Sonata: Multi-Database Transactions Made Fast and Serializable

Chuzhe Tang
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
t.chuzhe@sjtu.edu.cn

Zhaoguo Wang
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
zhaoguowang@sjtu.edu.cn

Jinyang Li
New York University
jinyang@cs.nyu.edu

Haibo Chen
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
haibochen@sjtu.edu.cn

ABSTRACT

Today, the wide adoption of distributed service-oriented applications has rendered multi-database transactions increasingly important. They protect cross-service workflows that access multiple database systems from concurrency anomalies and failures. This paper presents Sonata, a new multi-database transaction system that provides high performance, global serializability, and seamless integration with existing applications and database systems. Sonata builds on the theory of commitment ordering to ensure global serializability and uses two-phase commit for atomicity and durability. Instead of treating database systems as black box storage, Sonata reuses existing database systems' concurrency control yet refrains from exposing or modifying their internals. It performs additional non-blocking coordination only at prepare time via application-level shim layers, allowing applications to incorporate Sonata without changing their existing queries or database systems. Evaluation using TPC-C shows that Sonata incurs 7.1% coordination overhead on average and outperforms prior work by up to 1114.3%.

PVLDB Reference Format:
Chuzhe Tang, Zhaoguo Wang, Jinyang Li, and Haibo Chen. Sonata: Multi-Database Transactions Made Fast and Serializable. PVLDB, 18(10): 3449 - 3462, 2025.
doi:10.14778/3748191.3748207

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://ipads.se.sjtu.edu.cn:1312/opensource/sonata.

1 INTRODUCTION

With recent embodiments like microservices and serverless architectures, distributed service-oriented applications are become increasingly popular [45, 47, 50, 63]. In such applications, functionalities are partitioned into standalone, loosely coupled services. Rather than relying on a centralized database, each service is equipped with a dedicated database to manage its own data [47, 49, 52]. As a result, complex workflows inevitably become distributed, touching different services and creating multi-database transactions.

A multi-database transaction consists of subtransactions created by different services, each reading and writing data at a distinct

Proceedings of the VLDB Endowment, Vol. 18, No. 10 ISSN 2150-8097.
doi:10.14778/3748191.3748207

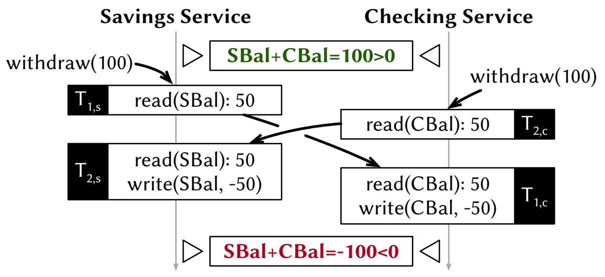

Figure 1: A banking example where cross-service workflows without global serializability violate business constraints.

database system. Figure 1 shows an example of a banking application comprising two database-backed services, *savings* and *checking*, which track users' savings and checking accounts. This application allows overdrafts as long as a user's total balance at both services is not negative. Therefore, to fulfill a withdrawal, one service must read the other's balance before proceeding, creating a multi-database transaction with two subtransactions.

Without proper coordination, multi-database transactions can lead to concurrency anomalies and violate business constraints. In this example, the user initially has $50 each in his two accounts, and there are two concurrent $100 withdrawal requests that interleave differently at the two services. Although each service executes its subtransactions as serializable local transactions (e.g., $T_{1,s}$ and $T_{2,s}$), the whole transactions (e.g., T_1) are not globally serializable. As a result, both services consider the total balance sufficient and authorize the withdrawals, leaving a negative total balance.

For mitigation, multi-database transactions should be globally serializable, e.g., only one withdrawal is allowed as if they were handled sequentially [18]. Without global serializability, the responsibility of ensuring application correctness falls on developers, e.g., by manually blending coordination logic for these database operations into application code, which can be error-prone and challenging [25, 26, 67–70, 76].

Earlier work has proposed multi-database transaction protocols that ensure global serializability, provided each database system ensures local serializability. However, they make conservative assumptions that significantly limit their performance. For example, the ticket method [15, 36] and altruistic locking [9, 62] disallow concurrent execution of subtransactions (e.g., $T_{1,s}$ and $T_{2,s}$) at the same database system. Furthermore, they globally order multi-database transactions that access shared database systems, even when there is no conflict, forcing their subtransactions at different databases to execute in this order. Our evaluation shows that these restrictions often lead to more than 20-fold performance degradation.

For better performance, recent work shifts more concurrency control responsibilities to the application level, imposing restrictions on how applications interact with database systems and sometimes settling for weaker guarantees. Epoxy [46] and Cherry Garcia [28, 29] provide snapshot isolation (SI) across databases. Instead of running subtransactions as serializable local transactions, they perform both intra- and inter-subtransaction coordination in their application-level libraries, using the underlying database systems for storage only. As a result, advanced database features like unique and foreign key constraints and predicate-based updates cannot be used. Furthermore, since they mix additional metadata and data versions with application data, the underlying databases are no longer directly accessible without the requisite application wrapper. ScalarDB [78] extends Cherry Garcia to provide global serializability by either treating all reads as writes or performing an additional Silo [71]-style validation phase to avoid SI anomalies. Therefore, similar restrictions still apply, and additional performance overhead is incurred. In addition, although taking control of transaction coordination unleashes protocol-level parallelism, it can incur great overhead as existing database transaction mechanisms may still be exercised, e.g., relational databases execute all operations in transactions, including those not explicitly declared[1].

This paper proposes Sonata, a high-performance, non-intrusive ACID multi-database transaction system that ensures global serializability. Compared with earlier solutions that assume local serializability [9, 15, 36, 62], Sonata avoids the performance pitfalls through careful reuse of the underlying databases' concurrency control mechanisms. Compared with recent application-level solutions [28, 29, 46, 78], Sonata introduces much less application-level coordination and does not impact application schemas and data, making them directly accessible even without Sonata. For atomicity and durability, Sonata uses two-phase commit (2PC)[18, 38, 48].

Sonata takes a gray-box approach to achieving global serializability. Specifically, we observe that most practical systems use one of two concurrency control families: serializable snapshot isolation (SSI)[21, 22] or strict two-phase locking (S2PL)[18, 32].[2] This allows Sonata to coordinate basing common properties that hold across popular database systems, such as the absence of dangerous structures [33]. Our coordination protocol builds on a condition derived from the theory of commitment ordering [57–59]. It involves only subtransaction prepare and commit events at individual database systems, allowing Sonata to intervene locally only at prepare time while leaving intact the rest of the subtransaction execution and commit. As a result, Sonata can work as application-level shims that require no modification to either the applications or databases. We have proven the correctness of our shims, and our analysis shows that they introduce no false positives, i.e., unnecessary aborts, given that the underlying database systems accurately detect conflicts. Sonata can accommodate other systems, even those providing only

[1]In such cases, operations are executed as single-statement transactions.

[2]SSI systems, mainly PostgreSQL and derivatives [1–5, 17], are fewer in number than S2PL systems yet still widely used. PostgreSQL is the most popular database with a nearly 50% adoption rate by Stack Overflow's Developer Survey 2023 and 2024 [6, 7] and named DBMS of the Year by DB-Engines in 2017, 2018, 2020, and 2023 [10–12, 35].

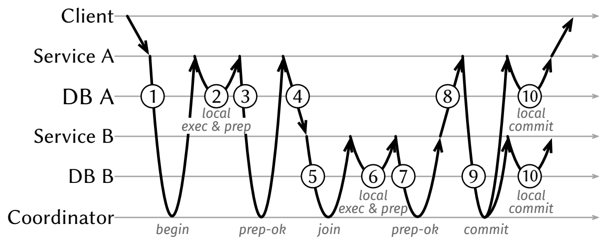

Figure 2: An example workflow of a 2PC-based multi-database transaction. The transaction is initiated at step 1 and committed at step 9. During its lifetime, two subtransactions are registered, executed, and prepared in steps 1–3 and 5–7, respectively, and are finally committed at step 10. Steps 4 and 8 indicate inter-service communication.

single-record atomicity, with generic fallback shims that perform full application-level concurrency control as done in [28, 29, 46, 78].

We have implemented and evaluated Sonata using TPC-C and microbenchmarks in a multi-database setting. Our experience confirms that adapting Sonata does not require changes to the applications' query statements, the database drivers, or the database systems themselves. Our evaluation results show that Sonata adds only 7.1% coordination overhead on average to a 2PC-only baseline, where no cross-subtransaction isolation is guaranteed. We have also compared Sonata with the ticket method [15, 36], ScalarDB [78], and Epoxy [46]. Due to the careful reuse of existing database concurrency control mechanisms, Sonata outperforms them by 74.1% to 1114.3%, 305.5% to 423.9%, and 14.0% to 241.6%, respectively.

