Practical Cross-Engine Transactions in Dual-Engine Database Systems

by

Jianqiu Zhang

B.Sc., Northeastern University, China 2017

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

© Jianqiu Zhang 2021
SIMON FRASER UNIVERSITY
Summer 2021

Copyright in this work is held by the author. Please ensure that any reproduction
or re-use is done in accordance with the relevant national copyright legislation.

Declaration of Committee

Name: **Jianqiu Zhang**

Degree: **Master of Science**

Thesis title: **Practical Cross-Engine Transactions in Dual-Engine Database Systems**

Committee: **Chair:** Nick Sumner
Associate Professor, Computing Science

Jiannan Wang
Committee Member
Associate Professor, Computing Science

Keval Vora
Examiner
Assistant Professor, Computing Science

Tianzheng Wang
Senior Supervisor
Assistant Professor, Computing Science

Abstract

With the growing DRAM capacity and core count in modern servers, database systems are becoming increasingly multi-engine to feature a heterogeneous set of engines. In particular, a memory-optimized engine and a conventional storage-centric engine may coexist to satisfy various application needs. However, handling cross-engine transactions that access more than one engine remains challenging in terms of correctness, performance and programmability.

This thesis describes Skeena, an approach to cross-engine transactions with proper isolation guarantees and low overhead. Skeena adapts and integrates past concurrency control theory to provide a complete solution to supporting various isolation levels in dual-engine systems, and proposes a lightweight transaction tracking structure that captures the necessary information to guarantee correctness with low overhead. Evaluation on a 40-core server shows that Skeena only incurs minuscule overhead for cross-engine transactions, without penalizing single-engine transactions.

Keywords: Dual-engine database systems; cross-engine transactions

Acknowledgements

I would like to give my great thanks to Dr. Tianzheng Wang for his support of my journey to the Master's degree. Special thanks to my teammate Kaisong Huang who helped with the writing and evaluation of this work. Many thanks to the examine committee members Dr. Keval Vora, Dr. Jiannan Wang and to the chair Dr. Nick Sumner. Their valuable comments and editing suggestions help me refine the thesis. I would also like to thank my parents and my girlfriend who supported me mentally throughout this work. This thesis would not be possible without their support.

Table of Contents

Declaration of Committee **ii**

Abstract **iii**

Acknowledgements **iv**

Table of Contents **v**

List of Tables **vii**

List of Figures **viii**

1 Introduction **1**
1.1 Cross-Engine: Poorly-Supported Necessity 1
1.2 Skeena 3
1.3 Contributions and Thesis Organization 4

2 Background **5**
2.1 Modern Fast-Slow Systems 5
2.2 Database Model and Assumptions 5
2.3 Cross-Engine Anomalies and Motivation 7

3 Design Principles **10**

4 Skeena Design **11**
4.1 Overview 11
4.2 Basic Cross-Engine Snapshot Registry 13
4.3 Lightweight Multi-Index CSR 15
4.4 Commit Protocol 17
4.5 Durability and Recovery 18

5 Skeena in MySQL **20**

6 Evaluation **22**

6.1 Experimental Setup 22
6.2 Benchmarks 23
6.3 Single-Engine Performance 24
6.4 Cross-Engine Performance 26
6.5 CSR Overhead 28
6.6 Latency 29

7 Related Work **31**

8 Conclusion and Future Work **33**

Bibliography **34**

List of Tables

Table 2.1 Multi-engine vs. distributed and federated systems. 6

Table 6.1 Abort rate comparison between single-engine and cross engine TPC-C transactions 29

List of Figures

Figure 1.1 Dual-engine database system. Data accesses are routed to the corresponding storage engines. 2

Figure 2.1 Inconsistent snapshots. (a) S uses an older (newer) snapshot in E_1 (E_2). (b) U sees partial results (U_1 sees T_1's results, U_2 does not see T_2's). Both also happen under distributed SI [8], but isolation failure affects all isolation levels. 7

Figure 2.2 Non-serializable execution of cross-engine transactions when each engine guarantees serializability. (a) Each engine executes a serializable schedule (b) without cyclic dependencies. (c) Overall cyclic dependency between T and S. 8

Figure 4.1 Skeena overview. ⓿–❶ Transactions access data without explicitly declaring whether they are cross-engine. ❷ Upon accessing an additional engine, the transaction ❸ consults CSR to obtain a proper snapshot. ❹ Cross-engine transactions use CSR for commit check and if passed, goes through the pipelined commit protocol to conclude. 12

Figure 4.2 Multi-index CSR. 16

Figure 6.1 Microbenchmark. 25

Figure 6.2 Memory-resident TPC-C. 28

Figure 6.3 Storage-resident(tmpfs) TPC-C. 28

Figure 6.4 Storage-resident(ext4) TPC-C. 28

Figure 6.5 Throughput(TPS) and Abort Rate(%) of TPC-C Default Mixed Transaction with Table Scheme Optimize-payment. 29

Figure 6.6 95th percentile latency of microbenchmarks at single connection and 80 connections under varying storage-resident workloads. 30

Chapter 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 could fit the working set or entire databases, enabling memory-optimized database engines [29, 39, 54, 35, 34, 37, 22, 52, 33, 5] that perform drastically better with lightweight concurrency control, index 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 very hard 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, side-by-side with the traditional engine, resulting in a *dual-engine* database system (Figure 1.1). The application can judiciously create and access tables in both engines. Although engines share certain services (e.g., SQL parser, networking, schema management), they are autonomous, each implementing its own indexes, concurrency control, etc. Some systems (e.g., SQL Server [20], MySQL [42] and PostgreSQL [46]) already take this approach for easier migration and backward compatibility.

1.1 Cross-Engine: Poorly-Supported Necessity

As an experienced architect—perhaps even before a user did—you realized it was necessary to support cross-engine transactions that use multiple engines. For example, a finance application may use a memory table for fast trading and keep other data in the traditional engine for low cost; some transactions need to access both engines [19]. The application may use a unified SQL interface to access all engines, but internally, because of engine autonomy the system has to employ each engine's own transaction abstractions to access data; we refer to them as *sub-transactions*. A *transaction* then consists of one or multiple sub-transactions. In Figure 1.1, S is single-engine with S_1, while T is cross-engine with T_1 (memory-optimized) and T_2 (storage-centric). Cross-engine transactions can be very

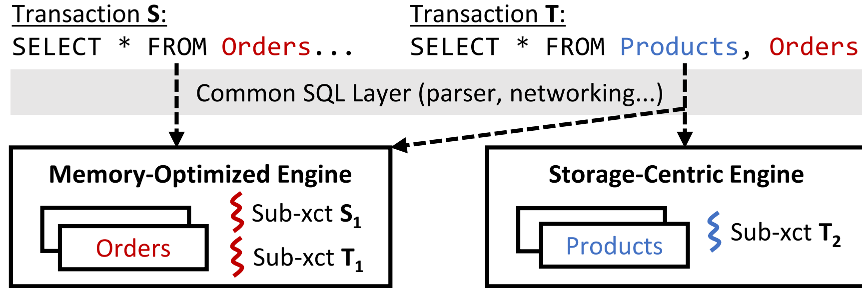

Figure 1.1: Dual-engine database system. Data accesses are routed to the corresponding storage engines.

useful, but existing support is inadequate and leaves non-trivial challenges in correctness, performance and programmability.

Although it suffices to simply start and commit sub-transactions to support single-engine transactions, doing so does not guarantee correct cross-engine execution. As we elaborate later, cross-engine transactions over two engines that both implement correct snapshot isolation (SI) [6] can still end up using inconsistent data and run under a lower-than-SI isolation level. Even if both engines guarantee serializability, the overall execution is not necessarily serializable. Simply committing sub-transactions also risks durability and atomicity if any sub-transaction commit failed. Similar issues were also found in federated and distribute systems where each "engine" is a full system. But prior solutions [47, 31, 50, 25, 18, 51, 11, 8, 48] do not consider the characteristics of modern dual-engine systems.

Modern dual-engine systems often exhibit the *fast-slow* structure where a memory-optimized engine and a conventional engine coexist in a single node. The former can outperform the latter by orders of magnitude, so it is vital for the cross-engine solution to impose low (if any) overhead, especially for the faster engine. Previous solutions were not designed in this context, by integrating systems interconnected via a network, and two-phase commit (2PC) was widely used for atomicity. However, fast-slow systems allow engines to communicate via fast shared memory and some designs explicitly avoid 2PC for scalability [62], calling for new solutions.

These problems become more challenging when it is desirable to (1) keep engine autonomy for maintainability as engines are usually developed by different teams, and (2) ease application development. Prior solutions often require non-trivial changes to the application, by requiring users to declare whether a transaction is cross-engine upon start, and forcing the application to use particular isolation levels [20], which can be complex and affect performance.

1.2 Skeena

We present Skeena, an efficient approach to consistent cross-engine transactions that solves the aforementioned problems in the context of multi-versioned, fast-slow systems: multi-versioning is widely used and as evidenced by real use cases, cross-engine transactions can be very useful in fast-slow systems.

We make two key observations to guide Skeena's design. First, as noted by prior work [8], inconsistent snapshots can be avoided by carefully selecting a snapshot in each engine. This requires efficient tracking of snapshots whose results can be safely used by later transactions. Second, since engines are developed and/or well understood by the same vendor, it is usually feasible to make non-intrusive changes to engines, allowing more optimizations.

Based on these observations, we design Skeena to consist of (1) a cross-engine snapshot registry (CSR) and (2) an extended pipelined commit protocol. The former ensures efficient and correct snapshot selection, and the latter ensures atomicity and durability without expensive 2PC. Both building blocks can be easily plugged into an existing system without much engine-level change (if any).

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 as the key to select a snapshot in E using which would avoid incorrect executions. Further, with CSR one only needs to set each engine to use a serializable protocol that exhibits commit ordering to guarantee serializability. Later, we discuss the detailed algorithms to realize this idea and techniques that make it lightweight and easy to maintain. In fast-slow systems, CSR incurs negligible overhead as the storage accesses in the traditional engine present a bigger bottleneck, and single-engine transactions do not need to access CSR at all.

