dis/skeena.

Table 1: Multi-engine vs. distributed and federated systems.

	Multi-Engine	**Federated**	**Distributed**
Engine Internals	Transparent	*Opaque*	Transparent
Engine Types	Heterogeneous	Heterogeneous	*Homogeneous*
Autonomy	*Almost full*	Full	Low
Scalability	*Up/out*	Out	Out

2 Background

In this section, we give the necessary background for cross-engine transactions and motivate our work.

2.1 Modern Fast-Slow Multi-Engine Systems

Several systems already adopted the fast-slow architecture: SQL Server supports memory tables managed by its Hekaton main-memory engine [10,17]; PostgreSQL supports additional engines through foreign data wrappers, which are used by Huawei GaussDB to integrate a main-memory engine [2].

Multi-engine systems bear similarities to distributed and federated systems [4, 6, 7], but are unique in several ways. As Table 1 summarizes, a multi-engine system integrates engines developed and/or understood by the same vendor; whereas federated systems consist of opaque systems from different vendors. Distributed systems typically involve a cluster that runs the same engine. Fast-slow systems integrate different engines that vary in performance. Inefficient cross-engine support may penalize single-engine transactions, defeating the purpose of adopting a fast engine; mitigating such overhead is the major goal of our work. Multi-engine systems can also allow slightly trading autonomy for performance and compatibility, e.g., by managing schemas in all engines centrally. They can scale up and out, whereas the other two types of systems mainly focus on scaling out. We focus on single-node cases and leave scaling out as future work.

2.2 Database Model and Assumptions

Now we lay out the preliminaries for analyzing cross-engine transactions in fast-slow systems.

Multi-Versioning. Given the popularity of multi-versioning in real systems, we target multi-versioned systems in this paper. We model databases as collections of records, each of which is a totally-ordered sequence of *versions* [1]. Updating a record appends a new version to the record's sequence. Inserts and deletes are special cases of updates that append a valid and special "invalid" version, respectively. Obsolete versions (as a result of deletes/updates) are physically removed only after no transaction will need them, using reference counting or epoch-based memory management.

Reading a record requires locating a proper version. This is usually realized by maintaining a global, monotonically increasing counter that can be atomically read and incremented. In a multi-engine system, engines maintain their own timestamp counters; for now, we assume single-engine transactions and expand to cross-engine cases later. Each transaction is associated with a begin timestamp and a commit timestamp, both drawn from the counter. Upon commit, the transaction

obtains its commit timestamp that determines its commit order by atomically incrementing the counter. Each version is associated with the commit timestamp of the transaction that created it. Transactions access data using a *snapshot* (aka *read view*), which is a timestamp that represents the database's state at some point in (logical) time.

Isolation Levels. For read committed, we always read the latest committed version. SI allows the transaction to read the latest version created before its begin timestamp obtained upon transaction start or the first data access. A transaction can update a record if it can see the latest committed version. Serializability can be achieved by locking or certifiers that forbid certain dependencies [23].

Cross-Engine ACID Properties. Compared to single-engine systems, a multi-engine system must maintain ACID properties for both single- and cross-engine transactions:

- Atomicity: All the sub-transactions should eventually reach the same commit or abort conclusion, i.e., either all or none of the sub-transactions commit.
- Consistency: All transactions (single- or cross-engine) should transform the database from one consistent state to another, enforcing constraints within and across engines.
- Isolation: Changes in any engine made by a cross-engine transaction must not be visible until the cross-engine transaction commits, i.e., all sub-transactions have committed.
- Durability: Changes made by cross-engine transactions should be persisted while guaranteeing atomicity.

Enforcing cross-engine ACID requires careful coordination of sub-transactions to avoid anomalies, as we describe next.

2.3 Cross-Engine Anomalies

The relative ordering of sub-transaction begin/commit events determines correctness, as certain ordering may lead to anomalies and violate ACID requirements, as described next.

Issue 1: Inconsistent Snapshots. There are two cases where a transaction may be given an inconsistent snapshot. In Figure 2(a), S started in E_1 with snapshot 1000, and T started in E_2 with snapshot 100. Suppose another transaction in E_1 committed by incrementing E_1's timestamp counter to 3000. Then, T accesses E_1, which assigns T_1 its latest snapshot 3000, and S_2 obtains snapshot 200 in E_2. Compared to S, T sees a newer version of the database in E_1, but an older version in E_2. This would require S and T start before each other, which is impossible under SI [1]. This corresponds to the "cross" phenomenon in distributed SI (DSI) [4].

Another anomaly may make partial results visible. In Figure 2(b), T first commits T_1 with timestamp 4000. Before T_2 is committed, none of T's changes should be visible to other transactions. Meanwhile, U starts in E_2 with timestamp 250, and opens U_1: since U_1 started after T_1 committed, by definition it should see $T1$'s changes. Thus, U sees partial results: T's results are visible in U's snapshot in E_1, but not E_2. This anomaly corresponds to the serial-concurrent phenomenon in DSI [4]. Compared to inconsistent (skewed) snapshots which concern the order in which sub-transactions are opened, isolation failure arises when sub-transaction begin and commit actions are interleaved and inflict different write-read dependency orders in different engines.