To summarize, this paper makes the following contributions.

- A locally enforceable condition for multi-database global serializability and a corresponding coordination framework.
- Two shim layer designs for SSI- and S2PL-based systems that enforce this condition without modifying the database systems.
- Implementation and evaluation of Sonata that demonstrate its low overhead and performance advantage over prior work.

2 BACKGROUND AND MOTIVATION

2.1 Multi-Database Transactions

Multi-database transactions lift the familiar transaction abstraction [18, 39] to coordinate data access across multiple database systems. They are also called *global transactions* to emphasize the distinction with local transactions. Figure 2 shows an example workflow of a 2PC-based multi-database transaction. First, a client request arrives at service A, which initiates the multi-database transaction and thus called the *root service.* ① Service A first notifies the 2PC coordinator with a unique transaction ID of the start of this multi-database transaction and the start of the first subtransaction. ② Then, service A executes its subtransaction as a local transaction and prepares it when the local processing is finished, e.g., via commands like Prepare Transaction <id>. ③ Next, service A notifies the coordinator of the completion of its subtransaction. ④ Service B is then invoked to continue handling the client request with the transaction ID propagated as RPC metadata, e.g., HTTP headers. Like service A, service B ⑤ notifies the coordinator of the presence of its subtransaction, ⑥ initiates a local transaction and prepares it when finished, and ⑦ notifies the coordinator of the completion. ⑧ When service A receives a response and has determined that the processing is successful, ⑨ it signals the coordinator to commit. The coordinator follows 2PC to produce a durable decision and broadcasts it to all participating services. ⑩ Finally, each service commits its subtransaction accordingly.

Ideally, multi-database transactions should enjoy the same ACID properties [18, 39, 41] as single-database transactions. While existing database systems typically support 2PC via standards like XA [77] for atomic commit and durability, global serializability is still missing. Many serializable protocols exist for single-database transactions [24, 31, 44, 56, 73–75]. However, it is challenging to use them for multi-database transactions, as they typically require detailed knowledge about transaction execution that is not exposed or even maintained by individual systems. Meanwhile, existing global serializability protocols are either too conservative [9, 15, 20, 36, 51, 62] or rely on costly application-level concurrency control [78], leading to unsatisfactory performance (§6).

2.2 Transaction Model and Notation

To formally reason about multi-database transactions, we follow and extend the model from [8]. Readers familiar with transaction theories should find our additions straightforward, which are briefly summarized below. We use T_i to denote a multi-database transaction and use $T_{i,j}$ to denote its subtransaction on database system j. When necessary, operations can be similarly subscripted to indicate where they take place, e.g., $w_{i,j}$. As Sonata uses 2PC, we use $p_{i,j}/c_{i,j}/a_{i,j}$ to denote the prepare/commit/abort events of subtransaction $T_{i,j}$. d_i denotes the 2PC decision event of T_i that takes place at the coordinator. A local history contains only the events that take place at a single database system, while a global history contains all events from all participating database systems, plus the 2PC decision events. We use *global* and *local serializability* to refer to the serializability of global and local histories, respectively, i.e., the acyclicity of their corresponding direct serialization graphs. As Figure 1 shows, local serializability at all participating databases does not imply global serializability.

2.3 Commitment Ordering

Commitment ordering (CO)[57, 58] is a property of global histories. A history H satisfies CO if for any two transactions T_i and T_j in H, $T_i \longrightarrow T_j$ implies $d_i <_H d_j$. CO implies global serializability, i.e., a global history that satisfies CO is also serializable [57, Theorem 3.1].[3] Therefore, ensuring that only CO-compliant histories are produced is sufficient for guaranteeing global serializability in a multi-database environment. Equivalently, a history H satisfies CO if for any two subtransactions $T_{i,k}$ and $T_{j,k}$ at the same database system, $T_{i,k} \longrightarrow T_{j,k}$ implies $d_i <_H d_j$ [57, Theorem 5.2]. This definition considers only dependencies among subtransactions on the same database system, which provides an opportunity to enforce CO, and thus global serializability, without communicating dependencies between database systems.

[3]The original formulation considers transitive transaction dependencies, while our model does not. The theorem holds still with an almost identical proof.

Yet, simply combining existing databases with 2PC does not guarantee CO. As the definitions suggest, enforcing these conditions requires controlling the order of 2PC decisions of conflicting global transactions. However, individual databases can only control local transaction execution. It is the 2PC coordinator that controls the decision order of global transactions, which is often simply the order in which global transactions' prepare messages are fully received.

Long-term locks used in existing systems could partially intervene in the decision order but are insufficient to enforce CO. For example, in an SSI system like PostgreSQL, writing to a data item by one subtransaction does not prevent another subtransaction from reading it. This is because SSI systems read from a snapshot by default unless locking is explicitly requested via SQL clauses like For Share. As a result, an rw-dependency is established between the two concurrent subtransactions, and both can proceed to prepare without blocking each other, allowing a CO violation: $r_1(x_0), w_2(x_2), p_2, d_2, p_1, d_1$, where $T_1 \longrightarrow T_2$ but d_2 precedes d_1. CO violations are also possible in S2PL systems like MySQL (§3.4.2).

Prior work has proposed algorithms based on serialization graphs to enforce CO [57–59]. They require database systems to explicitly track the dependencies among active transactions and export a centralized view to a commit scheduler. However, to the best of our knowledge, no existing database fully tracks transaction dependencies and either exposes them to the outside or allows pluggable commit schedulers. Therefore, instead of implementing these serialization graph-based algorithms within existing databases, we propose to enforce CO from the outside with lightweight shims.

3 SONATA DESIGN

Sonata is a middleware system between service-oriented applications and database systems that provides full ACID guarantees, notably global serializability, for multi-database transactions.

3.1 Opportunities

Sonata is enabled by two key insights. First, we derive a sufficient condition for CO that permits local enforcement at individual database systems without involving the 2PC coordinator. Specifically, this condition states that, for any two committed subtransactions $T_{i,k}$ and $T_{j,k}$ at any database system k, $T_{i,k} \longrightarrow T_{j,k}$ implies $c_{i,k} < p_{j,k}$.[4] The sufficiency is straightforward. By 2PC, a decision event is ordered before all corresponding subtransaction commit events and after all corresponding subtransaction prepare events. Therefore, $d_i < c_{i,k}$ and $p_{j,k} < d_j$, which implies $d_i < d_j$, satisfying the alternative CO definition shown in §2.3. With this condition, CO of global histories, and thus global serializability, can be achieved by merely controlling the order of prepare and commit events that are local to each database system.

Second, we observe that existing database systems deployed in practice typically use either SSI or S2PL for concurrency control. While specific implementation details vary, important properties hold for all systems. For example, while a multi-version storage engine with various optimizations is used in MySQL, the classic principle of two-phase well-formed transactions [32] is still the

[4]This condition differs from the alternative CO definition in §2.3 as all events considered here are local to each database system.

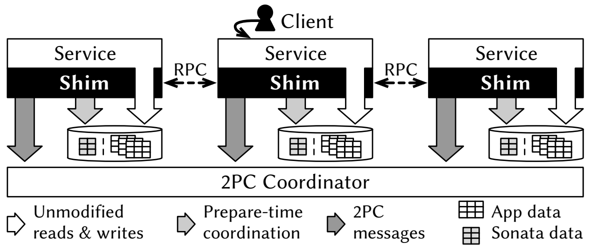

Figure 3: Sonata architecture.

```
1  @GlobalTransactional
2  CHECKING::withdraw(user_id, amount):
3    s_bal := invokeRemoteService(SAVINGS.readBal, user_id)
4    invokeInLocalTxn(lambda:
5      c_bal := SELECT bal FROM Checking WHERE user_id = user_id
6      if c_bal + s_bal < amount:
7        raise InsufficientFundsException
8      UPDATE Checking SET bal = c_bal-amount WHERE user_id = user_id )
9  SAVINGS::readBal(user_id):
10   invokeInLocalTxn(lambda:
11     s_bal := SELECT bal FROM Savings WHERE user_id = user_id
12     return s_bal )
```

Figure 4: An example of cross-service withdrawal using Sonata multi-database transaction API.

foundation of its local serializability guarantee. Similarly, the absence of dangerous structures [33] holds for all SSI systems despite differences in the specific mechanisms used to detect and prevent them [21, 22, 55]. Therefore, this observation enables us to take a gray-box approach to global serializability that leverages common properties of popular database systems.