Leveraging the fact that engines can communicate via fast shared memory (e.g., by sharing the same process address space), Skeena extends the widely-used group/pipelined commit protocols [32, 56, 60] 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 committer thread monitors the queue and durable log sequence numbers in both engines, to dequeue transactions whose sub-transactions' log records have been fully 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 default InnoDB engine and ERMIA [34], an open-source main-memory engine.[1] This required < 100 LoC changes (out of its over 200k LoC) in InnoDB and no change in ERMIA. Evaluation

[1]Downloaded from https://github.com/sfu-dis/ermia/releases/tag/alpha1.

using variants of YCSB [17] microbenchmarks and TPC-C [53] on a 40-core server shows that Skeena retains memory-optimized engine's high performance without impacting single-engine transactions, and incurs up to 5% overhead for cross-engine transactions.

Note that our goal is *not* to build faster database engines, nor to invent new concurrency control protocols for cross-engine transactions; both are well studied by prior work. Instead, our goal is to (1) enable cross-engine transactions without unnecessary overhead and (2) explore system designs that fit modern fast-slow systems.

1.3 Contributions and Thesis Organization

This thesis makes four contributions. ❶ We analyze the correctness requirements of cross-engine transactions under various isolation levels, ranging from read committed to snapshot isolation. ❷ We distill a set of desirable properties and design principles to be followed by dual-engine systems. ❸ We design Skeena to enable consistent cross-engine transactions by leveraging the properties of the fast-slow architecture. ❹ We show Skeena's feasibility and explore practical design issues by integrating an open-source memory-optimized engine (ERMIA [34]) into MySQL alongside its storage-centric engine (InnoDB), and conduct a comprehensive evaluation using microbenchmarks and TPC-C variants.

Next, we describe background in Chapter 2, design principles and details of Skeena in Chapter 3–4, and how Skeena works in real MySQL in Chapter 5, followed by evaluation results in Chapter 6. We survey related work in Chapter 7 and conclude in Chapter 8. t

Chapter 2

Background

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

2.1 Modern Fast-Slow Systems

We have described the basic ideas of multi-engine systems in Chapter 1. Several production systems already adopt the fast-slow architecture: SQL Server supports memory-optimized tables managed by its Hekaton main-memory engine [22, 41]; PostgreSQL supports additional engines through foreign data wrapper [46], which is used by Huawei GaussDB to integrate a main-memory engine [5].

Multi-engine systems bear similarities to distributed and federated database systems [51, 11, 47, 31, 38, 8, 25, 9, 18], but are unique in several ways. As Table 2.1 summarizes, a multi-engine system typically integrates engines developed and/or understood by the same vendor, instead of opaque systems developed by different vendors in federated systems. Distributed systems typically involve a set of nodes that run the same engine carefully designed to support distributed transactions, exhibiting low autonomy. Fast-slow systems integrate different engines that vary in performance, so an inefficient cross-engine solution would penalize single-engine transactions, defeating the purpose of using a memory-optimized engine; mitigating such overhead is the major goal of our work. Note that multi-engine systems often allow slightly trading autonomy for performance and compatibility, e.g., some systems manage schemas in all engines centrally. However, federated systems allow little room to do so as each system can be a proprietary package. Multi-engine systems can scale up and scale out, whereas the other two types of systems mainly scale out. We focus on fast-slow, dual-engine systems and leave systems with three or more engines to future work

2.2 Database Model and Assumptions

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

Table 2.1: 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 and/or out*	Out	Out

Multi-Versioning. Many fast-slow systems are multi-versioned, including all those mentioned previously. Following prior work [13, 8, 57, 2, 1], we model databases as collections of records, each of which is a totally-ordered sequence of *versions*. Updating a record appends a new version to the record's sequence. Inserts and deletes are special cases of updates that appends a valid and special "invalid" version, respectively. Obsolete versions (as a result of deletes) are physically removed only after no transaction will need them, using reference counting or epoch-based memory management [34, 10].

Reading a record requires locating a proper version; this is dictated by the concurrency control protocol. We base our discussion on a common design [22, 34, 37, 59] where the engine maintains a global, monotonically increasing counter that can be atomically read and incremented. Note that in a multi-engine system, engines maintain their own timestamp counters, i.e., each engine maintains its own internal "timeline" invisible to other engines; 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 (which determines its commit order relative to other transactions) 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 a particular point in (logical) time.

Isolation Levels. Various isolation levels can be implemented under this database model. Read committed translates into always reading the latest committed version. The transaction refreshes its read view by reading the counter upon each record access. Under snapshot isolation, the transaction uses its begin timestamp (obtained upon the first data access or transaction creation) as its read view and is allowed to access the latest versions created before its read view. A transaction can update a record if its read view is newer than (i.e., it can "see") the latest committed version of the record. Serializability can be achieved by various approaches, such as locking [15, 16, 23] and certifiers [13, 57, 36]. The goal of our work is to ensure these isolation levels are properly enforced in the presence of cross-engine transactions.

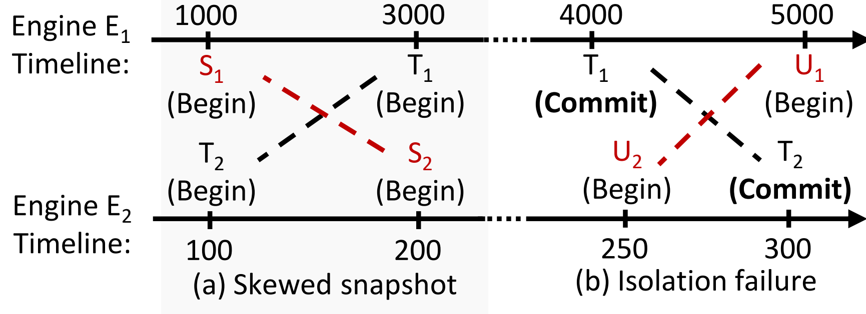

Figure 2.1: Inconsistent snapshots. (a) S uses an older (newer) snapshot in E_1 (E_2). (b) U sees partial results (U_1 sees T_1's results, U_2 does not see T_2's). Both also happen under distributed SI [8], but isolation failure affects all isolation levels.

Cross-Engine ACID Properties. Analogous to individual engines that run ACID sub-transactions, a cross-engine system must maintain ACID properties for 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 in their corresponding engines.
- 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 durably persisted while guaranteeing atomicity.

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

2.3 Cross-Engine Anomalies and Motivation

To access data records, cross-engine transactions rely on engine-level sub-transactions, which may begin and commit in any order depending on the workload. The relative ordering and timing of sub-transaction commit and begin events directly determine correctness, as certain ordering may lead to anomalies and violate ACID requirements. We summarize the issues below.

Issue 1: Inconsistent Snapshots. There are two cases where a transaction may be given an inconsistent view of data. As Figure 2.1(a) shows, S started in E_1 with a snapshot 1000, while 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

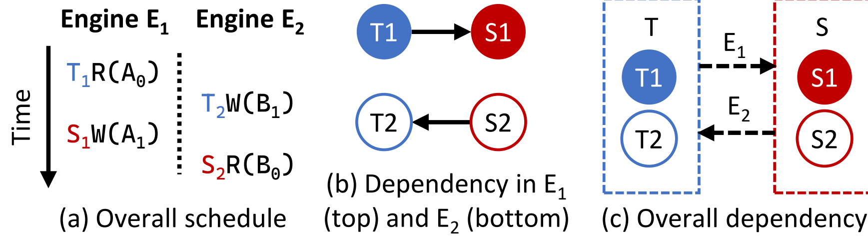

Figure 2.2: Non-serializable execution of cross-engine transactions when each engine guarantees serializability. (a) Each engine executes a serializable schedule (b) without cyclic dependencies. (c) Overall cyclic dependency between T and S.

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 to start before each other, which is impossible in a correct SI schedule [2]. This is the same as the "cross" phenomenon in distributed SI [8].

Another anomaly (isolation failure) may allow partial results to become visible earlier than they should be. In Figure 2.1(b), T first commits T_1 with timestamp 4000. Until T_2 is fully committed, T is still in-progress, so none of its changes should be visible to other transactions. Meanwhile, U starts in E_2 with timestamp 250, and continues to open U_1: since U_1 started after T_1 committed, by definition its read view should include $T1$'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 distributed SI [8]. Compared to 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 a different write-read dependency per engine.

Issue 2: Serializability. Snapshot isolation may lead to non-serializable results with the write-skew and read-only anomalies [27, 28]. Various approaches can forbid them [26, 36, 57, 13, 22], however, in a multi-engine system, even if both engines guarantee full serializability, the overall execution may not be serializable. As Figure 2.2(a) shows, 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 2.2(b).[1] However, as shown in Figure 2.2(c), the overall execution exhibit write skew with cyclic dependencies ($T \rightarrow S \rightarrow T$), indicating non-serializable execution.

Issue 3: Atomicity and Durability. Similar to single-engine transactions, a cross-engine transaction needs to be committed in its entirety, i.e., either all sub-transactions are successfully committed with their changes persisted, or none of them. This means the system needs to commit each sub-transaction in its corresponding engine. Should any sub-transaction

[1] Unlike 2PL and OCC, some serializable SI schemes allow safe anti-dependencies to improve concurrency and lower abort rate [13, 57, 14].

fail to commit (e.g., due to serializability violations), other sub-transactions must also abort. The dominant solution in distributed systems has been 2PC, but it may not be the best choice for single-node multi-engine systems. Unlike distributed systems where each node often has baked-in 2PC support, engines in a multi-engine system may not directly support 2PC: many newer main-memory systems explicitly avoided 2PC by design [7, 62]. Moreover, 2PC can be unnecessarily complex to implement and heavyweight for a shared memory system.