Issue 2: Serializability. Even if both engines guarantee full serializability, the overall execution may not be serializable. In Figure 3(a), S and T are concurrently executing in two engines that offer serializability. Each engine runs a serializable schedule, with an anti-dependency shown in Figure 3(b).

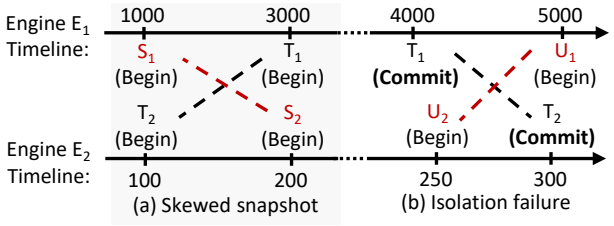

Figure 2: Inconsistent snapshots. (a) S uses an older/newer snapshot in E_1/E_2. (b) U sees T_1's results, but not T_2's.

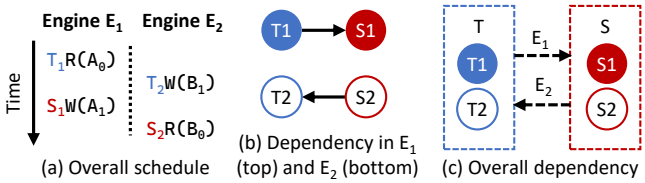

Figure 3: Non-serializable execution of cross-engine transactions. (a–b) Each engine executes a serializable schedule. (c) Overall cyclic dependency between T and S.

However, as shown in Figure 3(c), the overall execution exhibits write skew with cyclic dependencies ($T \rightarrow S \rightarrow T$), indicating non-serializable execution.

Issue 3: Atomicity and Durability. A cross-engine transaction should commit either all or none of its sub-transactions. Distributed systems usually solve this problem with 2PC, but newer engines may not support it [24]. 2PC's coordination overhead can also be heavyweight for shared memory, slowing down the (faster) main-memory engine. As we describe later, additional checks are needed in addition to a traditional 2PC prepare-commit protocol. Thus, 2PC may not be the best choice for single-node multi-engine systems.

2.4 State-of-the-Art and Motivation

Prior work can avoid the anomalies [4, 20], but did not consider fast-slow systems, leading to missing or limited cross-engine support in real systems, motivating our work. For example, MySQL supports multiple engines and users may issue multi-engine transactions, but correctness is undefined. SQL Server supports cross-engine transactions with many restrictions [9], e.g., if both the traditional engine and Hekaton use SI, cross-engine transactions are not allowed, yet SI is among the most popular isolation levels in Hekaton [9]. These significantly limit the use of cross-engine transactions.

3 Design Principles

We distill a set of desired properties and design principles that a cross-engine mechanism like Skeena should follow:

- **Low Overhead.** The mechanism should introduce as low overhead as possible. It should try not to penalize single-engine transactions, especially those in the faster engine.
- **Engine Autonomy.** Engines should be kept as-is, or only be minimally modified to work with the cross-engine mechanism or optimize for performance.
- **Full Functionality.** The mechanism should support various isolation levels for both single- and cross-engine transactions, unless it is limited by individual engine capabilities.
- **Easy Adoption.** The application should not be required

to make logic changes. It should only need to declare the "home" engine of each table in the schema.

4 Skeena Design

Skeena targets fast-slow systems with a memory-optimized and a storage-centric engine. We first give an overview of Skeena, and then discuss its design in detail.

4.1 Overview

Skeena ensures correct snapshot selection and atomic commit. As Figure 4 shows, Skeena consists of (1) the cross-engine snapshot registry (CSR) that tracks valid snapshots and (2) a pipelined commit protocol for atomically committing cross-engine transactions. Now we describe the high-level transaction workflow under Skeena.

Initialization. Transactions (single- or cross-engine) can keep using the database system's unified APIs (e.g., SQL), without additionally specifying whether a transaction will be cross-engine. Figure 4 shows an example program written in the same way as without Skeena. Skeena does not force transactions to run under specific isolation levels. However, the system may allow users to specify an isolation level (e.g., `SET TRANSACTION ISOLATION LEVEL` [18]) which can be detected and enforced by Skeena across all engines.

Data Accesses. The system routes requests to the target engine which uses a sub-transaction to access data. Skeena requires no change to the existing routing mechanism. Upon start or accessing the first record, the sub-transaction obtains a snapshot. Depending on whether the transaction is single- or cross-engine, the system may directly give the latest snapshot in the engine, or use CSR to obtain a snapshot that would not cause anomalies (steps ❷–❹). If such a snapshot does not exist, the transaction will be aborted.