3.2 System Overview

Architecture. Figure 3 shows the architecture of a Sonata application. Any service can receive client requests, incorporate other services through RPCs, and wrap the processing with multi-database transactions. Sonata shims sits at the application level and communicates a 2PC coordinator to initiate, participate in, and complete multi-database transactions on behalf of the services. Sonata shims intercept RPC messages to incorporate and propagate transaction information, and they monitor local transactions to impose prepare-time coordination. Sonata's coordination requires a single per-database table that is transparent to the application. As a result, reads and writes are issued unmodified during execution, and the original schemas and data remain intact as well.

Interface. A @GlobalTransactional annotation is provided for marking functions and all service invocations nested within as multi-database transactions. Figure 4 shows how the cross-service withdrawal from the banking example in Figure 1 can be programmed. The annotated withdraw function first invokes the readBal function in the SAVINGS service to read the savings balance in a local transaction in SAVINGS's database. Then, it executes a local transaction in CHECKING's database to read the checking balance and updates it if the total balance is sufficient. Due to the

```
13 invokeInGlobalTxn(func):
14   gtid := genUuid()
15   setThreadLocal(GTID, gtid)
16   begin2pc(gtid)
17   res := invoke(func)
18   commit2pc(gtid)
19   unsetThreadLocal(GTID)
20   return res
21 invokeInLocalTxn(func):
22   gtid := getThreadLocal(GTID)
23   stid := genUuid()
24   if gtid is not NULL:
25     registerSubTxn(gtid, stid)
26     res := SHIM.invokeAsSubTxn(func, gtid, stid)
27     updateSubTxn(gtid, stid, PREPARED)
28   else:
29     res := SHIM.invokeAsSubTxn(func, NULL, stid)
30   return res
31 invokeRemoteService(target, request):
32   gtid := getThreadLocal(GTID)
33   if gtid is not NULL:
34     request.setHeader(GTID, gtid)
35   return target.invoke(request)
36 handleRemoteInvocation(request):
37   gtid := request.getHeader(GTID)
38   if gtid is not NULL:
39     setThreadLocal(GTID, gtid)
40     res := handle(request)
41     unsetThreadLocal(GTID)
42   else:
43     res := handle(request)
44   return res
```

Figure 5: Sonata procedures for initiating, propagating, and completing multi-database transactions.

@GlobalTransactional annotation, these two local transactions are treated as subtransactions of the same global transaction.

Assumptions. Sonata assumes that the underlying databases support serializable local transactions, use SSI or S2PL for concurrency control, and implement 2PC participant procedures. S2PL systems could employ early lock release, but only for read-only transactions. Popular systems, including MySQL, PostgreSQL, and SQL Server, typically meet these requirements.

3.3 Sonata Workflow

Sonata follows the same overall 2PC workflow as shown in §2.1. @GlobalTransactional-annotated functions are intercepted and executed with the invokeInGlobalTxn procedure, as shown in Figure 5. This procedure first generates a global transaction ID (line 14) and sets it as a thread-local variable, GTID (line 15). Sonata assumes that each service invocation is exclusively bound to a single thread of execution but does not require any specific threading implementation. Therefore, GTID unambiguously indicates the presence of a global transaction for the associated service invocation. The procedure then notifies the 2PC coordinator to begin a global transaction (line 16) and executes the function (line 17). Any local transaction created during execution will be treated as a subtransaction of the global transaction. Upon completion, the procedure notifies the 2PC coordinator to commit the global transaction (line 18) and clears the current thread's GTID (line 19). Commit or abort decisions will be sent to corresponding shims to complete registered subtransactions.

Sonata intercepts local transactions and executes them using the invokeInLocalTxn procedure. This procedure checks if the current thread is already within a global transaction by inspecting the current thread's GTID (line 22) and generates a unique subtransaction ID (line 23). If a global transaction is present, this subtransaction is registered with the 2PC coordinator (line 25). A database system-specific shim is then invoked to execute the function within a local transaction and prepare it when finished (line 26). This shim ensures that the CO condition from §3.1 is maintained. Then, the 2PC coordinator is notified that the subtransaction has been successfully prepared (line 27). If no global transaction is active, no global transaction ID is given to the shim (line 29). Calling into the shim is necessary as additional coordination might be necessary, e.g., when the selective coordination optimization (§5) is disabled.

To propagate multi-database transactions across services, Sonata intercepts service invocation and request handling with the invokeRemoteService and handleRemoteInvocation procedures. The first procedure attaches the current thread's GTID to the request header before sending it to the target service (line 34). Sonata does not require specific communication protocols or message formats as long as GTID can be included, e.g., as a string field. Upon receiving the request, the target service retrieves GTID from the request header (line 37) and sets it as a thread-local variable before handling the request (line 39). This way, local transactions at remote services can be consistently handled and participate in the same global transaction. Like in invokeInGlobalTxn, the current thread's GTID is cleared after the request is handled (line 41).

3.4 Commitment Ordering Shims

3.4.1 SSI Shim Layer. SSI is a family of concurrency control protocols that ensures local serializability. It restricts an SI protocol to produce only serializable histories [21, 22, 55][5]. In SI, read operations always return the latest committed versions from a snapshot taken at the beginning of the local transaction [16], eliminating the need for long-term read locks to block concurrent writes. SSI extends existing SI protocols with mechanisms to prevent dangerous structures [33], i.e., consecutive rw-dependencies $T_1 \xrightarrow{\text{rw}} T_2 \xrightarrow{\text{rw}} T_3$ among committed local transactions such that T_1 and T_2 are concurrent and T_2 and T_3 are concurrent. Regardless of the specific detection and prevention mechanisms, an SSI system must guarantee the absence of dangerous structures in the histories produced.

We observe that, in an SSI system, CO violations arise when local transactions have rw-dependencies, while ww- and wr-dependencies do not lead to CO violations. Consider two committed local transactions T_1 and T_2. If $T_1 \xrightarrow{\text{ww}} T_2$, then T_1 must commit before T_2 starts; otherwise, they would have been concurrent, which is disallowed in SI. If $T_1 \xrightarrow{\text{wr}} T_2$, then T_1 must commit before T_2 starts as well; otherwise, the version created by T_1 would not have been in the snapshot of T_2. In both cases, T_1 is decided before T_2, and CO is

[5]Although named PSSI (Precisely SSI), the algorithm from [60] is essentially a graph-based detection algorithm for dependency cycles, not dangerous structures. It is not used in practice due to its overhead [55]. Thus, we do not consider it in this paper.

```
45 SSI::invokeAsSubTxn(func, gtid, stid)
46   Begin Transaction
47   res := invoke(func)
48   dummy_key := prepareHelper(gtid, stid)
49   Update Dummy Set value = rand() Where key = dummy_key
50   if gtid is not NULL:
51     Prepare Transaction gtid + stid
52   else: # directly commit if not part of a global transaction
53     Commit Transaction
54     Abort Prepared HELPERS[gtid + stid]
55   return res
56 SSI::subTxnCommit(gtid, stid)
57   Commit Prepared gtid + stid
58   Abort Prepared HELPERS[gtid + stid]
59 prepareHelper(gtid, stid)
60   dummy_key := uniqueRand()
61   Begin Transaction # not nested in the caller's transaction
62   Select * From Dummy Where key = dummy_key
63   hid := genUuid()
64   HELPERS[gtid + stid] := hid
65   Prepare Transaction hid
66   return dummy_key
```

Figure 6: SSI shim procedures.

maintained. However, if $T_1 \xrightarrow{\text{rw}} T_2$, due to the absence of long-term read locks, CO violations might arise, as in the history shown in §2.3: $r_1(x_0), w_2(x_2), p_2, d_2, p_1, d_1$, where T_1 is decided after T_2.

Interestingly, rw-dependencies are the only ones that constitute dangerous structures, presenting an opportunity for reusing existing SSI facilities by turning CO violations into dangerous structures. At a high level, our shim introduces temporary rw-dependencies (dashed below) towards ready-to-prepare local transactions (T_i/T_j) using additional helper transactions (T_i'/T_j'):

$$\begin{tikzcd} T_i' \arrow[dr, dashed, "\text{rw}"] & & T_j' \arrow[dr, dashed, "\text{rw}"] & \\ & T_i \arrow[rr, "\text{rw}"] & & T_j \end{tikzcd}$$

As a result, any pair of concurrent transactions that already has an existing rw-dependency is promoted to a potential dangerous structure, e.g., $T_i' \overset{\text{rw}}{\dashrightarrow} T_i \xrightarrow{\text{rw}} T_j$. In the case of a CO violation, i.e., T_j decides before T_i, T_i will be aborted by the underlying SSI database's dangerous structure prevention mechanism; otherwise, the temporary rw-dependency, $T_i' \overset{\text{rw}}{\dashrightarrow} T_i$, will be removed and T_j will be able to prepare and decide afterward.