Summary and Motivation. As noted earlier, the issues identified here also arise in distributed SI and federated systems, and prior work has laid the foundation in concurrency control theory for solving these problems [48, 8] (described later). In practice, however, existing approaches targeted distributed environments without considering the characteristics of single-node, fast-slow systems. For example, incremental distributed SI [8] allows transactions to acquire consistent snapshots as needed following a set of rules, but uses global IDs across all participating nodes that require engine-level changes, as well as a central coordinator node. Prior solutions relied on 2PC for atomicity, which as we mentioned does not suit single-node systems. Finally, although serializability and concurrency control has been discussed extensively [12, 48, 31, 47], there has been little focus on consolidating these results to devise an efficient solution for various isolation levels in fast-slow systems.

Chapter 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 low overhead. Especially, it should not penalize single-engine transactions, especially those running in the faster engine; only cross-engine transactions should pay the extra cost (if any).
- **Engine Autonomy.** Engines should be kept as-is, or be modified in non-intrusive ways to cope with the cross-engine mechanism and/or optimize for performance, leveraging the fact that engines are typically developed by the same vendor.
- **Full Functionality.** The mechanism should support various isolation levels for both single- and cross-engine transactions, unless it is limited by individual engine capabilities.
- **Transparent Adoption.** The application should not be required to make logic changes. Rather, it should only need to declare the "home" engine of each table in database schema.

Chapter 4

Skeena Design

We start with an overview and transaction workflow, and then describe its design in detail. For clarity, we base our discussion on two engines that use snapshot isolation, before expanding to other isolation levels and supporting more than two engines.

4.1 Overview

Skeena's key functionality is to ensure correct snapshot selection and atomic commit with proper ACID guarantees. It can be thought as a global "snapshot provider" and commit protocol that engines use to handle cross-engine transactions. Figure 4.1 gives an overview of Skeena with its key building blocks: (1) the cross-engine snapshot registry (CSR) and (2) a pipelined commit protocol that is aware of cross-engine transactions. The former tracks recent valid cross-engine snapshots as a result of starting and committing cross-engine transactions. The latter is an adaptation of the well-known pipelined group commit protocol [32] to support atomically committing cross-engine transactions that consist of multiple sub-transactions. As Chapter 5 shows, adopting neither in a real system requires intrusive changes (if any) to engines. We briefly describe how transactions are handled under Skeena.

Initialization. Transactions (single- or cross-engine) can keep using the APIs (e.g., SQL) provided by the database system. Skeena does not require additional hints from the application (e.g., whether a transaction will be cross-engine). Figure 4.1 shows an example of a SQL program, which is written in the same way without Skeena.

Skeena does not restrict transactions to run under specific isolation levels. However, the system may provide additional commands to set the desired isolation level (e.g., `SET TRANSACTION ISOLATION LEVEL` in MySQL [43]). Each individual engine then would be set to use the specified isolation level. As part of the integration effort, Skeena can detect such settings and enforce the corresponding isolation levels across all engines (Chapter 4.2).

Data Accesses. Upon receiving a data access request, the system routes it to the proper engine which uses a sub-transaction access data. We expect the routing mechanism is given as multi-engine systems already implement them; Skeena requires no change to it. Upon

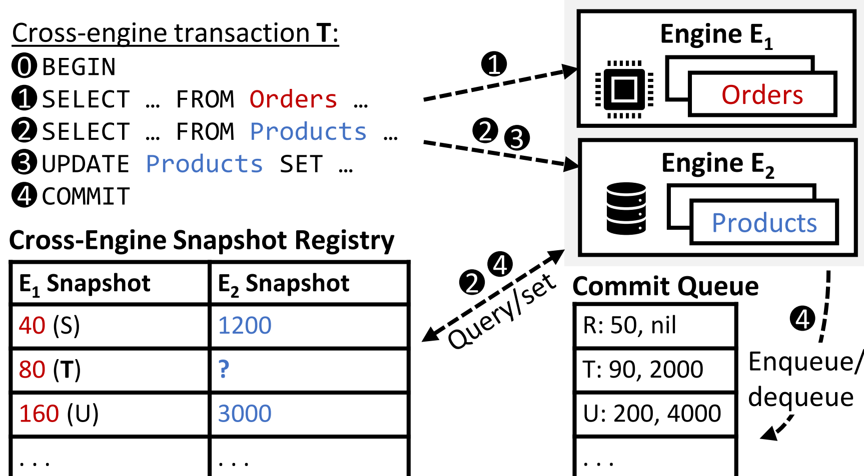

Figure 4.1: Skeena overview. ⓿–❶ Transactions access data without explicitly declaring whether they are cross-engine. ❷ Upon accessing an additional engine, the transaction ❸ consults CSR to obtain a proper snapshot. ❹ Cross-engine transactions use CSR for commit check and if passed, goes through the pipelined commit protocol to conclude.

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 obtain the latest snapshot in the underlying engine, or consult CSR to obtain a proper snapshot that would not cause anomalies (steps ❷–❹ `Query/set`). As we will see later, such a snapshot may not always exist due to necessary trade-offs for performance and easier implementation. In this case, the transaction will be aborted; we quantify the impact on realistic workload in Chapter 6.

Finalization. Upon commit, the cross-engine transaction consults CSR to verify that committing all sub-transactions would not cause future transactions to obtain inconsistent snapshots; if such commit check fails, the transaction is aborted.[1] Single-engine transactions commit directly without going through Skeena mechanisms. Transactions that passed CSR verification are marked as pre-committed and placed on the commit queue. Once their log records are persisted, we dequeue it and notify the application of a successful commit (step ❹ `Enqueue/Dequeue`). If the verification failed, we abort the transaction by rolling back all sub-transactions.

In the rest of this chapter, we describe how Skeena facilitates snapshot selection, atomicity and recovery, beginning with the high-level functionality and algorithms that CSR should support.

[1] An alternative is to adjust the commit timestamps of cross-engine transactions until they can pass the commit check. However, this may require intrusive changes to other subsystems, such as logging and recovery, violating our design principle to maintain engine autonomy.

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 e1_snap
9     e2_snap = max(candidates)
    CSR.map(e1_snap, e2_snap)
11  return e2_snap
```

4.2 Basic Cross-Engine Snapshot Registry

The key to avoiding inconsistent snapshots is ensuring the sub-transactions of different cross-engine transactions follow the same start order in each engine [8]. 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 S_2 in the other engine, E_2. For example, in Figure 4.1, T first started as a single-engine transaction accessing `Orders` in E_1, using snapshot 80. When T starts to access `Products` in E_2, T needs to find a snapshot (s) in E_2 in CSR such that s falls between the snapshots being used by its "neighbor" transactions in E_1, i.e., S_1 and U_1. Therefore, 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 snapshot selection process, CSR tracks valid snapshots (i.e., commit timestamps of recent cross-engine transactions) that can be safely used by cross-engine transactions. Conceptually, it is a table that supports point and range queries, where each "row" (CSR entry) is a pair of snapshots (i.e., commit timestamps), one from each engine as depicted by Figure 4.1. When a transaction crosses to access an additional engine, it uses the current engine's snapshot as the key to query CSR for a snapshot in the target engine. As shown in Algorithm 1, to access a new engine `e2`, the worker thread issues a non-inclusive forward scan over CSR using the snapshot in the current engine `e1` as the key (`e1_snap`) to obtain a set of candidate snapshots. The scan returns once a first key greater than `e1_snap` is met or no such key is found. If the scan returned an empty set, then no past transaction has set up any mapping or the current transaction is using the latest `e1` snapshot. Then we proceed to use the latest `e2` snapshot (lines 4–6). However, if any candidate is found, we must take an `e2` snapshot that is already mapped to `e1_snap` to avoid anomalies (lines 7–9). Finally, we setup the new mapping for future transactions to avoid anomalies. Under snapshot isolation, the algorithm is executed only once per transaction when it becomes cross-engine. Subsequent accesses continue to use the previously acquired

Algorithm 2 Commit check for cross-engine transactions.

```
1 def cross_engine_commit_check(sub_t1&, sub_t2&):
    # Obtain lower and upper bounds for sub-transaction t2
3   low = -inf
    candidates[] = CSR.reverse_scan_1st(sub_t1.commit_ts)
5   if candidates[] is not empty:
      low = max(candidates)
7
    high = +inf
9   candidates[] = CSR.forward_scan_1st(sub_t1.commit_ts)
    if candidates[] is not empty:
11    high = min(candidates)