Finalization. To commit, a cross-engine transaction consults CSR to verify that committing it would not lead to inconsistent snapshots for future transactions; single-engine transactions commit without using CSR. After passing verification, the transaction will further go through the pipelined commit protocol (step ❹). If the verification fails, we abort the transaction by rolling back all the sub-transactions.

Next, we describe how Skeena facilitates the above transaction workflow, beginning with CSR.

4.2 Cross-Engine Snapshot Registry

The key to avoiding inconsistent snapshots is to ensure the sub-transactions of different cross-engine transactions follow the same start order in each engine [4]. That is, if T's sub-transaction T_1 uses an older snapshot than S_1 does in engine E_1, then T_2 should also use an older snapshot compared to that of S_2 in E_2. For example, in Figure 4, T first started as a single-engine transaction accessing `Orders` in E_1, using snapshot 80. When T accesses `Products` in E_2, T needs to use a snapshot (s) in E_2 such that s is between the snapshots of its "neighbors" in E_1, i.e., S_1 and U_1. Thus, T may use any valid E_2 snapshot between 1200 and 3000 (inclusive), although using 3000 would allow it to see fresher data.

To facilitate such a snapshot selection process, CSR tracks valid snapshots (i.e., commit timestamps of past cross-engine transactions) that can be safely used by future cross-engine transactions. Conceptually, CSR is a table of many-to-many mappings, where each "row" (CSR entry) is a pair of snapshots (i.e., commit timestamps), one from each engine as

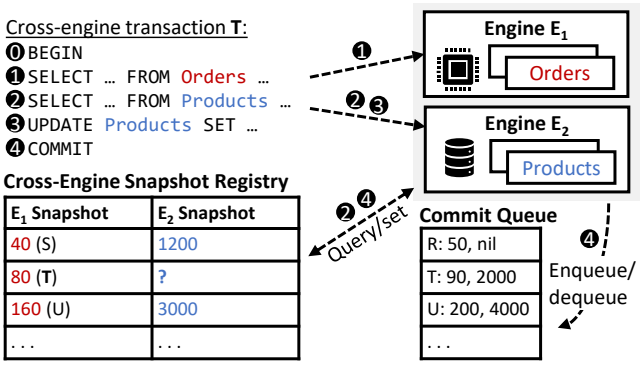

Figure 4: Skeena overview. ❶–❶ Transactions access data without explicitly declaring whether they are cross-engine. ❷ Upon accessing another engine, the transaction ❸ queries CSR for a snapshot. ❹ Cross-engine transactions use CSR for commit check and finish with the pipelined commit protocol.

Algorithm 1 Snapshot selection for cross-engine transactions.

```
1  def select_snapshot(e1_snap, engine &e2):
     # Find existing snapshots that could be used
3    candidates[] = CSR.forward_scan_1st(e1_snap)
     if candidates is empty:
5      # No existing mapping, obtain the latest from e2
       e2_snap = e2.timestamp_counter
7    else:
       # Use the latest snapshot mapped to s <= e1_snap
9      e2_snap = max(candidates)
     CSR.map(e1_snap, e2_snap)
11   return e2_snap
```

shown in Figure 4. When a transaction crosses from engine `e1` to engine `e2` as Algorithm 1 shows, it uses its snapshot in `e1` (`e1_snap`) as the key to query CSR for a snapshot in `e2`. This is done with a non-inclusive forward scan over CSR (line 3). The scan returns `e2` snapshots that are mapped to `e1_snap` or the latest snapshot before `e1_snap`. If the return set is empty, we use the latest `e2` snapshot (lines 4–6). Otherwise, we take the latest `e2` snapshot from the returned set to avoid anomalies (lines 7–9) and set up the mapping at line 10. Under SI, the algorithm is executed only once per transaction when it becomes cross-engine. Subsequent accesses continue to use the previously acquired snapshots.

In addition to acquiring snapshots, committing a cross-engine transaction implicitly limits the ranges of snapshots (future) for cross-engine transactions: the commit timestamp of a previous transaction T in fact is the snapshot of a future transaction that will read the results generated by T. Thus, CSR also sets up new mappings when committing cross-engine transactions. Similar to tracking snapshots, we ensure that committing a cross-engine transaction—i.e., adding a new mapping entry to CSR—would not add skewed snapshots to CSR. Suppose a transaction with two sub-transactions, `sub_t1` and `sub_t2`. Upon commit, we issue a reverse scan and a forward scan over CSR using the commit timestamps of a sub-transaction (e.g., `sub_t1`) to obtain the lower and higher bounds for the other commit timestamp. If `sub_t2`'s commit timestamp falls between the bounds, we can safely commit this cross-engine transaction and set up a new mapping in CSR. Otherwise, the transaction is aborted.