Figure 6 shows the specific algorithm. The invokeAsSubTxn procedure starts a local transaction (line 46) and executes the given function (line 47). Before preparing the transaction, another local transaction is spawned by the prepareHelper procedure (line 61). This helper transaction reads a random row from a two-column table Dummy maintained by Sonata (line 62) and prepares itself (line 65).[6] Helper transaction IDs are randomly generated (line 63) and stored in a service-local in-memory map HELPERS indexed by the global transaction ID and subtransaction ID (line 64). After preparing the helper, the original subtransaction updates the same

[6]Dummy key uniqueness is not required for correctness, but helps avoid false positives and head-of-line blocking. Thus, unique keys are used in Sonata.

row with a random value (line 49), establishing an rw-dependency with the helper. If the global transaction ID is present, the original subtransaction prepares itself (line 51). When committing or aborting the subtransaction, Sonata aborts the associated helper transaction (line 58). If the global transaction ID is NULL, indicating that a single-database transaction should be used, the shim directly commits the local transaction (line 53) and aborts the helper (line 54) as if this transaction is prepared and decided immediately.

We now revisit the previous example and provide an intuitive argument for the shim's correctness. If T_j prepares and decides before T_i, which violates CO, since the helper T_i' is prepared before T_i, the database's dangerous structure prevention mechanism will not allow T_i to prepare. Otherwise, once all three local transactions are prepared, though they have not yet strictly formed a dangerous structure, it is up to the 2PC coordinator, not the database system, to make the commit decision. Suppose the coordinator decides to commit all three transactions. In that case, the database faces a dillema: it must either proceed as decided and keep the dangerous structure, thereby violating its local serializability guarantee, or abort one of the prepared transactions, which would break the 2PC protocol. For the same reason, T_j cannot prepare either if T_i prepares earlier until the helper T_i' is aborted, which happens after T_i is decided at the 2PC coordinator. After that point, preparing and deciding T_j will not cause CO violations.

3.4.2 S2PL Shim Layer. S2PL is a classic family of pessimistic concurrency control protocols that ensure local serializability. It inherits the two-phase property from 2PL, meaning long-term read/write locks are only acquired in the first phase and released in the second [32]. Being strict means that all write locks are held until the local transaction commits or aborts.

We observe that, in an S2PL system, CO violations can only arise from rw-dependencies as well. Since write locks are held until commit, similar to SSI systems, no ww- and wr-dependencies are possible between two committed concurrent local transactions, and thus, they cannot violate CO. However, a S2PL system does not necessarily hold long-term read locks until commit. When releasing them early, e.g., immediately after prepare [37], CO violations can occur. For example, consider two local transactions T_1 and T_2 such that $T_1 \xrightarrow{\text{rw}} T_2$ and T_1 has prepared but not committed yet. If T_1's read lock is released immediately after it prepares, T_2, which modifies the data item that T_1 has read, can prepare and commit before T_1 commits, violating CO: $r_1(x_0), p_1, w_2(x_2), p_2, d_2, d_1$. If the read lock is held until commit instead, such violations are not possible.

Fortunately, many S2PL systems do not release read locks early (e.g., SQL Server), and others only implement it for read-only local transactions (e.g., MySQL and Db2).[7] Therefore, our S2PL shim only needs to introduce a dummy write operation to read-only local transactions to disable early lock release for the latter systems. Figure 7 shows the full procedures. The dummy write is performed on a randomly chosen row in a two-column table DUMMY that is maintained by Sonata. A simple heuristic to determine whether a local transaction is read-only is to check whether the local transaction contains any statements other than SELECT. In some systems like MySQL, seemingly read-write local transactions are treated as

[7]We restrict our attention to the top 10 database systems from the DB-Engines Ranking.

```
S2PL::invokeAsSubTxn(func, gtid, stid)
  Begin Transaction
  res := invoke(func)
  if gtid is not NULL:
    if isReadOnly():
      Update Dummy Set value = rand() Where key = uniqueRand()
    Prepare Transaction gtid + stid
  else: # directly commit if not part of a global transaction
    Commit Transaction
  return res
S2PL::subTxnCommit(gtid, stid)
  Commit Prepared gtid + stid
```

Figure 7: S2PL shim procedures.

read-only ones internally if there is no change in concrete values, rendering above heuristic insufficient. For such systems, dummy writes are added to all local transactions for such systems. While we are unaware of any S2PL systems for which dummy writes are insufficient, for completeness, mitigations for such cases include SQL clauses like FOR UPDATE that upgrade read locks to write locks, shim-layer read locks that take effect only after the owner local transactions have prepared, and fallback shims as discussed in §3.6.

3.5 Durability and Failure Recovery

Sonata follows the standard 2PC protocol to persist and recover both the participant and coordinator states [18, 38, 48]. Additionally, after a participant has finished 2PC recovery, Sonata must handle potentially dangling prepared helper transactions due to the lost in-memory HELPERS map of the SSI shim. Specifically, after restarting, the participant blocks all requests that would create new subtransactions. It periodically polls the 2PC coordinator for any decisions on subtransactions that the participant has previously prepared. When the 2PC coordinator signals that no such subtransactions exist, prepared transaction still pending at the participant are the dangling helpers. The participant then queries the database for such transactions and aborts them, e.g., using the XA RECOVER command in MySQL or querying the pg_prepared_xacts table in PostgreSQL. Finally, the participant resumes normal operation.

3.6 Discussion

Supporting Other Databases. Databases beyond SSI and S2PL can be accommodated as well. If the database provides transactions (not necessarily with local serializability), a fallback shim that serializes them via sequential execution can trivially enforce CO. When local transactions are unavailable, notably in NoSQL systems, generic fallback shims that perform application-level CO-compliant concurrency control, e.g., S2PL without early lock release, can be used. Designing such shims is straightforward and should resemble those from [28, 29, 46, 78], requiring only single-record atomicity. We expect performance overhead to be similar to the gap between Epoxy/ScalarDB and Sonata shown in §6. Nevertheless, this overhead is restricted to generic shims only; SSI and S2PL systems in the same cluster are unaffected.

Changes in Database Internals. Changes in database internals, although unlikely for stabilized mainstream systems, may affect Sonata's effectiveness. For example, in principle, an SSI database may conservatively abort any helper on a potential dangerous structure, forcing the shim to abort the corresponding local transaction. While the correctness is intact (§4.1), unnecessary aborts may appear and degrade performance. Yet, PostgreSQL's safe retry property [55] forbids this change, as it always tries to abort the middle transaction, which would never be the helper, to break dangerous structures. Similarly, examining the source code of MySQL, we find its early lock release behavior also unlikely to change, e.g., extending to read-write transactions. MySQL checks if a finishing transaction has an empty redo log and, if so, directly rolls back the transaction to reduce disk IO, early releasing any read lock acquired as a side effect. Therefore, extending to read-write transactions, which always have non-empty redo logs, would require building new mechanisms from scratch.

Enforcing CO inside Databases Enforcing CO inside databases when source code is available potentially reduces shim layer overhead if tailored modifications based on our shim design are used, instead of prior generic serialization graph-based algorithms [57–59]. For example, PostgreSQL already maintains per-transaction pointers for rw-dependencies. Therefore, our local condition can be maintained by checking the prepare/commit state of the pointed transactions at prepare time and aborting the to-be-prepared transaction when the pointed ones have already prepared or committed.

4 CORRECTNESS ANALYSIS

4.1 Global Serializability

In this section, we prove the correctness of Sonata, namely that it ensures global serializability for multi-database transactions. As discussed in §2.3, CO implies global serializability. Therefore, we prove that Sonata produces CO-compliant histories only. Specifically, we show that each shim satisfies our local condition derived in §3.1, which is a sufficient condition for CO, and we restate below.

Definition 1. A system is locally CO if, for any two subtransactions T_i and T_j in its local history H, $T_i \rightarrow T_j$ implies $c_i <_H p_j$.

4.1.1 *Correctness of SSI Shim.*

THEOREM 4.1. *An SSI system with local transactions intercepted by Sonata SSI shim is locally CO.*

Before proving the theorem, we present two useful properties adapted from [33]. They are originally proposed to describe SI histories. SSI histories are a subset of SI histories, as dangerous structure detection mechanisms do not affect transaction scheduling during execution. Therefore, these properties also hold for SSI histories.