13  # Check if committing t2 would cause future anomalies
    if low > sub_t2.commit_ts or high < sub_t2.commit_ts:
15    return false
    else:
17    # Check passed, setup mapping and return
      CSR.map(sub_t1.commit_ts, sub_t2.commit_ts)
19    return true
```

snapshots. Following the above process, a cross-engine transaction should always be able to obtain a valid timestamp using CSR.

In addition to acquiring snapshots, committing a cross-engine transaction implicitly limits the ranges of snapshots a (future) cross-engine transaction may use: recall that the commit timestamp of a previous transaction T in fact is the begin timestamp (i.e., snapshot) of a future transaction that reads the results generated by T. We therefore also track commit timestamps of cross-engine transactions in CSR. Algorithm 2 describes the process at a high level. Here, we assume the sub-transaction commit timestamps are already available (as the `commit_ts` member in each sub-transaction); we revisit this assumption later in more detail. The idea is to ensure that committing a cross-engine transaction—i.e., adding a new mapping entry to CSR—would not cause the new table to exhibit skewed snapshots. To achieve this, upon commit, we issue a reverse scan and a forward scan over CSR using a sub-transaction's (`sub_t1`) commit timestamp to obtain the lower and higher bound for the other commit timestamp (lines 4-11 in Algorithm 2). Then, if `sub_t2`'s commit timestamp falls between the higher and lower bounds, we can safely commit this cross-engine transaction and setup a new mapping in CSR (line 18). Otherwise, the transaction must abort. Note that the mapping process in Algorithm 1 is still necessary because (1) single-engine commits are not covered by CSR to avoid unnecessary overheads, and (2) a cross-engine transaction may access data generated by single-engine transactions and form new cross-engine snapshots.

Since a transaction may access engines in any order (e.g., 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 easily implemented using a ordinary 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 to allow queries originated from both sides. However, this can create dependency on a particular engine and incur much overhead of keeping two index structures consistent, as well as possible overhead of maintaining extra (meta)data of a full-fledged relational table. The snapshot selection algorithms and CSR structure discussed so far assumed single-threaded execution; a practical design must also support concurrent accesses for reasonable performance. We address these issues next.

4.3 Lightweight Multi-Index CSR

We take advantage of the unique properties of fast-slow systems to devise a lightweight CSR that mitigate the above issues. In a fast-slow system, compared to the storage-centric engine, it is typically very cheap to obtain a snapshot in the memory-optimized engine. For example, in most engines this is as simple as reading the timestamp counter which is an 8-byte integer, without even having to enter a critical section. However, obtaining a snapshot in a storage-centric system can be much more complex, e.g., the process in MySQL InnoDB involves taking multiple mutexes and computing multiple watermark values. Therefore, we simplify our cross-engine mechanism to always follow the snapshot order by one of the engines, removing the need and possibility of querying the CSR using a snapshot from the storage-centric engine. We call this engine the *anchor* engine which typically is the engine where it is cheaper to acquire a snapshot; it can be designated at system initialization time. This way, transactions always start by acquiring the latest snapshot from the anchor engine, and use it as the key to query the CSR when it extends to the other engine. At a high level, this allows us to simplify the design of CSR to be a single range index structure. Our current implementation uses Masstree [40], a high-performance in-memory index, but any concurrent data structure that supports range queries would suffice.

A side effect is that any transaction that accesses the "slower" engine now becomes cross-engine and is subject to go through Algorithms 1–2 even if they do not access any record in the anchor engine. As we show later in Chapter 6, the overhead is minuscule: compared to actual data accesses (which may go through the storage stack), the extra steps required are very lightweight and even negligible because the entire CSR is an in-memory structure. For clarity, in the rest of this thesis, we define transactions to be single- or cross-engine from the user's perspective: if the user application only accesses the non-anchor engine—although internally it accesses both engines—we refer to it as "single-engine."

Because CSR tracks past cross-engine snapshot and commit histories, its size can grow quickly over time and entries that are no longer useful should be properly cleaned up. As the CSR index structure grows, querying it may become slower, limiting system performance. We solve these problems by partitioning the CSR by snapshot ranges, reminiscent of multi-rooted B-trees [45]. The result is a "multi-index" design shown in Figure 4.2 where each partition is

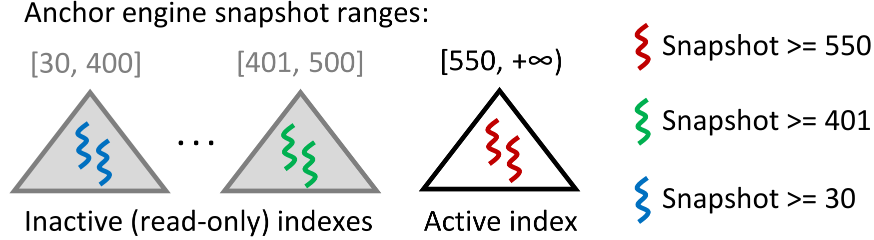

Figure 4.2: Multi-index CSR.

a standalone index (Masstree in our case) and covers a range of snapshots. Depending on the requesting transaction's snapshot from the anchor engine, we use the corresponding index partition for snapshot selection and commit check. As shown by Figure 4.2, the first two indexes cover snapshot mappings whose snapshots from the anchor engine are in the ranges of [30, 400] and [401, 500], respectively. Note that each index here covers a unique range, such that a transaction only needs to work on a single index. Each partition has a fixed capacity and new indexes (partitions) are created when the current open index is full. A full partition becomes read-only once a newer index is created. As a result, we maintain one and only one open index that accepts new mappings; the other indexes become read-only and can continue to serve existing transactions for snapshot selection. The rationale behind maintaining multiple read-only indexes and allowing only one read-write index is to avoid skewed snapshots more easily: if new mappings were allowed in an inactive index I, then a transaction that is using a very old anchor snapshot may setup a new mapping that uses a very recent snapshot from the other engine that is newer than the most recent snapshot in I. The mapping may potentially conflict with mappings in other indexes. Avoiding this problem in turn would require transactions to check more than one index which increases the overall overhead. We therefore keep the inactive indexes read-only for simplicity and efficiency. The drawback is that abort rate may increase, but we find it rare in practice; we show details in Chapter 6.

The multi-index design allows us to easily recycle stale snapshot mappings: an inactive index is deleted as a whole once its snapshot range is no longer needed even by the oldest transaction running in the system. The recycling procedure first iterates over all the running transactions to obtain the oldest anchor-engine snapshot being used (`min_snap`), and then goes through the list of indexes to bulk delete any index whose snapshot range is below `min_snap`. Our current implementation triggers recycling work in between CSR accesses based on a user-defined threshold (e.g., once every 10000 accesses); this could also be delegated to a background thread.

We maintain a list to track all indexes. Upon snapshot selection or commit check, the accessing thread traverses the list to find the appropriate index to work on. We protect the list using a reader-writer lock that provide mutual exclusion between threads that only

query an index (without modifying the index list) and those that may add (e.g., growing the CSR by creating a new index) or remove (garbage collection) an index.

An important trade-off brought by the multi-index design is that unlike the basic mechanism described in the previous section, a cross-engine transaction may have to be aborted if it fails to find a proper snapshot (e.g., because its partition index is already read-only). In practice, as we show in Chapter 6, such aborts are rare.

4.4 Commit Protocol

Once all accesses are done, we need to ensure both sub-transactions are committed or none of them. In order to conduct the commit check mentioned earlier, for each sub-transaction Skeena needs to obtain its commit timestamp. However, we must not commit the sub-transaction as a result of obtaining a commit timestamp. This is usually straightforward for memory-optimized engines as they often use optimistic flavored concurrency control methods. In these engines, the commit process involves a "pre-commit" phase first that obtains a tentative commit timestamp. Then it uses the obtained timestamp to verify the transaction can commit safely without violating serializability or other correctness criteria [22, 34]). For the storage-centric engine, Skeena requires a similar pre-commit phase which may not be already available. However, note that the different engines in a multi-engine systems are typically developed and well understood by the same vendor. This justifies simple surgical changes to expose such a pre-commit interface that only obtains commit ordering without actually committing the sub-transaction. Chapter 5 describes our experience of exporting this interface for InnoDB, where we only needed to change fewer than ten lines of code. As a result, we break the monolithic commit process into two phases, including pre-commit and post-commit which finishes up committing a sub-transaction after pre-commit. The post-commit process is expected to succeed except for disastrous events (e.g., disk full, crash or power failure), i.e., after pre-commit and CSR commit check, the cross-engine transaction can safely commit. Single-engine transactions directly execute the two phases without intermediate steps and bypass the cross-engine commit protocol below.