Since a transaction may access engines in any order (from the storage-centric engine and crosses over to the memory-optimized engine, and vice versa), CSR needs to support queries from *either* engine. CSR may be implemented using a relational table in one of the supported engines with full-table scan or two range indexes, each of which is built on a "column" of the CSR table. However, this can create dependency on a particular engine and incur much table and index maintenance overhead. A practical design must also support concurrency. We address these issues next.

4.3 Lightweight Multi-Index CSR

We take advantage of the properties of fast-slow systems to devise a lightweight CSR that mitigates the above issues.

Anchor Engine. Compared to storage-centric engines, it is much cheaper to obtain snapshots in main-memory engines. This is often as simple as manipulating an 8-byte counter in a lock-free manner without using a mutex. For example, ERMIA [14] only needs to read the counter; Hekaton [10] increments the counter using atomic fetch-and-add (`FAA`) [11]. But obtaining a snapshot in a storage-centric system can be much more complex. For example, MySQL InnoDB needs to take multiple mutexes to compute watermark values [25].

Leveraging the existence of a fast and a slow engine, Skeena designates an *anchor* engine and always follows the snapshot order in the anchor engine. The anchor engine should be the one where it is cheaper to acquire a snapshot (e.g., memory-optimized). Then a transaction always starts by acquiring the latest snapshot from the anchor engine, and uses it to query CSR later. This allows us to maintain one-to-many mappings (instead of many-to-many mappings), which simplifies CSR to become a range index that uses snapshots in the anchor engine as "keys" and lists of snapshots in the other engine as "values." We currently use Masstree [16], a fast in-memory index, but any concurrent range index would suffice. A side effect is transactions that only access the slower engine also become cross-engine. As Section 5 shows, the overhead is negligible compared to data accesses which may involve the storage stack while CSR is fully in-memory.

Multi-Index. Since CSR tracks cross-engine snapshots and commit histories, its size can grow quickly, slowing down query speed over time; entries that are no longer needed should also be cleaned up. Our solution is to partition CSR by snapshot ranges, reminiscent of multi-rooted B-trees [19]. The result is a multi-index CSR (Figure 5). Each partition is an index and covers a unique range of snapshots so that a transaction only uses a single index. In Figure 5, the first two indexes cover mappings in the ranges of [30, 400] and [401, 500], respectively. Each partition covers a fixed number of keys. A new index is created when the current open index is full. There is always one and only one open index that can accept new mappings; other indexes are read-only but can continue to serve existing transactions for snapshot selection. Since inactive indexes are read-only, a transaction that needs to set up a new mapping in an inactive index will be aborted.

Snapshot Acquisition and Commit Check. With multiple indexes and an anchor engine, a transaction acquires snapshots by (1) obtaining a snapshot S from the anchor engine, (2) locating the index I that covers S, and (3) using S to query I and if needed, create a new entry in I following Algorithm 1 with `e1_snap` = S and `CSR` at line 3 being I. Note that steps 2 and 3 are only executed if the transaction accesses the non-anchor engine. For example, if the main-memory

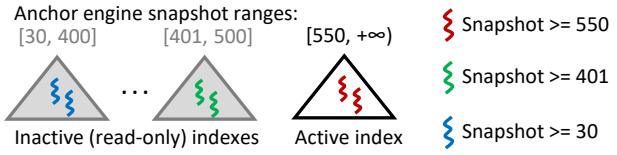

Figure 5: Multi-index CSR. Each index covers a range of anchor snapshots, and is recycled in its entirety later.

engine is the anchor and the transaction only accesses an in-memory table, steps 2 and 3 are never executed. For step 2, we track all the indexes in a list/array. Each entry records the minimum snapshot of the partition and a pointer to the index. Since we keep only one open index, entries in the list are sorted by snapshot ranges. We search for I by traversing the list backwards and stopping at the first entry whose smallest snapshot is smaller than or equal to the given snapshot. In step 3, a new entry is inserted if and only if I is open; otherwise the transaction is aborted. Commit check follows a similar logic and can proceed only if I is open.

4.4 CSR Concurrency and Maintenance

Now we discuss how Skeena handles concurrent accesses and manages/recycles indexes in multi-index CSR.

Concurrency. Although latches can be a potential bottleneck in multicore systems, a latch-based solution in Skeena can be efficient thanks to the fast-slow property: compared to executing transactions in the slower engine, using latches and high-performance indexes present negligible overhead and little impact on overall performance. Each index is protected by a mutex, and we protect the array of all indexes using a reader-writer lock for mutual exclusion between threads that only query an index without modifying the list using the reader mode and those that may add or remove an index using the writer mode. A transaction starts by latching the index list in shared (reader) mode to locate the target index I. Then the thread latches I for exclusive access to run Algorithm 1. If the thread needs to create a new index, it (1) releases the list latch, (2) re-acquires the latch in writer mode to allow inserting to the list, and (3) checks if such an index has been inserted by another thread between steps 1 and 2, and if so, retries the entire process after releasing the list latch; otherwise we (4) append the new index with a new mapping to the list. Commit check follows the same logic.