LEMMA 4.2 (LEMMA 2.2 IN [33]). *In an SSI local history H, if there are two local transactions T_i and T_j such that $T_i \rightarrow T_j$, then T_i starts before T_j commits.*

LEMMA 4.3 (LEMMA 2.3 IN [33]). *In an SSI local history H, if there are two local transactions T_i and T_j such that $T_i \rightarrow T_j$ and they are concurrent, then $T_i \xrightarrow{rw} T_j$.*

PROOF OF THEOREM 4.1. A local CO violation in a local history H is defined as a local transaction dependency $T_i \rightarrow T_j$ where $p_j <_H c_i$. We first show that, for any local CO violation in a vanilla SSI system, such T_i and T_j must be concurrent, and the dependency must be an rw-dependency. Assume, for contradiction, that T_i and T_j are not concurrent. There are two possibilities: (i) T_i commits before T_j starts, or (ii) T_i starts after T_j commits. In case (i), T_i would be decided before T_j, which contradicts the premise that T_i and T_j constitute a local CO violation and thus $p_j <_H c_i$. Case (ii) contradicts Lemma 4.2. Therefore, T_i and T_j must be concurrent, and, by Lemma 4.3, $T_i \xrightarrow{\text{rw}} T_j$.

We next show that, with the algorithm in Figure 6, concurrent local transactions T_i and T_j cannot both commit in local history H if $p_j <_H c_i$ and $T_i \xrightarrow{\text{rw}} T_j$. Assume, for contradiction, that such T_i and T_j both commit, meaning they both prepare successfully along with their helpers T_i' and T_j'. There are two possibilities: (i) $p_i <_H p_j$ or (ii) $p_j <_H p_i$. In case (i), since $p_j <_H c_i$, when T_j prepares, both T_i and its helper T_i' are in a prepared state. Given that prepared local transactions cannot be unilaterally aborted by the database system, they are conservatively treated as committed by the dangerous structure detection mechanism of SSI [55]. Consequently, T_j cannot successfully prepare; otherwise, a dangerous structure $T_i' \xrightarrow{\text{rw}} T_i \xrightarrow{\text{rw}} T_j$ is permissible, where the former dependency is established by the dummy read and dummy write in T_i and its corresponding helper T_i', respectively, according to the algorithm in Figure 6. In case (ii), when T_i prepares, T_j is in either a prepared or a committed state. Since T_i's helper is prepared before T_i, T_i cannot successfully prepare; otherwise, the same dangerous structure as described above is permissible as well. In both cases, T_i and T_j cannot both commit. Therefore, local CO violation is impossible, and by Definition 1, the system is locally CO. □

4.1.2 *Correctness of S2PL Shim.*

THEOREM 4.4. *An S2PL system with local transactions intercepted by Sonata S2PL shim is locally CO.*

PROOF. Assume, for contradiction, that T_i and T_j constitute a local CO violation in an S2PL system with the Sonata shim. Since S2PL produces only well-formed local histories [32], T_i must have acquired the corresponding long-term locks to either read or write a data item before T_j accesses the same item. In the case that a write lock is acquired for the data item that establishes the dependency in the CO violation, due to the strictness property of S2PL [18], this lock is held and blocks T_j until T_i commits. Therefore, T_i commits before T_j prepares, which contradicts the premise that T_i and T_j constitute a local CO violation. In the case of a read lock, due to the algorithm in Figure 7 that disables early lock release, this lock is also held until T_i commits, and the same contradiction arises. Therefore, local CO violation is impossible in an S2PL system with the Sonata shim. By Definition 1, this system is locally CO. □

4.2 False Positives

False positives refer to local transactions aborted by Sonata shims whose commit would not have caused local CO violations. Unlike correctness, which can be proven based on common properties, analyzing false positives requires concrete implementation context. We use PostgreSQL and MySQL as representative systems and show that false positives are only possible when these systems fail to determine transaction conflicts accurately.

We begin assuming databases accurately detect conflicts. In PostgreSQL, serializable local transactions are aborted when (i) deadlocks occur, (ii) SI anomalies occur, or (iii) dangerous structures are detected. Helpers in the SSI shim are read-only and non-blocking. Thus, they cannot cause deadlocks or SI anomalies, i.e., violation of the first-committer-wins rule for concurrent writers [16]. We now consider the last case. Let $T_1 \xrightarrow{\text{rw}} T_2 \xrightarrow{\text{rw}} T_3$ be the dependency chain of the dangerous structure that is caused by a helper and triggers a PostgreSQL abort. Since helpers are read-only, only T_1 can be the helper (of T_2), as others sit on the writer ends of rw-dependencies. PostgreSQL intervenes to break the dangerous structure only after T_3 prepares [55]. Since T_3 has already prepared, depending on when PostgreSQL detects the dangerous structure, either T_1 or T_2 will be aborted by PostgreSQL and the other will be aborted by our shim. If T_2 was not aborted and able to commit, since T_3 has already prepared, such a situation is always a local CO violation by Definition 1. Thus, such aborts are not false positives.

MySQL aborts serializable local transactions when deadlocks occur. Suppose a deadlock is caused by the S2PL shim's dummy write. Then, there are at least two transactions T_1 and T_2 on the deadlock wait-for cycle, where T_1 waits for a lock held by T_2 on the dummy table, and T_2 waits for another lock held by some transaction. Since T_2 has already acquired a lock on the dummy table, which is only accessed by the S2PL shim's prepare-time coordination, T_2 must have executed the last dummy write and all preceding operations. As a result, T_2 cannot be waiting for any additional locks, making deadlocks caused by dummy writes impossible. Therefore, the S2PL shim introduces no new aborts and thus no false positives.

Prior conclusions assume accurate conflict detection, which might not always hold for PostgreSQL. PostgreSQL detects rw-dependencies with tuple-level locks by default and will promote them to coarser-grained ones when space is constrained [55], causing false positives. For example, for the dependency chain mentioned above, T_2 might not be the subtransaction associated with the helper, i.e., a different dummy key is used. In contrast, MySQL does not promote lock granularity and always acquires row-level locks for dummy writes, avoiding false positives.

5 OPTIMIZATIONS

Helper Sharing. The basic SSI shim creates a helper transaction for each subtransaction. Although this helper is read-only, disk writes are still required for 2PC prepare states. This optimization allows a batch of N subtransactions to share a single helper transaction, which now reads N distinct random rows from the Dummy table. The N dummy keys and the helper ID are recorded as global variables. Each time prepareHelper is invoked, it attempts to return an unused dummy key and only spawns a new helper if all keys are consumed. Similarly, subTxnCommit only aborts the helper if all associated subtransactions have been completed. This design effectively amortizes the helper prepare overhead. Since the helper lifespan still covers all associated subtransactions from their prepare to commit, the correctness of the algorithm is not affected.

Selective Coordination. Our basic algorithms add helpers and dummy table operations to all local transactions, even those not participating in global transactions. Fortunately, it has been shown that the requirement for multi-database transactions to be decided in their dependency order is only necessary when more than one subtransaction is involved at both ends of a dependency [59]. Therefore, Sonata coordination can be skipped for single-database transactions while still preserving global serializability.

6 EVALUATION

6.1 Evaluation Setup

6.1.1 Sonata Implementation. We have prototyped Sonata in Java based on Apache Seata, an open-source distributed transaction middleware [13]. We introduced less than 600 lines of code of changes to perform prepare-time coordination and maintain metadata like the HELPERS map. Deadlock detection for multi-database transactions is an orthogonal problem, and we use a simple 5-second timeout to break all potential deadlocks.

6.1.2 Baselines. The Ticket method [36] is a classic globally serializable multi-database transaction protocol. It takes a black-box approach, only requiring local serializability from local databases. It coordinates subtransactions with ticket operations, i.e., reading and incrementing a per-database shared integer counter. ScalarDB [78] is a recent globally serializable multi-database transaction middleware using application-level concurrency control. It abstracts local databases with a Bigtable [23]-like key–value (KV) model and performs Silo [71]-style concurrency control. The open-source implementation (version 3.14.0) is used for evaluation. Epoxy [46] is a recent multi-database transaction protocol that provides SI also with application-level concurrency control. A PostgreSQL server is required as the primary database for global coordination. Additional database servers are as simple multi-version storage and termed secondary databases. We ported its open-source implementation, including its two optimizations: (1) subtransactions at secondary nodes are executed in local transactions with default isolation level, and (2) snapshots are cached for read-only transactions. The 2PC baseline runs the original Apache Seata middleware [13], providing no cross-subtransaction isolation but only atomicity and durability. The Local baseline discards both cross-subtransaction coordination and 2PC. Subtransactions are executed as serializable local transactions and commit independently. Thus, this baseline guarantees no ACID property and serves as a performance upper bound.