With the pre-commit interfaces, the first step to commit a cross-engine transaction is to pre-commit both sub-transactions. Then, using the commit timestamp obtained from the anchor engine, Skeena queries the CSR to finish the commit check following Algorithm 2. If the commit check passes, we need to finish the post-commit phase for both engines, before which the new results by the cross-engine transaction (i.e., from any sub-transaction) should not be made visible. However, from the perspective of an individual engine, a post-committed (sub-)transaction is fully "committed" with its results visible to future transactions. Therefore, Skeena must ensure the partial results are not visible until all sub-transactions are post-committed.

Skeena solves this problem by adapting pipelined/group commit [32, 56] which is already in use in many real systems [55, 21].[2] In a single-node, single-engine setting, transactions may be detached from threads so that log flushing does not block the underlying thread, which can continue to process other transactions while log I/O is in-progress. Upon commit, instead of directly issuing a log flush, the thread detaches the transaction and appends it to a global commit queue (or a partitioned, distributed queue to avoid introducing a central bottleneck). A dedicated committer thread (which sometimes can be the log flusher thread) then tracks transactions awaiting log durability on the commit queue, and dequeues transactions whose log records have been persisted. Results by these transactions are immediately visible internally to other transactions, however, the results are not returned to the client application until the transaction's log records have been persisted. This approach has been used by several systems to improve throughput without sacrificing correctness [32, 58, 56].

We adapt this approach to solve the partial result issue: sub-transactions are pushed on the commit queue as shown by Figure 4.1 during post-commit. The underlying thread then continues to execute other transactions, while log flushes are happening in the background. The commit daemon thread then only dequeues transactions once their log records are persisted. While sub-transactions are on the commit queue, their results are already visible to other transactions. However, same as the single-node case, any results depending on these in-progress results will not be returned to the client application until the log records are persisted, maintaining correctness for client applications. Moreover, if one or more engines already implements commit pipelining, Skeena can directly piggyback on it to realize the aforementioned functionality. Note that single-engine and read-only transactions must also go through the above protocol to ensure correctness, because they might read changes that are generated by cross-engine transactions; we quantify its effect in Chapter 6.

4.5 Durability and Recovery

In multi-engine systems, each engine implements its own approach to durability and crash recovery. Therefore, sub-transactions still follow their corresponding engines' approach to persist data and generate and persist log records. Checkpoints can be taken as usual independently by each engine for its own data. To ensure atomicity of cross-engine transactions, Skeena maintains a lightweight log to denote the pre-commit and post-commit of cross-engine transactions. This can be done by maintaining a standalone log or piggybacking on individual engines. The latter can be easier to implement: upon starting pre-commit a sub-transaction the engine appends a `commit-start` record, and after post-commit finishes, the engine appends a `commit-end` record. This way, upon recovery, each engine executes

[2] Aurora and Taurus DB also refer to it as "asynchronous commit" which is different compared to academic definition of "asynchronous commit" that allows a transaction to commit (with results returned to clients) without persisting the log.

its recovery mechanism to redo/undo log records, and then rolls back any changes done by cross-engine transactions whose sub-transactions are not all fully committed, using the additional log information. Alternatively, the recovery procedure may first inspect the logs of each engine, and simply truncate it at the first "hole" where only one sub-transaction of a cross-engine is recorded as committed. This is safe because the commit pipelining in Skeena ensure that the results of a transaction are only released when all of its depending log records are persisted [32]. Later transactions that depend on partially committed cross-engine transactions will wait on the commit queue and therefore can be safely discarded in case of a crash as they were never made visible to the client application.

Chapter 5

Skeena in MySQL

With the design of Skeena laid out, now we describe our experience of using it to realize cross-engine transactions in MySQL between its default InnoDB storage-centric engine and ERMIA [34], an open-source main-memory engine. To start with a fair, up-to-date baseline, we modify the community version of MySQL by adding commit pipelining and thread pool support, as they are used by industry-strength MySQL variants by major vendors (such as Amazon Aurora [55] and Huawei Taurus [21]). As a result, each client connection is served by a thread drawn from the thread pool. The thread then uses a routing mechanism (which is already part of original MySQL) to access the corresponding engine based on the data request. Upon commit, the thread detaches the transaction to place it on the commit queue maintained by the pipelined commit protocol. The thread is then returned to the thread pool to continue to handle the next request.

Skeena selects ERMIA as the anchor engine. The snapshot mechanism is implemented in a more lightweight fashion in ERMIA than in InnoDB. For ERMIA, acquiring a snapshot does not need any latches. For InnoDB, acquiring a snapshot may require taking a system-wise mutex if the transaction is not read-only. We identified three challenges.

The first challenge is the conversion from conceptual CSR snapshot to the InnoDB read view. To approach this, the CSR stores the high watermark of a read view. An InnoDB read view consists of a low watermark, a high watermark, and an active transactions list. The current transaction should not see any transaction with transaction ID (`trx_id`) greater than or equal to the high watermark, and can see those with `trx_id` strictly smaller than the low watermark. A transaction with `trx_id` in between the two watermarks is not visible if it is also in the active transactions list. This list contains currently active read-write transactions. Therefore, to make the read view compatible for cross-engine transactions, we only need to further adjust the low and high watermarks based on the CSR snapshot after they are first prepared by the original InnoDB MVCC logic. The high watermark stored in the CSR is from a history read view such that it is no larger than the one before adjustment. Because the active transaction list remains unchanged after adjustment, the invisibility of the transactions in the list is still guaranteed.

The second challenge is the acquisition of InnoDB LSN in pre-commit. Skeena commit protocol requires both engines to obtain the pre-commit LSN for commit check before the actual commit. For ERMIA, it already has a pre-commit interface. For InnoDB, the commit LSN is generated when a log is written to disk, making it impossible to get the commit LSN before commit. The commit LSN indicates the order of commits, so it is also feasible to use an equivalent counter that indicates the same order in InnoDB. We discover that during the commit, InnoDB acquires a serialization number (`serialisation_no`) before writing the log. The serialization number is acquired from the same counter that generates `trx_id`, and follows the same order of the commit LSN. So instead of acquiring the commit LSN, the pre-commit interface simply takes the serialization number from this counter. After commit check, the commit entry with commit LSNs from both engines will be inserted to CSR. For ERMIA, it is safe to do so since the ERMIA commit LSN is the begin LSN of the next transaction. For InnoDB, the `serialisation_no` comes from the same monotonically increasing counter that generates the read view watermarks, so the `serialisation_no` is the high watermark of the next transaction at the same time.

The last challenge is the implementation of the commit pipelining. Original MySQL worker threads wait for the transaction to become durable and then return the results in a synchronous fashion. Now the pipeline breaks the MySQL commit logic into two stages. The first stage ensures the transaction is committed. Once a transaction is committed, the worker thread enqueues the commit entry into the commit queue in ERMIA and then can be detached to process another new transaction. The commit entry consists of ERMIA commit LSN, InnoDB commit LSN, and a callback function. The dedicated dequeuer thread will not invoke the callback function and dequeue the entry from the commit queue until both commit LSNs are smaller than the durable LSNs in their respective engines. This callback function can be treated as the second stage where MySQL returns the results to the user, and performs clean-ups. The single-engine transactions also go through this commit pipeline mechanism to prevent users from reading partial results.

Chapter 6

Evaluation

In this chapter, we empirically evaluate Skeena in the MySQL environment described in Chapter 5 to understand its performance impact and explore the potential of cross-engine transactions under realistic workloads. Through experiments, we show the following:

- Skeena retains the performance benefits of memory-optimized engines by incurring no overhead over single-engine memory-optimized transactions;
- Only cross-engine transactions and single-engine transactions that access the "slow" engine need to pay additional costs, which however is also very small;
- By judiciously placing tables in different engines, Skeena can effectively improve transaction throughput;
- Skeena realizes cross-engine transactions without mandating fundamental changes to the application, which only needs to declare the table's home storage engine.

6.1 Experimental Setup