Index Maintenance. Using multiple indexes simplifies garbage collection (GC) as we can delete an entire index once its mappings are no longer needed, instead of issuing many key delete operations. To recycle, we first iterate over all the active transactions to find the oldest anchor-engine snapshot (`min_snap`). Then we remove stale indexes that cover ranges below `min_snap`. If long-running transactions prevent `min_snap` from growing, one may remove unused indexes covering newer but still unused ones, reminiscent of GCing long version chains in multi-versioned systems [5].

4.5 Commit Protocol

Once all accesses are finished, Skeena checks whether both sub-transactions can commit using engine-level commit timestamps that represent the sub-transactions' commit ordering. Some OLTP engines already break the commit process into pre- and post-commit [10, 14], making it easy to obtain commit timestamps. During pre-commit, the engine assigns a

commit timestamp and uses it to determine whether the transaction can commit correctly. If such interfaces are not available, the engines need to be modified to do so. This is straightforward by breaking a monolithic "commit" function into a pre- and post-commit function, given engines in a fast-slow system are maintained by the same vendor.

With the pre- and post-commit interfaces, Skeena commits a cross-engine transaction in three steps. (1) Pre-commit both sub-transactions to obtain commit timestamps. (2) Use the timestamp obtained from the anchor engine to conduct the commit check. (3) If the check passes, post-commit both sub-transactions. From a high-level, Skeena's commit protocol resembles 2PC: step 1 may correspond to 2PC's prepare phase that collects commit decisions from each engine; step 3 may correspond to 2PC's commit phase. But Skeena differs from 2PC by requiring an additional check (step 2) even after all the engines have pre-committed the transaction. So an "all-yes" result from the 2PC-equivalent prepare phase does not necessarily mean a cross-engine transaction can commit.

Before both sub-transactions are post-committed, changes by either should be kept invisible, yet from the perspective of an engine, a post-committed (sub-)transaction is fully committed with its results visible. Skeena must ensure partial results are not visible until all sub-transactions are post-committed. We observe that a simple yet effective solution is to extend the pipelined commit protocol [12] which was initially for hiding log flush latency. Skeena maintains a global commit queue (or a partitioned queue to avoid introducing a bottleneck) of transactions whose log records are not persisted yet. Upon commit, we first push both sub-transactions onto the commit queue upon post-commit. Results by these transactions are now visible internally but are not returned to applications until their log records are persisted. Thus, single-engine and read-only transactions must also use commit pipelining as they may read cross-engine transactions' results. A commit daemon then monitors both engines' logs to dequeue transactions once their log records are persisted.

4.6 Durability and Recovery

In a multi-engine system, each engine implements its own approach to durability and crash recovery. A sub-transaction still follows its corresponding engine's approach to persist data and log records. To ensure atomicity of cross-engine transactions, Skeena can record the pre- and post-commit of cross-engine transactions, by maintaining a standalone log or piggybacking on individual engines. During recovery, each engine executes its recovery mechanism and rolls back changes done by cross-engine transactions whose sub-transactions are not fully committed. Alternatively, the recovery procedure may inspect each engine's log and truncate at the first "hole" where only one sub-transaction of a cross-engine transaction is committed. This is safe because commit pipelining ensures results from partially committed cross-engine transactions were never made visible to applications.

4.7 Serializability

As noted by prior work [20], disallowing anti-dependencies (i.e., using commit order as dependency order) in all engines is sufficient for cross-engine serializability. This translates into choosing a concurrency control protocol for each engine where a sub-transaction can only commit if its read records are not concurrently modified by a newer transaction. A wide range of engines exhibit this property based on 2PL (by blocking readers and writers) and OCC (by verification at commit time). Some protocols [23] can tolerate certain safe anti-dependencies, but would require implementing verification in Skeena, tightly coupling Skeena with engine design and sacrificing engine autonomy. Thus, we take the former approach that imposes no engine-level changes.

4.8 Discussions

In essence, Skeena is a coordinator that enforces correct snapshots and atomic commit in fast-slow systems. Since Skeena does not implement extra concurrency control logic to avoid tight coupling with engines, to achieve an overall isolation level (e.g., SI), each engine needs to run at least at it (e.g., SI). Thus, the overall isolation level guaranteed by Skeena is at most the lowest level across all engines. For example, if two engines respectively use RC and SI, then Skeena can only guarantee RC overall.