6.1.3 Testbed and Configuration. By default, a three-node cluster is used for evaluation: one benchmark client and two service nodes. Each service node runs a workload server and an exclusive, colocated database. Multi-database transactions are initiated from one of the service nodes. Service nodes can be invoked multiple times to simulate scenarios with more than two database systems, creating multiple subtransactions in the same database system. Each node is a Huawei Cloud ECS c7n.2xlarge.2 instance with 8 vCPUs, 16 GiB memory, and virtualized SSD storage, running Ubuntu 22.04. One service node deploys MySQL 8.0.39, and one runs PostgreSQL 14.13. Both database systems have undergone basic tuning using MySQLTuner [42] and PgTune [72].

Each database system maintains a 1M-row dummy table, which takes around 56 MB for PostgreSQL and 28 MB for MySQL. The helper sharing optimization uses a batch size of 10. For a fair comparison with Epoxy, which uses PostgreSQL to coordinate atomic

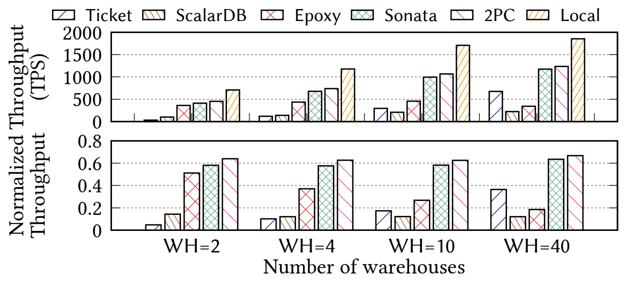

Figure 8: TPC-C peak throughput when PT=WH.

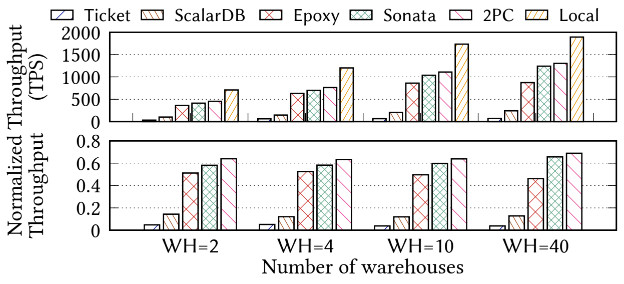

Figure 9: TPC-C peak throughput when PT=2.

commit, we by default colocate coordinators in other baselines, if any, on the same service node that runs PostgreSQL.

6.1.4 Workloads and Measurement. All data points are measured by running each experiment 3 times and taking the average. Each run lasts 160 seconds, with the first 30-second warm-up and the last 10-second cool-down periods excluded from measurement.

Multi-Database TPC-C. TPC-C models an OLTP application consisting of geo-distributed business regions, each with a warehouse serving nearby customers. We partition TPC-C by warehouse into different regional services; half use MySQL, and the other half use PostgreSQL. This turns TPC-C's two most frequent transactions, new-order and payment, into multi-database transactions constituting 88% of the workload. Our TPC-C implementation is based on BenchBase, formerly OLTPBench [30]. For ScalarDB, we use its open-source TPC-C implementation, which also derives from BenchBase but replaces SQL with its customized APIs.

Microbenchmarks. Our microbenchmarks derive from the following parameterized template. Each multi-database transaction consists of S subtransactions, each performing R random reads followed by W random updates to a N-row table. Each subtransaction accesses a different table, so changing S does not affect the per-subtransaction contention level. Each table has two integer columns, one for the primary key and one for the value. Each update sets the value to a random integer. Each subtransaction accesses a randomly chosen database system with probability P to be PostgreSQL.

6.2 TPC-C Performance

One Warehouse per Partition. We first consider the case where the number of partitions (PT) equals the number of warehouses (WH). Figure 8 shows the results. Local achieves the highest peak throughput, as it does not incur any cross-subtransaction coordination. Sonata and 2PC have similar throughput: Sonata achieves 58.2% to 63.4% of Local's peak throughput, while 2PC achieves 62.5% to 66.7%. Compared with 2PC, Sonata incurs an average 7.1% reduction in throughput. Epoxy performs progressively worse as the number of warehouses/partitions increases, peaking at two warehouses with 51.0% of Local's throughput and dropping to 18.6% at 40 warehouses. This translates to a 14.0% to 241.6% performance advantage of Sonata over Epoxy. Such degradation is due to the increasing number of partitions and subtransactions at secondary databases, where record updates are amplified as an insert followed by an update and queries are taxed by additional Where clauses for selecting the version visible to the transaction. Furthermore, application-level concurrency control is performed for each secondary subtransaction. For ScalarDB, the amount of work for its application-level concurrency control is unaffected by the number of subtransactions. Thus, it shows stable performance but at a lower throughput than Sonata and 2PC, ranging from 12.0% to 14.3% of Local. Compared with ScalarDB, Sonata achieves up to 423.9% higher throughput. As a later latency breakdown will show, ScalarDB's overhead comes from expressing transaction logic through KV interfaces and its application-level concurrency control. Ticket performs progressively better as the number of partitions increases, starting at a normalized throughput of 4.8% at two partitions and reaching 36.4% at 40 partitions. This translates to a 74.1% to 1114.3% performance advantage of Sonata over Ticket. With more partitions, the contention on the per-partition ticket counters is reduced, and the performance thus improves.

Fixed Number of Partitions. We next consider the case where the number of partitions is fixed to two, the minimal number where using multi-database transactions is meaningful. This setup permits at most two subtransactions in a multi-database transaction. Figure 9 shows the results. Compared with the previous case, the performance of ScalarDB, Sonata, 2PC, and Local remains similar, as their performance is mainly decided by the work performed within the transaction, not the number of subtransactions or partitions. Meanwhile, Epoxy and Ticket show different performance trends. For Epoxy, now with fixed partitions, the relative amount of application-level concurrency control remains similar across different warehouse numbers, leading to a stable normalized throughput ranging from 46.2% to 52.5%. Similarly, there are always two shared ticket counters for Ticket, imposing high contention regardless of the warehouse number, resulting in a stable but low normalized throughput ranging from 3.8% to 5.1%.

Throughput-Latency Curves. The throughput-latency graphs for P50 and P99 latency using two one-warehouse-per-partition configurations are shown in Figure 10. Sonata has latency similar to 2PC, and both are consistently higher than Local due to round trips for 2PC messages. For example, at two partitions, when reaching half of the peak throughput, Sonata's P50 latency is 6.4% higher than Local (11.6 ms vs. 10.9 ms), and its P99 latency is 4.0% higher (27.3 ms vs. 26.3 ms). Meanwhile, as Epoxy consolidates atomic commit with concurrency control, it avoids 2PC round trips, achieving 10.8 ms P50 latency at half of its peak throughput and two partitions. Yet, there is a significant drop in Epoxy's throughput after it has peaked. This is due to its NO_WAIT [61] deadlock prevention policy,

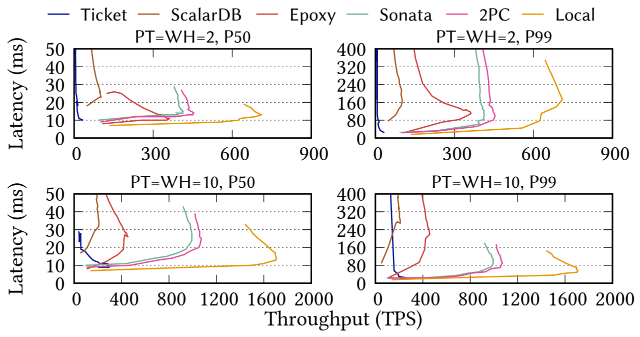

Figure 10: TPC-C throughput vs. P50 and P99 latency.

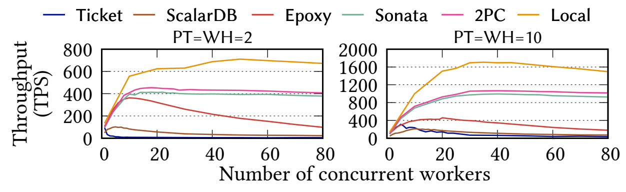

Figure 11: TPC-C throughput vs. concurrent level.