We run experiments on a dual-socket server equipped with two 20-core Intel Xeon Gold 6242R processors (80 hyperthreads in total), 384GB of main memory, and a 400GB Micron SSD with peak bandwidth of 2GB/s. Each CPU has 35.75MB of cache and is clocked at 3.1GHz.

We conduct experiments in the aforementioned MySQL environment that integrates ERMIA and its default InnoDB engine. The MySQL version used is 8.0.[1] We use the standard SysBench [44] tool to issue database benchmarks (described later). Since MySQL employs a client-server architecture, to reduce the impact of network latency, we run the database server on one CPU socket, and the client on another socket; because our server and client are on the same machine, we directly use a Unix Domain Socket (instead of TCP/IP) for faster communication between the server and client (SysBench) [61]. We use snapshot isolation

[1] Downloaded from https://github.com/mysql/mysql-server.

(repeatable read in InnoDB) to run all experiments and report the throughput and latency, each run lasts for one minute. All queries are interpreted without compilation, however, one could still employ it to achieve even better performance for the memory-optimized engine; we leave it as future work.

To stress test Skeena and both engines, we store persistent data (such as data files and logs) in `tmpfs` so that all storage accesses in fact go through main memory. ERMIA is memory-optimized so all records are in heap memory. For InnoDB we test three different storage settings:

- **memory-resident:** The database fits in the buffer pool to avoid accessing storage and file system stack.
- **storage-resident(tmpfs):** It uses a small buffer pool that would mandate accessing the storage stack, the storage back-end is tmpfs and all data is stored in DRAM.
- **storage-resident(ext4):** It uses a small buffer pool that would mandate accessing the hard disk below ext4 file system. The data is stored in the storage device mentioned above.

In all cases, we reinitialize the database for each run which then starts with a warm buffer pool.

To further investigate the overhead introduced by CSR multi-index design, we conduct a set of experiments varying different CSR index capacity and recycle threshold parameters. We describe the detailed experimental setup later.

6.2 Benchmarks

We first run YCSB-like [17] microbenchmarks to stress test Skeena, and then explore the usefulness and behavior of cross-engine transactions in realistic scenarios with TPC-C [53].

Microbenchmarks. We devise three microbenchmarks based on access patterns: read-only, read-write and write-only. Each transaction accesses ten records uniform randomly chosen from a set of tables. Out of the ten accesses, for read-write transactions, eight are point reads and two are updates. For 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` and one `VARCHAR` fields. For memory-resident experiments, each table contains 25000 records which brings the total data size of 250 tables to ∼1.35GB; the buffer pool size in InnoDB is set to 32GB. For storage-resident (both ext4 and tmpfs) 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. Under both settings, ERMIA is populated with the same amount of data, which is all stored in memory. For each transaction, we further vary the percentage of InnoDB and

ERMIA accesses (0%, 30%, 50%, 80%, and 100% InnoDB accesses out of the ten accesses). For example, under the 30% InnoDB setup, 30% of (i.e., three out of ten) record accesses per transaction will happen in InnoDB, the remaining seven accesses will happen in ERMIA.

TPC-C.[2] We use TPC-C for the dual-purpose of (1) understanding Skeena's performance under non-trivial transactions, and (2) exploring the potential benefits and trade-offs of cross-engine transactions in realistic scenarios. Similarly, we run the experiments under three different storage settings: memory-resident, storage-resident (tmpfs) and storage-resident (ext4). The memory-resident one uses a scale factor which is the same as the number of concurrent connections and the storage-resident (both tmpfs and ext4) experiments use 200 warehouses. For memory-resident experiments, each connection works on a different home warehouse but the 1% of New-Order and 15% of Payment transactions may respectively access a remote warehouse; we set InnoDB buffer pool to be 32GB which is large enough to hold all the data (∼14GB). With 200 warehouses, the total data size is ∼55GB, for which we set InnoDB to use a buffer pool of 5GB and set each thread (connection) to always pick a random warehouse as its home warehouse to ensure the footprint covers the entire database. Again, any table stored in ERMIA is full in-memory and accessing them does not involve the storage stack. Finally, we adapt selected TPC-C transactions to become cross-engine by changing table placement. This allows us to further explore the effectiveness of cross-engine transactions in realistic workloads, and distill useful suggestions. We discuss the detailed setup later.

6.3 Single-Engine Performance

Microbenchmark. We evaluate the single-engine performance in microbenchmarks and TPC-C. As shown in Figure 6.1, InnoDB is the more heavyweight one out of the two engines. Besides, the performance of single-engine transactions with CSR turned on, i.e., "0% InnoDB" and "100% InnoDB", is comparable to the performance of those with CSR turned off, which means that CSR incurs minuscule overhead to the single-engine transactions. The throughputs of pure InnoDB experiments on tmpfs and ext4 are very different, but show the same trend. This is because logging is a bottleneck for the storage-resident (ext4) case, and the write-intensive microbenchmarks such as read-write and write-only transactions require more I/O than read-only transactions by persisting log records. Therefore, the system is bottlenecked by storage I/O. But even under extreme cases that storage becomes the bottleneck, our experiments still show that the CSR does not incur high overhead to single-engine transactions.

TPC-C. As shown in Figure 6.3 and Figure 6.2, pure-innodb is the "lower bound" across all the types of transactions, while pure-ermia is the "uppper bound". The performance of

[2]Implementation adapted from https://github.com/Percona-Lab/sysbench-tpcc.

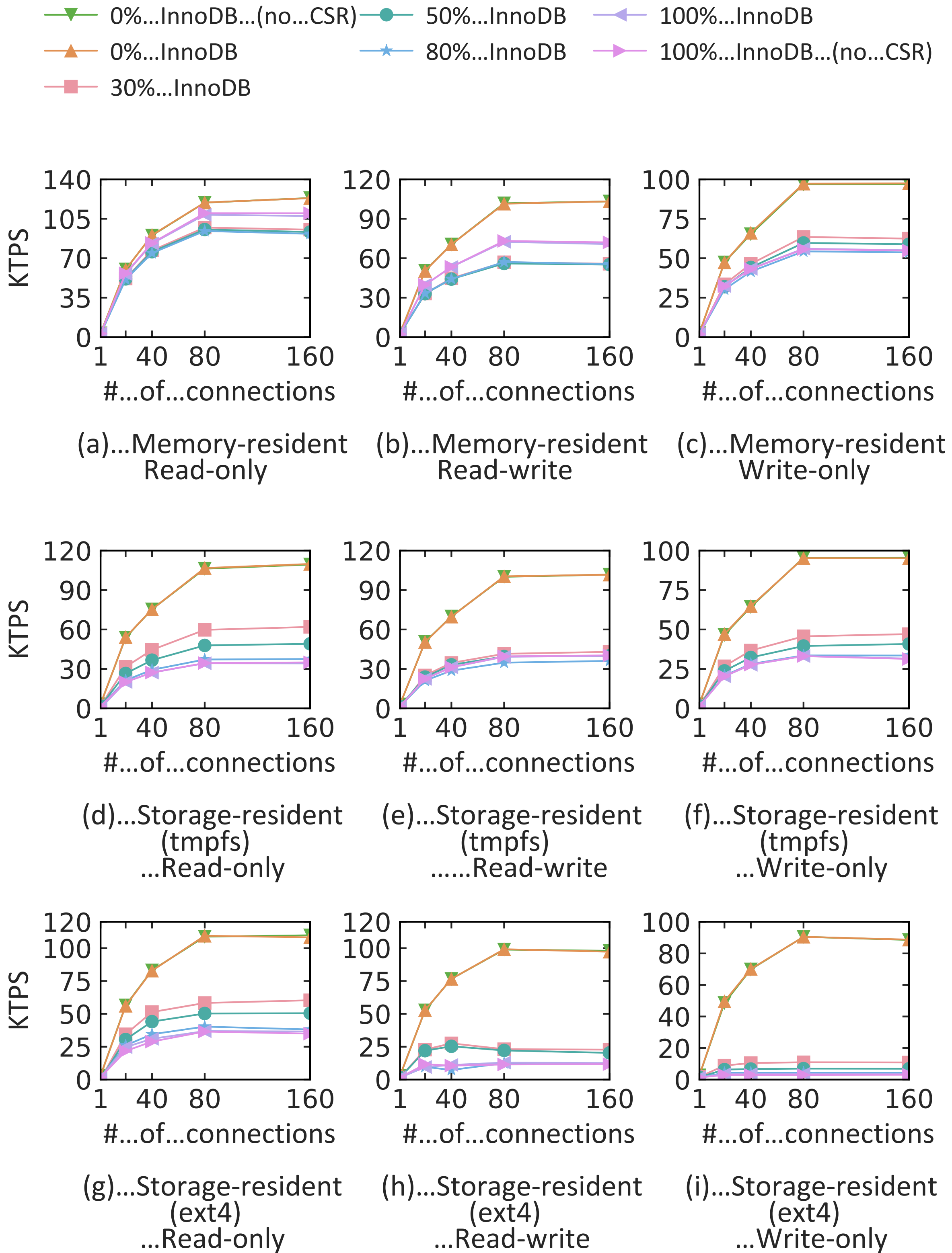

Figure 6.1: Microbenchmark.

most types of transactions become flat since the system is fully saturated on 40 threads. For the memory resident Delivery transaction, the latency increases as the worker threads grows, leading to a performance decrease because of the thread context switching and high latency.

6.4 Cross-Engine Performance

Microbenchmark. Since InnoDB is more "heavyweight", with the increasing percentage of ERMIA accesses in a transaction, the performance is expected to grow, e.g., 30% InnoDB should yield higher TPS than 100% InnoDB, yet the 100% InnoDB outperforms 30 - 80% InnoDB in Figure 6.1a and Figure 6.1c. The reason for this counter-intuitive finding is two-fold. First, for the read-only sub-transactions, ERMIA still generates a fixed-size commit record. In other words, with more ERMIA accesses, the CSR tree structure becomes bigger, requiring more operations on the tree to be done for each access, which is non-negligible for the read-involved workloads in the main-memory environment. Second, and more importantly, with 100% InnoDB, no ERMIA accesses are ever done, so the CSR is very small, with simply one key (ERMIA LSN) inserted, making it very cheap to query, and no tree recycling is ever needed. However, with other percentages, such as 30 - 80% InnoDB, more ERMIA accesses results in more LSN-TID mappings being stored in the CSR, as LSNs (keys) are inserted to the tree when ERMIA sub-transactions commit, thus it is more expensive to query the CSR, and garbage collection becomes necessary in this case. The memory-resident write-only workload exhibits more of expected trends in Figure 6.1e. The throughput is negatively correlated with the InnoDB percentage, since write-only transactions only execute updates, which translates to InnoDB holding exclusive locks under the hood, thus making the performance worse with higher InnoDB percentage despite all the data being memory-resident.

The memory-resident InnoDB is an extreme configuration, due to the fact that, in real life, a storage-based DBMS may store up to terabytes of data, thus the performance gains brought by lightweight ERMIA accesses cannot be easily canceled out as they are in the aforementioned cases. As the storage subsystem accounts for the biggest proportion of the overhead in the storage-resident version, the overhead incurred by CSR becomes negligible, with the expected behavior across all the cases: the higher ERMIA percentage, the higher TPS is. Figure 6.1b and 6.1f respectively show that "30% InnoDB" is up to 75% faster than "100% InnoDB" for read-only, and up to 40% faster for write-only. For read-write, cross-engine transactions have a 4% abort rate in average that comes from commit checks compared to 0% for single-engine transactions, and InnoDB rollback is non-trivial effort, making the cross-engine throughput less distinguishable from that of pure InnoDB. More details of abort rates are discussed next against TPC-C results.

TPC-C. We run the following TPC-C experiments:

- **Pure-ermia**: Single-engine setting with all the TPC-C tables in ERMIA.

- **Pure-innodb**: Single-engine setting with all the TPC-C tables in InnoDB.
- **Optimize-payment**: Cross-engine setting aiming at optimizing the performance of "Payment" transaction, which involves intensive access for the Customer table. With this setting, the Customer table is stored in ERMIA, other tables are stored in InnoDB.
- **Optimize-new-order**: Cross-engine setting aiming at optimizing the performance of "New Order" transaction. With this setting, the Customer and Item tables are stored in ERMIA, other tables are stored in InnoDB.
- **Archive-only**: Cross-engine setting with a more aggressive use of main memory engine. With this setting only the history data in History table is stored in InnoDB, all the other tables are stored in ERMIA for faster transaction processing.