Skeena can be applied to systems that (1) support multiple engines and (2) follow the database model in Section 2.2. Both are widely available in practice. For example, MySQL, SQL Server and PostgreSQL already support multiple engines. Moreover, Skeena does not require significant engine-level changes. The most notable (yet simple) change (mainly for conventional engines) is exposing commit ordering via a pre-commit interface. Skeena only expects the commit/abort decision of sub-transactions, without dictating engine internals, such as whether cascading abort is possible or how writes and versions are organized. The remaining effort is mainly put into integrating Skeena with existing multi-engine support. These changes are not intrusive or complex; more details can be found in the full version of this paper [25].

5 Evaluation

We empirically evaluate Skeena under microbenchmarks and realistic workloads. Through experiments, we show that:

- Skeena retains the high performance of memory-optimized engines in fast-slow systems;
- Skeena only incurs a very small amount of overhead for cross-engine transactions;
- By judiciously placing tables in different engines, Skeena can effectively improve performance for realistic workloads.

5.1 Experimental Setup

We integrated ERMIA [14], a memory-optimized engine into MySQL to co-exist with its storage-centric InnoDB. Both engines share MySQL's SQL layer and thread pool. The integration effort required $<$ 2000 LoC because MySQL already comes with well-defined interfaces for adding new engines. The application specifies each table's home engine in its schema, which is already supported by MySQL. We then modified 83 LoC in InnoDB for it to use Skeena to choose read views and commit sub-transactions. CSR itself is implemented as a separate module of ∼600LoC.For ERMIA, we modified its commit pipelining to consider both engines.

We run experiments on a dual-socket server equipped with two 20-core Intel Xeon Gold 6242R CPUs (80 hyperthreads in total), 384GB of main memory and a 400GB Micron SSD with peak bandwidth of 760MB/s. Each CPU has 35.75MB of cache and is clocked at 3.1GHz. All experiments are conducted in MySQL 8.0 with InnoDB and ERMIA under SI (repeatable read in InnoDB). We report the average throughput and latency of three 60-second runs.

ERMIA is memory-optimized so all records are in heap memory. For InnoDB, we test both the memory- and storage-resident cases: the memory-resident variant (`InnoDB-M`) uses a large enough buffer pool to avoid accessing storage; the storage-resident variant (`InnoDB`) uses a small buffer pool that would mandate accessing the storage stack. To stress test Skeena, we store persistent data (such as data files and logs) in `tmpfs`, so that I/O is as fast as memory, making it easier to expose Skeena's overhead. More experiments using SSDs can be found in the full version of this paper [25].

5.2 Benchmarks

We use microbenchmarks and TPC-C to test Skeena and explore the effect of cross-engine transactions.

Microbenchmarks. In each engine, we create 250 tables, each of which contains a certain number of records depending on whether the experiment is memory- or storage-resident for InnoDB. Each record is 232-byte, consisting of two `INTEGER`s and one `VARCHAR`. For memory-resident experiments, each table contains 25000 records, bringing the total data size of 250 tables to ∼1.35GB; the buffer pool size in InnoDB is set to 32GB. For storage-resident experiments, we set each table to contain 250000 records, and the total data size is ∼13.5GB; we set the buffer pool to be 2GB. We then devise three transaction types: read-only, read-write and write-only. Each transaction accesses ten records uniform randomly chosen from the created tables. In particular, for read-write transactions, eight out of the ten accesses are point reads and two are updates.

TPC-C. We use TPC-C for the dual-purpose of (1) testing Skeena under non-trivial transactions, and (2) exploring the benefits of cross-engine transactions in realistic scenarios. We set the scale factor to be the number of connections and each connection works on a different home warehouse, but 1% of New-Order and 15% of Payment transactions may respectively access a remote warehouse. InnoDB buffer pool size is set to be 32GB which is large enough to hold all the data (∼14GB). This way, the entire workload is memory-resident, allowing us to stress Skeena since data accesses do not involve I/O, although they are still done via the buffer pool. By gradually moving tables from InnoDB to ERMIA, we make the affected transaction cross-engine and distill several useful suggestions on how to optimize performance in fast-slow systems. We focus on the results of these suggested table placements in this paper. More experiments (including storage-resident ones and the impact of storing each table in ERMIA) can be found in the full version of this paper [25].

5.3 Single-Engine Performance

An important goal of Skeena is to ensure single-engine transactions (especially those in the faster engine, ERMIA) pay little additional cost. We evaluate this by turning Skeena on and off under six ERMIA- and InnoDB-only variants. To stress test Skeena, we use the memory-resident InnoDB (`InnoDB-M`) and the storage-resident InnoDB with `tmpfs` (`InnoDB`). Table 2 summarizes the results; variants with Skeena turned on carry an `S` suffix. Skeena incurs negligible overhead with the slightly more complex logic in commit pipelining. Note that "single-engine" transactions in `InnoDB-S`/`InnoDB-MS` are in fact cross-engine, by following the start order in the anchor engine (ERMIA) even if they do not access any records in ERMIA. This means CSR will only maintain a single mapping (using ERMIA's initial snapshot) which incurs a constant but very small amount of overhead (up to 5.6%). Compared to `InnoDB`, `InnoDB-M` performs up to over ∼ 3× better thanks to its large buffer pool. `InnoDB-M` and `InnoDB-MS` perform similarly to ERMIA under the read-only microbenchmark, but fall behind as we add more writes, signifying the benefits of memory-optimized engines. `ERMIA-S` performs as well as `ERMIA` since CSR is never used. These results verify that Skeena retains the advantage of memory-optimized engines.