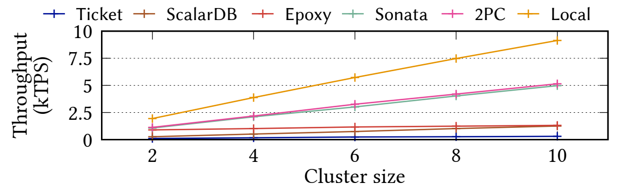

Figure 12: TPC-C peak throughput at different scales.

and with more concurrent transactions, transactions are more likely to encounter lock conflicts and be aborted. At ten partitions, the latency of Epoxy increases more quickly due to its increased amount of subtransactions at secondary databases. ScalarDB shows a much higher latency due to its costly application-level concurrency control: every record updated results in two disk writes during prepare and commit, and every record read is re-read during validation. Ticket's throughput at two partitions has dropped since the first data point (two concurrent clients) due to the limited number of shared ticket counters. At ten partitions, its P99 latency bumps up to 5 seconds (out of the y-axis range) due to distributed deadlocks caused by conflicts on the shared ticket counters.

Scalability. We first examine how the number of concurrent workers affects the throughput. As shown in Figure 11, all systems except for Ticket at PT=2 show performance improvement as more workers are added. Local, 2PC, and Sonata are able to sustain high throughput for a larger range of workers after peaking. Across all worker numbers, Sonata's coordination overhead is consistently small. Meanwhile, others' throughput drops quickly after peaking.

Next, we examine the performance when more database nodes are added. Each node is assgined a 10-warehouse partition, and coordinator instances are scaled out as well when possible to avoid bottlenecks. Figure 12 shows the results. Except for Epoxy, all show

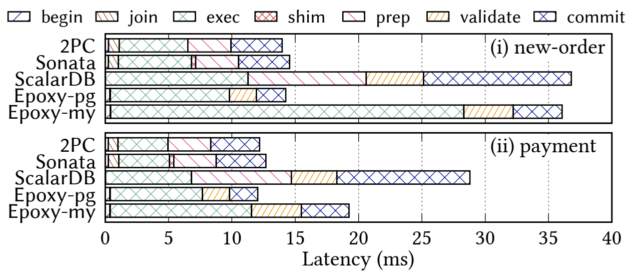

Figure 13: Latency breakdown of TPC-C's new-order and payment transactions. Sonata and 2PC have no validate phase; Epoxy has no prepare phase; only Sonata has a shim segment.

a near-linear increase in throughput as none has a global singleton bottleneck in their designs. Sonata peaks at 4977 TPS with 10 nodes, slightly lower than 2PC's 5153 TPS, and is 4.65× of Sonata's 2-node peak. Epoxy's 10-node peak, 1319 TPS, is only 1.46× of its 2-node peak, as its singleton primary PostgreSQL bottlenecks the cluster.

Latency Breakdown. We now break down and analyze the new-order and payment latency. For stable measurement, each new-order includes one and only one remote item, and each payment transaction always involves a remote customer. The chance of invalid new-order input, originally 1%, is now 0%. Figure 13 shows the results. Compared with 2PC, Sonata does not lengthen any preexisting phase and only adds a small shim layer overhead, constituting 2.2% and 2.6% of the total latency, respectively. ScalarDB's longer execution phase stems from the additional schema-checking queries introduced by its KV abstraction layer and the cost of maintaining application-level metadata like read and write sets. Its prepare phase is also longer as its buffered write set is sent to the database through multiple write statements, whose parsing, execution, and persistence all happen in this phase. ScalarDB validates by rereading its read set, introducing multiple database queries. When ScalarDB commits, it updates all written records again to set them to a committed state, which takes similar time as its prepare phase. Meanwhile, Epoxy's latency distribution heavily depends on the proportion of work done at the primary and secondary databases. Thus, two situations where the home warehouse that corresponds to most of work sits in the primary PostgreSQL (Epoxy-pg) and the secondary MySQL (Epoxy-my), respectively, are evaluated. Shifting more work to secondary databases increases the execution, prepare, and commit phases due to its application-level concurrency control, which is proportional to the amount of work: all writes are amplified during execution as described earlier, examined for write-write conflicts during validation, and require application-level locks that are held until commit; all reads are burdened with additional predicates as described earlier. Still, when the primary database handles most of the work, Epoxy-pg has similar latency as 2PC and Sonata due to the absence of 2PC round trips.

6.3 Microbenchmark Performance

Impact of Contention. We first vary the table size N from 100 to 1M and set other parameters as follows: S=2, R=W=6, and P=50%. Figure 14 shows the results. As the table size decreases, the likelihood of conflicts increases, and all systems except Ticket

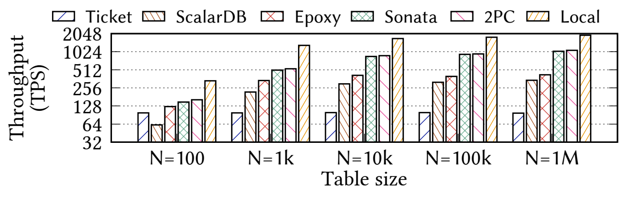

Figure 14: Peak throughput under different table sizes.

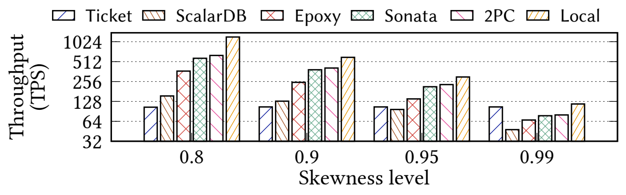

Figure 15: Peak throughput under skewed workloads.

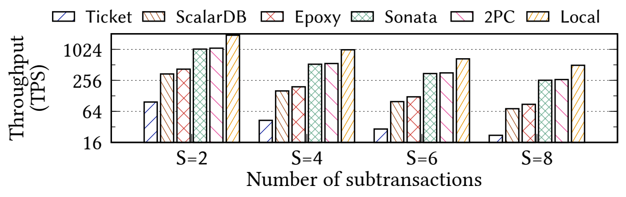

Figure 16: Peak throughput under different transaction sizes.

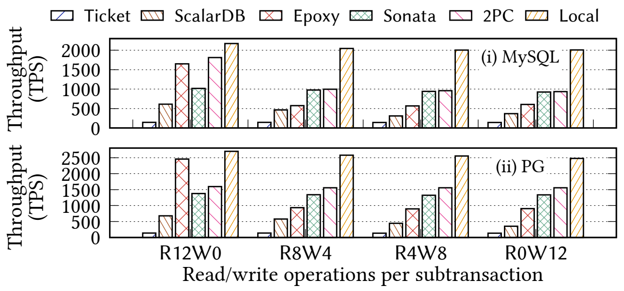

Figure 17: Peak throughput under different read-write ratios.

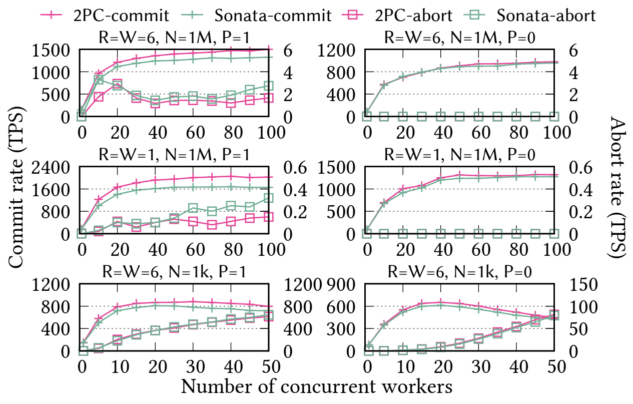

Figure 18: Commit/abort rates vs. concurrency level.

experience a decrease in throughput. When the table size N is 100, conflicts are the main factor for transaction aborts, and thus, Sonata only outperforms Epoxy by 19.2% and ScalarDB by 139.3%. As the table size increases, aborts due to conflicts decrease, and the performance difference becomes dominated by the coordination overhead. At N=1M, Sonata outperforms Epoxy by 145.5% and ScalarDB by 204.1%.

Next, we consider more contentious scenarios by making workloads write-only and introducing the skewness level parameter, $K \in [0.5,1)$, such that a fraction 1-K of all rows are accessed by a fraction K of all operations. We set N to 10k. Figure 15 shows the results. As K increases from 0.8 to 0.99, hot rows in a table drops from 2k to 100, significantly increasing conflicts. For example, with 10 concurrent workers, the abort ratio of 2PL/Sonata/ScalarDB/Epoxy grows from 7.4%/7.7%/26%/28% to 51%/52%/93%/75%. Still, across all configurations, Sonata performs similarly to 2PC and always outperforms Epoxy and ScalarDB, suggesting that Sonata is no more suspectable to skewness than other baselines.