The six workload types, combined with two InnoDB buffer pool settings and the five different table distribution scenarios make a total of 60 benchmark schemes.

Figure 6.2 and Figure 6.3 show that as the TPC-C tables are moved to ERMIA gradually, the performance of all types of TPC-C transactions increases and gets closer to `pure-ermia` performance. The observation matches the expectation that `pure-ermia` TPC-C has the best performance and act as the "upper bound" among all different engine schemes. However, the trend is different from the microbenchmark result. The reason is that the microbenchmark workload is simple, while the TPC-C workloads are more complex which includes SQL operations such as range scan and aggregation. The CSR overhead therefore becomes relatively low compared to the actual TPC-C workload. Another observation is that in both memory-resident and storage-resident TPC-C experiments, `archive-only` achieves almost the same performance regardless of InnoDB is storage-resident or memory-resident. Since in `archive-only` scheme, only the History table is stored in InnoDB. For the 200-warehouse setting of the storage-resident experiments, the total size of the history table is less than 600 MB, which means the whole table will be cached in buffer pool, so the performance is expected to be similar. The storage-resident (tmpfs) and storage-resident (ext4) experiments show similar throughput for most TPC-C transactions because logging is usually not a bottleneck, especially for those that are read-intensive. For write-intensive transactions, New-Order writes more log records than Payment does and logging becomes a bottleneck for New-Order. Therefore, performance of storage-resident (ext4) New-Order transactions is lower than the that of storage-resident (tmpfs). Nevertheless, we note that in either case they exhibit the same trend.

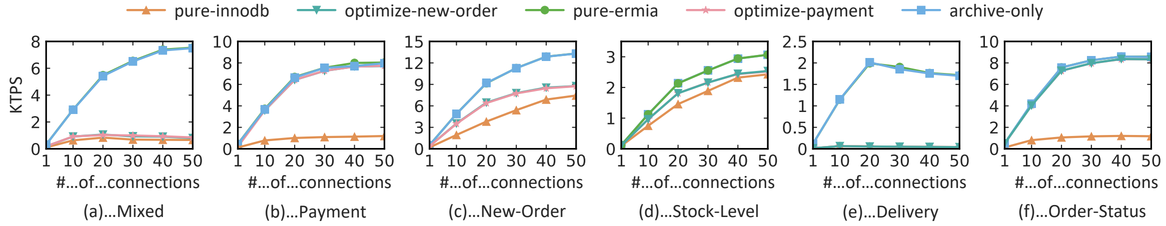

Figure 6.2: Memory-resident TPC-C.

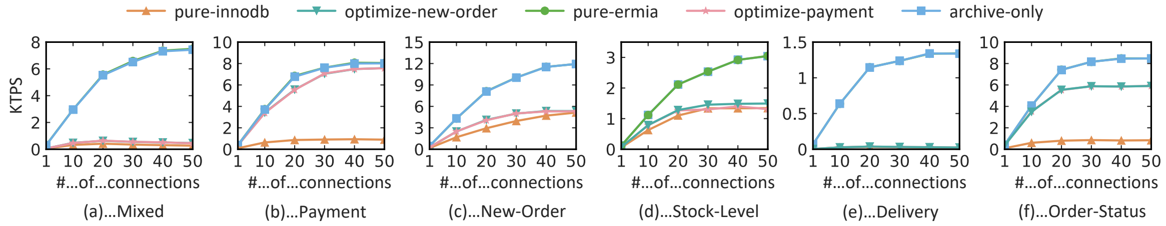

Figure 6.3: Storage-resident(tmpfs) TPC-C.

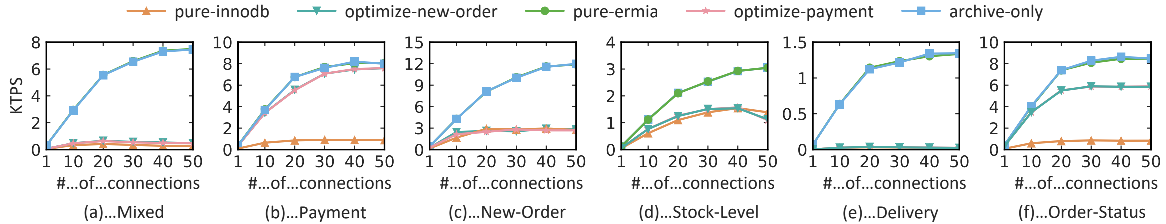

Figure 6.4: Storage-resident(ext4) TPC-C.

6.5 CSR Overhead

Now we study how multi-index CSR may impact the abort rate of different transactions by varying its parameters (the capacity of each index and the number of indexes before recycling is triggered). We conduct experiments using 50 connections under `optimize-payment` and run the TPC-C full mix. The reason we choose this setting is because the default Mix is representative that contains all TPC-C transactions, and 50 connections can fully saturate the server.

Figure 6.5 shows the throughput and abort rate numbers under both storage- and memory-resident configurations, with a varying number of CSR index capacity and recycling threshold settings. As shown by the figure, under extreme cases where CSR Index Capacity is set to 1, i.e., only one entry can be inserted to a CSR index, the abort rates are significantly higher than other cases (at around 4%). This is because more cross-engine transactions are aborted since the entries are recycled frequently and the anchor snapshot is often old, making it likely for transactions to intend to insert into a read-only index and subsequently abort. Note the two different CSR configurations here: (1) CSR Index Capacity = 1 and Recycle Threshold = 5000; (2) CSR Index Capacity = 5000 and Recycle Threshold = 1.

CSR...Index...Capacity	(a) 1	(a) 100	(a) 1000	(a) 5000	(b) 1	(b) 100	(b) 1000	(b) 5000	(c) 1	(c) 100	(c) 1000	(c) 5000
10000	470	466	451	454	1.3	1.6	1.5	1.6	460	500	482	464
5000	455	489	450	447	1.5	1.6	1.6	1.4	456	452	457	453
1000	427	459	465	450	1.5	1.5	1.4	1.7	468	442	445	452
100	437	443	450	448	1.5	1.6	1.5	1.7	479	454	446	467
50	453	449	460	439	1.8	1.8	2	1.8	452	445	447	465
1	430	435	439	444	4.6	4.5	4.7	4.6	456	457	426	463

CSR...Index...Capacity	(d) 1	(d) 100	(d) 1000	(d) 5000	(e) 1	(e) 100	(e) 1000	(e) 5000	(f) 1	(f) 100	(f) 1000	(f) 5000
10000	1.6	1.5	1.6	1.5	813	752	761	766	0.71	0.66	0.67	0.65
5000	1.4	1.6	1.5	1.5	761	858	788	830	0.64	0.7	0.7	0.57
1000	1.3	1.7	1.3	1.4	829	784	764	744	0.66	0.64	0.6	0.7
100	1.7	1.7	1.9	1.6	799	820	744	730	0.83	0.72	0.79	0.8
50	2	1.7	1.9	2	781	837	749	829	0.97	0.91	0.93	0.85
1	4.7	4.6	4.4	4.8	763	787	743	753	4	4	3.7	3.9

Columns: Recycle...Threshold — ...(a)...Throughput(TPS) ext4; ...(b)...Abort...rate(%) ext4; ...(c)...Throughput(TPS) tmpfs; ...(d)...Abort...rate(%) tmpfs; ...(e)...Throughput(TPS) DRAM; ...(f)...Abort...rate(%) DRAM

Figure 6.5: Throughput(TPS) and Abort Rate(%) of TPC-C Default Mixed Transaction with Table Scheme Optimize-payment.

Table 6.1: Abort rate comparison between single-engine and cross engine TPC-C transactions

single-engine	Pure-ermia	0.43%
	Pure-innodb	0.47%
cross-engine	Optimize-new-order	0.61%
	Optimize-payment	0.54%
	Archive-only	0.45%

Both of them have the same “total” CSR capacity (5000), while their abort rates differ by a lot. This is because the overhead of creating a new index is much higher than adding an entry into an existing index. As a result, when a cross-engine transaction is about to find a valid snapshot, the entry is more likely to be recycled in CSR configuration (1) compared to CSR configuration (2). As shown by the figure, we observed the similar trend for all storage setups. Overall, under non-extreme configurations (e.g., 50 index capacity), the multi-index CSR exhibits very low overhead. the overhead of CSR recycle mechanism remains low.

Table 6.1 further shows the abort rates for single-engine TPC-C vs. cross-engine TPC-C. under the memory-resident case since under this setup the contention is low (using equal numbers of warehouses and connections) and so aborts are mainly caused by CSR. The table shows that CSR only aborts less than 0.3% of transactions. We expect this to be tolerable in real world scenarios.

6.6 Latency

To explore the impact of CSR on transaction latency, we collect the results from the storage-resident microbenchmarks done in Chapter 6.2. As shown in Figure 6.6, besides the latency at single connection, the one at 80 connections is also chosen here for analysis because we can infer from Figure 6.1 that the throughput always peaks at 80 connections on our machine. With 80 times more connections, latency unavoidably increases. However, it is still within an acceptable range. The latency remains consistent across all three workloads at single connection (Figure 6.6a); and it increases in proportion to the number of InnoDB

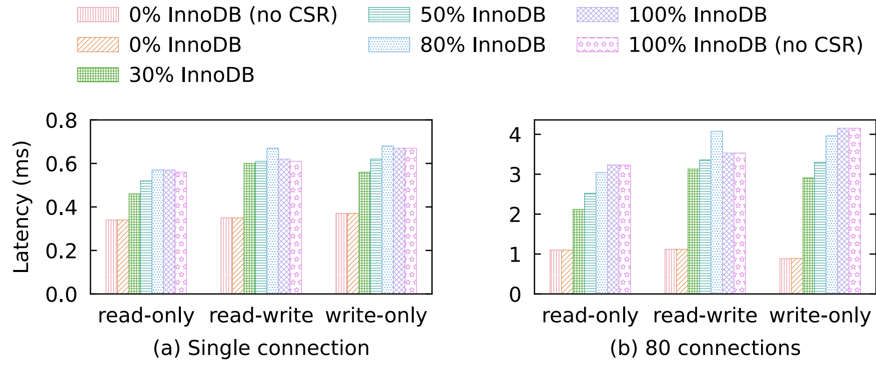

Figure 6.6: 95th percentile latency of microbenchmarks at single connection and 80 connections under varying storage-resident workloads.

sub-transactions at 80 connections (Figure 6.6b). Note that even the latency of 100% InnoDB transactions, which is the worst case, is still comparable to that of 100% InnoDB transactions bypassing CSR. Thus, we conclude that CSR does not significantly impact latency.

Chapter 7

Related Work

Our work builds upon a rich literature in federated and multi-database systems, distributed and replicated SI systems, and modern database engines.

Federated, Multi-database (MDBS) and Polystore Systems. These systems are mostly deployed in a distributed environment, where each DBMS is heterogeneous and residing in different hosts, assuming that, e.g., local databases do not know the existence of each other, no extra information exported for coordination etc. The autonomy and heterogeneity of local database systems (LDBS) make it difficult to enforce the serializability of global transactions in MDBS than in homogeneous distributed database systems. Georgakopoulos et al. [30] focus on enforcing the serializability of global transactions in MDBS by using ticket methods. Superdatabase [47] addresses the consistent update issue by exporting serial order information from element databases at commit time via the commit vote message. Breitbart et al. [12] propose a consistency protocol, which detects cycles in the site graph, where the absence of cycles guarantees the correctness of any execution mix, to resolve consistency and deadlock issues in MDBS. Myriad [31, 38] is a federated database prototype which uses 2PL as local concurrency control scheme, 2PC to ensure serializability, timeout to resolve deadlocks, and SQL for querying. Recent polystore systems [24, 3, 4] are dedicated to supporting cross-model query in the data analytic ecosystem, while Skeena focuses more on transaction-oriented tasks.

Distributed SI and Replication. Prior work on federated snapshot databases identified the anomalies that would lead to inconsistent read views [49]. Binnig et al [8] further identified the cross phenomena under distributed snapshot isolation and proposed incremental distributed SI to correctly support snapshot isolation in a distributed setting. We base our analysis of multi-engine transactions on these results. Compared to prior work, we further extended these results to supporting various isolation levels, including serializability using commit ordering [48]. Elnikety et al. [25] present Generalized Snapshot Isolation (GSI) for replicated databases. GSI differs from the conventional SI in the begin phase where a transaction now uses the older (i.e., latest local) snapshot instead of the latest snapshot, and in the commit phase where an update transaction now checks if there are conflict writes

between its snapshot (rather than start) and commit times. In Skeena, transactions may get an older snapshot from CSR as well, but the commit check only compares commit times.

Modern Database Engines. These engines benefit from the high parallelism that today's multi-core hardware promises, yet still struggles to scale well in presence of bottlenecks like a centralized log buffer. A scalable logging technique [32] that Johnson et al. introduce is a solution tailored to addressing the problem of logging contention by using a combination of early lock release and log flush pipelining, which is applied to our work. SQL Server 2016 [20] supports cross-container transactions, which touch both memory-optimized and disk-based tables, and users are unaware of the table types they are working with in most cases. Nevertheless, with respect to the isolation level schemes, SQL Server does not allow adopting SI uniformly in both engines like what we do in Skeena.