Table 2: Throughput (TPS) of single-engine microbenchmarks (80 connections) and TPC-C (50 connections). Skeena (-S) incurs negligible overhead.

Scheme	Read-only	Read-write	Write-only	TPC-C
`ERMIA`	1,427,071	1,252,146	1,091,606	7,550
`ERMIA-S`	1,430,137	1,253,368	1,095,056	7,546
`InnoDB-M`	1,326,710	930,249	710,697	626
`InnoDB-MS`	1,310,809	915,406	711,425	612
`InnoDB`	456,672	420,328	194,446	277
`InnoDB-S`	453,781	420,474	194,412	261

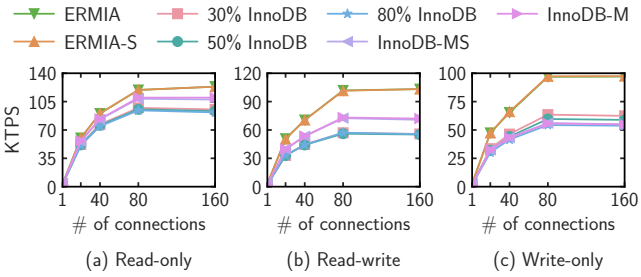

Figure 6: Throughput under memory-resident microbenchmarks. CSR cost can be comparable to data accesses, causing `InnoDB-M` to outperform cross-engine cases.

5.4 Cross-Engine Performance

Now we explore the behavior of cross-engine transactions using microbenchmarks. For each transaction, we vary the percentage of InnoDB and ERMIA accesses out of ten accesses. For example, with `30% InnoDB`, three accesses per transaction are done in InnoDB, the remaining seven accesses go to ERMIA. For cross-engine transactions we mark the percentage of InnoDB accesses and note whether the experiment is memory- or storage-resident as needed.

InnoDB is more heavyweight, so more accesses in it should lower performance, e.g., transactions with 30% InnoDB accesses should perform better than pure InnoDB transactions. But Figures 6(a)–(b) show the opposite: `InnoDB-MS` outperforms the cross-engine `30-80% InnoDB`. The reason is two-fold. First, ERMIA writes a commit log record for read-only transactions. So with more ERMIA accesses, CSR becomes larger and slower to access. This is non-negligible for read-intensive workloads, which are lightweight in ERMIA. Second, under `InnoDB-MS`, CSR is very small and only maintains one mapping. However, under `30-80% InnoDB`, more ERMIA accesses lead to more mappings in CSR, which then becomes more expensive to query. The memory-resident write-only workload follows the expectation in Figure 6(c), although the difference is not significant due to InnoDB's low raw performance. Under storage-resident workloads, Skeena's overhead

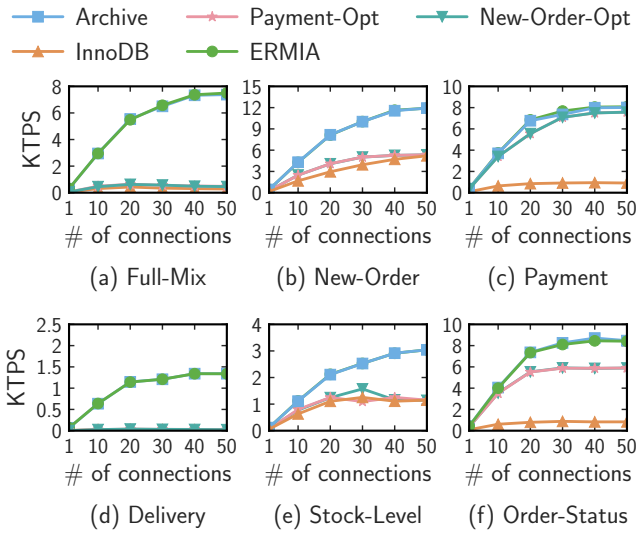

Figure 7: TPC-C throughput under select table placement schemes that optimize for different application scenarios.

becomes negligible, with more ERMIA accesses leading to higher performance: `30% InnoDB` is up to 75%/40% faster than `InnoDB` for read-only/write-only workloads.

5.5 Effect of Cross-Engine Transactions

Applications may use cross-engine transactions to improve performance (using a main-memory engine) and/or reduce storage cost with the storage-centric engine, by placing different tables in different engines. We use TPC-C to explore this aspect by analyzing several recommended table placement schemes distilled from our experiments [25]:

- `New-Order-Opt`: The `Customer` and `Item` tables are placed in ERMIA to optimize the New-Order transaction.
- `Payment-Opt`: Only `Customer` is placed in ERMIA to optimize the Payment transaction that heavily uses `Customer`.
- `Archive`: All the tables except `History` are placed in InnoDB, leveraging its cheaper storage cost.

The first two schemes improve performance with select use of main-memory tables, while `Archive` reduces storage cost of in-memory databases using a traditional engine. Figure 7 compares them to baselines that only use ERMIA or InnoDB. `New-Order-Opt` and `Payment-Opt` improve the performance of the affected transactions compared to `InnoDB`. Since `Archive` executes almost fully in ERMIA, its performance overlaps with `ERMIA` (`History` is never queried and occupies less than 600MB). In reality, such workloads can run for longer and occupy more space; placing the relevant tables in InnoDB can drastically reduce storage cost.

6 Summary

Cross-engine transactions can be very useful in modern fast-slow multi-engine systems, but were poorly supported with various limitations. This paper proposes Skeena, a holistic approach to efficient and consistent cross-engine transactions, to solve this problem. Skeena consists of a cross-engine snapshot registry (CSR) that tracks snapshots and a commit protocol for multi-engine systems. Skeena can be easily adopted by real systems, as demonstrated by our experience with MySQL. Evaluation on a 40-core server shows that Skeena incurs negligible overhead and maintains the benefits of memory-optimized engines, enabling new use cases of memory-optimized OLTP engines.

7 References

[1] A. Adya. Weak consistency: A generalized theory and optimistic implementations for distributed transactions, Mar. 1999. PhD Thesis MIT/LCS/TR-786.

[2] H. Avni et al. Industrial-strength OLTP using main memory and many cores. *PVLDB*, 13(12), Aug. 2020.

[3] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. *SIGMOD*, page 1–10, 1995.

[4] C. Binnig et al. Distributed snapshot isolation: Global transactions pay globally, local transactions pay locally. *VLDBJ*, 23(6):987–1011, Dec. 2014.

[5] J. Böttcher, V. Leis, T. Neumann, and A. Kemper. Scalable garbage collection for in-memory MVCC systems. *PVLDB*, 13(2):128–141, Oct. 2019.

[6] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. *VLDBJ*, 1(2):181–240, Oct. 1992.

[7] P. Chairunnanda, K. Daudjee, and M. T. Özsu. ConfluxDB: Multi-master replication for partitioned snapshot isolation databases. *PVLDB*, 7(11), 2014.

[8] P. Dave. SQL Server – memory optimized tables, transactions, isolation level and error, 2019.

[9] K. Delaney. SQL Server in-memory OLTP internals for SQL Server 2016. *Microsoft SQL Server Docs*, 2016.

[10] C. Diaconu et al. Hekaton: SQL Server's memory-optimized OLTP engine. *SIGMOD*, 2013.

[11] Intel Corporation. Intel 64 and IA-32 architectures software developer manuals. 2023.

[12] R. Johnson et al. Aether: A scalable approach to logging. *PVLDB*, 3(1):681–692, Sept. 2010.

[13] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. *ICDE*, pages 195–206, 2011.

[14] K. Kim, T. Wang, R. Johnson, and I. Pandis. ERMIA: Fast memory-optimized database system for heterogeneous workloads. *SIGMOD*, 2016.

[15] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM TODS*, 6(2), June 1981.

[16] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. *EuroSys*, 2012.

[17] Microsoft. *Microsoft SQL Documentation*, 2016.

[18] Oracle. MySQL 8.0 reference manual. 2021.

[19] I. Pandis, P. Tözün, R. Johnson, and A. Ailamaki. PLP: Page latch-free shared-everything OLTP. *PVLDB*, 4(10):610–621, July 2011.

[20] Y. Raz. The principle of commitment ordering, or guaranteeing serializability in a heterogeneous environment of multiple autonomous resource mangers using atomic commitment. *VLDB*, pages 292–312, 1992.

[21] R. Schenkel, G. Weikum, N. Weißenberg, and X. Wu. Federated transaction management with snapshot isolation. *Transactions and Database Dynamics*, 2000.

[22] M. Stonebraker et al. The end of an architectural era: (it's time for a complete rewrite). *VLDB*, 2007.

[23] T. Wang, R. Johnson, A. Fekete, and I. Pandis. Efficiently making (almost) any concurrency control mechanism serializable. *VLDBJ*, 26(4), 2017.

[24] E. Zamanian, C. Binnig, T. Harris, and T. Kraska. The end of a myth: Distributed transactions can scale. *PVLDB*, 10(6):685–696, Feb. 2017.

[25] J. Zhang, K. Huang, T. Wang, and K. Lv. Skeena: Efficient and consistent cross-engine transactions. *SIGMOD*, page 85–96, 2022.