Chapter 8

Conclusion and Future Work

Cross-engine transactions can be very useful. However, prior work does not consider the characteristics of modern “fast-slow” systems. Skeena is an efficient approach to consistent cross-engine transactions in modern fast-slow systems. It consists of a lightweight transaction snapshot tracking and pipelined commit protocol for obtaining consistent snapshots and ensuring atomic commit. Our experience of integrating a memory-optimized engine (ERMIA) into MySQL shows that Skeena can enable cross-engine transactions with few or no changes to core engine code. Experimental results further confirm that the overhead of the whole mechanism is low. And the mechanism is applicable to complex workload such as TPC-C. Future work includes exploring ways to support cross-engine transactions across more than two engines and serializability.

Bibliography

[1] A. Adya, B. Liskov, and P. O'Neil. Generalized isolation level definitions. In *Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073)*, pages 67–78, 2000.

[2] Atul Adya. Weak consistency: A generalized theory and optimistic implementations for distributed transactions, March 1999. PhD Thesis and Technical Report MIT/LCS/TR-786.

[3] Divy Agrawal, Sanjay Chawla, Bertty Contreras-Rojas, Ahmed Elmagarmid, Yasser Idris, Zoi Kaoudi, Sebastian Kruse, Ji Lucas, Essam Mansour, Mourad Ouzzani, et al. Rheem: enabling cross-platform data processing: may the big data be with you! *Proceedings of the VLDB Endowment*, 11(11):1414–1427, 2018.

[4] Rana Alotaibi, Damian Bursztyn, Alin Deutsch, Ioana Manolescu, and Stamatis Zampetakis. Towards scalable hybrid stores: constraint-based rewriting to the rescue. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1660–1677, 2019.

[5] Hillel Avni, Alisher Aliev, Oren Amor, Aharon Avitzur, Ilan Bronshtein, Eli Ginot, Shay Goikhman, Eliezer Levy, Idan Levy, Fuyang Lu, Liran Mishali, Yeqin Mo, Nir Pachter, Dima Sivov, Vinoth Veeraraghavan, Vladi Vexler, Lei Wang, and Peng Wang. Industrial-strength oltp using main memory and many cores. *Proc. VLDB Endow.*, 13(12):3099–3111, August 2020.

[6] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, page 1–10. Association for Computing Machinery, 1995.

[7] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The end of slow networks: It's time for a redesign. *Proc. VLDB Endow.*, 9(7):528–539, March 2016.

[8] Carsten Binnig, Stefan Hildenbrand, Franz Färber, Donald Kossmann, Juchang Lee, and Norman May. Distributed snapshot isolation: Global transactions pay globally, local transactions pay locally. *The VLDB Journal*, 23(6):987–1011, December 2014.

[9] Mihaela A. Bornea, Orion Hodson, Sameh Elnikety, and Alan Fekete. One-copy serializability with snapshot isolation under the hood. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 625–636, 2011.

[10] Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. Scalable garbage collection for in-memory MVCC systems. *Proc. VLDB Endow.*, 13(2):128–141, October 2019.

[11] Yuri Breitbart, Hector Garcia-Molina, and Avi Silberschatz. Overview of multidatabase transaction management. *The VLDB Journal*, 1(2):181–240, October 1992.

[12] Yuri Breitbart and Avi Silberschatz. Multidatabase update issues. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, pages 135–142, 1988.

[13] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.*, 34(4), December 2009.

[14] M. A. Casanova and P. A. Bernstein. General purpose schedulers for database systems. *Acta Inf.*, 15(4):471, August 1981.

[15] A. Chan and R. Gray. Implementing distributed read-only transactions. *IEEE Transactions on Software Engineering*, SE-11(2):205–212, 1985.

[16] Arvola Chan, Stephen Fox, Wen-Te K. Lin, Anil Nori, and Daniel R. Ries. The implementation of an integrated concurrency control and recovery scheme. In *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data*, SIGMOD '82, page 184–191. Association for Computing Machinery, 1982.

[17] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[18] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with snapshot isolation. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, pages 715–726, 2006.

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

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

[21] Alex Depoutovitch, Chong Chen, Jin Chen, Paul Larson, Shu Lin, Jack Ng, Wenlin Cui, Qiang Liu, Wei Huang, Yong Xiao, and Yongjun He. Taurus database: How to be fast, available, and frugal in the cloud. page 1463–1478, 2020.

[22] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: Sql server's memory-optimized oltp engine. pages 1243–1254, 2013.

[23] Deborah J. DuBourdieu. Implementation of distributed transactions. In *Proceedings of the Sixth Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 81–94, 1982.

[24] Aaron J Elmore, Jennie Duggan, Michael Stonebraker, Magdalena Balazinska, Ugur Cetintemel, Vijay Gadepally, Jeffrey Heer, Bill Howe, Jeremy Kepner, Tim Kraska, et al. A demonstration of the BigDAWG polystore system. *Proceedings of the VLDB Endowment*, 8(12):1908, 2015.

[25] Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. Database replication using generalized snapshot isolation. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, SRDS '05, pages 73–84, 2005.

[26] Jose M. Faleiro and Daniel J. Abadi. Rethinking serializable multiversion concurrency control. *Proc. VLDB Endow.*, 8(11):1190–1201, July 2015.

[27] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, June 2005.

[28] Alan Fekete, Elizabeth O'Neil, and Patrick O'Neil. A read-only transaction anomaly under snapshot isolation. *SIGMOD Rec.*, 33(3):12–14, September 2004.

[29] H. Garcia-Molina and K. Salem. Main memory database systems: an overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992.

[30] D. Georgakopoulos, M. Rusinkiewicz, and A. Sheth. On serializability of multidatabase transactions through forced local conflicts. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 314–323, 1991.

[31] S.-Y. Hwang, E.-P. Lim, H.-R. Yang, S. Musukula, K. Mediratta, M. Ganesh, D. Clements, J. Stenoien, and J. Srivastava. The MYRIAD federated database prototype. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, SIGMOD '94, page 518, 1994.

[32] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. Aether: A scalable approach to logging. *Proc. VLDB Endow.*, 3(1):681–692, September 2010.

[33] Alfons Kemper and Thomas Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 195–206, 2011.

[34] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. ERMIA: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1675–1687, 2016.

[35] Hideaki Kimura. Foedus: Oltp engine for a thousand cores and nvram. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 691–706, 2015.

[36] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[37] Justin Levandoski, David Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. High performance transactions in deuteronomy. In *Conference on Innovative Data Systems Research (CIDR 2015)*, January 2015.

[38] Ee-Peng Lim, Sah-Yih Hwang, Jaideep Srivastava, Dave Clements, and M. Ganesh. Myriad: Design and implementation of a federated database prototype. *Softw. Pract. Exper.*, 25(5):533–562, May 1995.

[39] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD ’17, pages 21–35, 2017.

[40] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196, 2012.

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

[42] MySQL 8.0 Reference Manual. Alternative storage engines. 2021.

[43] MySQL 8.0 Reference Manual. SET TRANSACTION statement. 2021.

[44] Oracle. Sysbench benchmark tool, 2021.

[45] Ippokratis Pandis, Pinar Tözün, Ryan Johnson, and Anastasia Ailamaki. PLP: Page latch-free shared-everything OLTP. *Proc. VLDB Endow.*, 4(10):610–621, July 2011.

[46] PostgreSQL Wiki. Foreign data wrappers. 2021.

[47] Calton Pu. Superdatabases for composition of heterogeneous databases. In *Proceedings of the Fourth International Conference on Data Engineering*, pages 548–555, 1988.

[48] Yoav Raz. The principle of commitment ordering, or guaranteeing serializability in a heterogeneous environment of multiple autonomous resource mangers using atomic commitment. In *18th International Conference on Very Large Data Bases Proceedings*, pages 292–312, 1992.

[49] Ralf Schenkel and Gerhard Weikum. Integrating snapshot isolation into transactional federations. In *Cooperative Information Systems*, pages 90–101, 2000.

[50] Ralf Schenkel, Gerhard Weikum, Norbert Weißenberg, and Xuequn Wu. Federated transaction management with snapshot isolation. In *Transactions and Database Dynamics*, pages 1–25, 2000.

[51] Amit P. Sheth and James A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comput. Surv.*, 22(3):183–236, September 1990.

[52] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it’s time for a complete rewrite). pages 1150–1160, 2007.

[53] TPC. TPC benchmark C (OLTP) standard specification, revision 5.11, 2010.

[54] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP ’13, pages 18–32, 2013.

[55] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon Aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1041–1052. ACM, 2017.

[56] Tianzheng Wang and Ryan Johnson. Scalable logging through emerging non-volatile memory. *Proc. VLDB Endow.*, 7(10):865–876, June 2014.

[57] Tianzheng Wang, Ryan Johnson, Alan Fekete, and Ippokratis Pandis. Efficiently making (almost) any concurrency control mechanism serializable. *The VLDB Journal*, 26(4):537–562, August 2017.

[58] Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. Query fresh: Log shipping on steroids. *Proc. VLDB Endow.*, 11(4):406–419, December 2017.

[59] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proc. VLDB Endow.*, 10(7):781–792, March 2017.

[60] Yu Xia, Xiangyao Yu, Andrew Pavlo, and Srinivas Devadas. Taurus: Lightweight parallel logging for in-memory database management systems. *Proc. VLDB Endow.*, 14(2):189–201, October 2020.

[61] Peter Zaitsev. Need to connect to a local mysql server? use unix domain socket!, 2020.

[62] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The end of a myth: Distributed transactions can scale. *Proc. VLDB Endow.*, 10(6):685–696, February 2017.