Impact of Transaction Size. We evaluate the impact of transaction sizes by varying the number of subtransactions S from 2 to 8 and set other parameters as follows: R=W=6, N=1M, and P=50%. Figure 16 shows the results. As the number of subtransactions S increases, the throughput of all systems decreases. This is expected as the amount of work grows linearly with S. As with TPC-C, both Sonata and 2PC achieve a stable relative performance to Local across different S: Sonata achieves 51.2% to 53.9% of Local, while 2PC achieves 52.6% to 56.1%. These numbers are slightly smaller than those from TPC-C since less service computation is involved in the microbenchmarks, making the coordination overhead more prominent. Still, Sonata outperforms Epoxy by 145.5% (at S=2) to 194.0% (at S=8). ScalarDB performs slightly worse than Epoxy and is outperformed by Sonata by 204.1% to 256.8%.

Single-Database Performance. We use single-subtransaction (S=1) workloads to exercise either PostgreSQL (P=1) or MySQL (P=0). We begin with different read-write ratios and fix the operation count to 12. Table size N is set to 1M. Figure 17 shows the results. In general, the performance of all systems decreases as the workload becomes more write-heavy. Sonata performs similarly with 2PC in most cases with a up to 3% overhead in MySQL and 14% in PostgreSQL. 2PC's read-only MySQL performance noticeably surpass Sonata due to a prepare-time optimization in MySQL, which reduces disk writes by directly rolling back the subtransaction if the redo log is empty. The dummy writes from the Sonata shim disable this optimization, making Sonata 43.9% slower than 2PC. Sonata outperforms Epoxy except in read-only workloads by up to 70.1% in MySQL and up to 47.5% in PostgreSQL. In read-only workloads, Epoxy's snapshot caching optimization effectively eliminates most application-level coordination and reduces transactions to local transactions. Sonata outperforms ScalarDB by up to 203.1% in MySQL and up to 281.7% in PostgreSQL.

Next, we compare Sonata with 2PC in details to analyze the shim layer overhead in each database. Figure 18 shows the commit and abort rates under different numbers of concurrent workers. With N=1M and R=W=6, we observe a 4–11% commit rate reduction with PostgreSQL and up to 5% with MySQL. For abort rates, due to the large table size, both 2PC and Sonata have zero aborts in MySQL, and Sonata adds only 1-2 TPS in PostgreSQL. We then reduce the amount of work by setting R=W=1. The impact on the commit rate slightly increases as shim operations now take a larger proportion. For aborts, the relative increase in abort rates is larger since CO-violating transactions, i.e., those with rw-dependencies that prepare in a CO-violating order, are less likely to be aborted by the databases due to the lower contention. However, also due to the lower contention, the absolute abort rates for both 2PC and Sonata are much lower (less than 1 TPS), making the impact of increased abort rates negligible. We also reduce N to 1k for higher contention. While the absolute increase in abort rates is much larger (up to 33 TPS), Sonata's relative impact becomes much lower, as CO-violating transactions are more likely to participate in SSI dangerous structures or S2PL deadlocks and thus aborted by the underlying database, not the shim layer. In terms of latency, Sonata shims generally add 1 ms to the total latency and, as in our previous breakdown (Figure 13), do not affect the time taken by existing 2PC phases. We skip latency details for brevity.

7 RELATED WORK

Global Serializability. Global serializability for multi-database transactions has been studied since the early days [19, 66]. The ticket method and refinements [15, 36] assume local serializability from the underlying databases. They force subtransactions to explicitly conflict by reading and incrementing per-database shared ticket counters. The transaction-site graph algorithm by Breitbart and Silberschatz [20] and altruistic locking [9, 62] further require strict local serializability [43]. Mehrotra et al. [51] proposed a serialization point-based approach, requiring subtransactions at each database to be serialized in the order of their local serialization point events. The knowledge of such events is assumed to be provided by the underlying databases. Raz [57–59] proposed the theory of commit ordering (§2.3) and a family of scheduling algorithms that test the local serialization graph to maintain the commit ordering condition. Overall, these protocols are either too conservative or pose requirements on databases systems that are hardly met in practice. In contrast, Sonata exploits common properties that hold for existing popular database systems to achieve general applicability and practical performance.

Application-Level Concurrency Control. Recent work explores shifting more concurrency control responsibilities to the application layer to achieve global serializability or weaker isolation guarantees. Epoxy [46] and Cherry Garcia [28, 29] are two protocols that provide SI across multiple databases. Since they perform coordination at the application level, they only require the underlying databases to provide linearizable [43] durable KV operations. They keep multiple data versions in the underlying databases with additional protocol-specific metadata, which can make the underlying system inaccessible without proper application wrappers. ScalarDB [78] extends Cherry Garcia to provide global serializability with two strategies: a pessimistic one that turns every read into writes and an optimistic one that performs an additional validation after the prepare phase, and the latter is the default. Therefore, it incurs more overhead with similar limitations as Cherry Garcia. In contrast, by focusing on SSI and S2PL systems, Sonata's application-level prepare-time coordination is lightweight and does not require maintaining additional data versions or metadata.

Relaxed Transaction Semantics. In distributed service-oriented applications today, multi-database transaction models with more relaxed semantics are often used. The Saga pattern [34] is a popular approach that breaks a multi-database transaction into a sequence of local transactions associated with compensating local transactions. Atomicity is relaxed as the effect of local transactions can be partly observed by others. The try-confirm/cancel pattern (TCC)[53] moves the coordination responsibility to the business level and requires business logic to be two-phased. The first phase checks business conditions and reserves resources, and the second either confirms or cancels the reservation, depending on whether all participants have succeeded in the first phase. Therefore, any multi-database transaction observed to be partially confirmed will eventually be fully confirmed and never canceled. The XA specification [77] defines interfaces for different databases to participate in 2PC, coordinated by any compliant coordinator. XA transactions are atomic and durable but without cross-subtransaction isolation. It is adapted by Java Transaction API and .NET TransactionScope to compose multi-database transactions in respective languages.

Layered Transaction Management. Distributed databases often adopt a layered approach to implement transactions on top of self-contained storage systems. Unlike multi-database solutions that support heterogeneous database systems, these systems are designed specifically with one storage system type. Percolator [54] builds upon BigTable [23] to provide distributed transactions with SI. It uses a client-coordinated MVCC protocol that stores locks and other metadata in additional BigTable columns. Similar to Percolator, Omid [40, 64, 65] builds upon HBase to provide SI. Unlike Percolator, Omid uses a centralized transaction status oracle server for conflict detection and snapshot management. Megastore [14] is another system that builds upon BigTable. It provides a semi-relational data model missing in Percolator with a weaker isolation guarantee. Citus [27] allows distributed transactions across multiple PostgreSQL instances. It uses PostgreSQL's native 2PC interface for atomicity and durability and adds a distributed deadlock detector that uses periodical polling to build wait-for graphs.

8 CONCLUSION

We have presented Sonata, an ACID multi-database transaction system. Sonata works as application-level shim layers. Sonata leverages the common properties of popular database systems to provides global serializability without fully performing transaction concurrency control at the application level or requring changes to applications' schemas, query statements, database drivers, or database systems themselves. Sonata adds only 7.1% coordination overhead on average, outperforming prior solutions by up to 1114.3%.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China under Grant Nos. 62422209, 62132014, and 62272304. Zhaoguo Wang is the corresponding author.

REFERENCES

[1] [n.d.]. AgensGraph. https://bitnine.net/agensgraph/.

[2] [n.d.]. EDB. https://www.enterprisedb.com.

[3] [n.d.]. Fujitsu Enterprise Postgres. https://www.postgresql.fastware.com.

[4] [n.d.]. Neon. https://neon.tech.

[5] [n.d.]. Tmax OpenSQL. https://www.global.tibero.com/product.

[6] 2023. Stack Overflow 2023 Developer Survey. https://survey.stackoverflow.co/2023.

[7] 2024. Stack Overflow 2024 Developer Survey. https://survey.stackoverflow.co/2024.

[8] A. Adya, B. Liskov, and P. O'Neil. 2000. Generalized Isolation Level Definitions. In *Proceedings of the 16th International Conference on Data Engineering (ICDE '00)*. IEEE Computer Society, USA, 67.

[9] Rafael Alonso, Hector Garcia-Molina, and Kenneth Salem. 1987. Concurrency Control and Recovery for Global Procedures in Federated Database Systems. *IEEE Data Eng. Bull.* 10, 3 (1987), 5–11. http://sites.computer.org/debull/87SEP-CD.pdf

[10] Paul Andlinger and Matthias Gelbmann. 2018. PostgreSQL is the DBMS of the Year 2017. https://db-engines.com/en/blog_post/76.

[11] Paul Andlinger and Matthias Gelbmann. 2019. PostgreSQL is the DBMS of the Year 2018. https://db-engines.com/en/blog_post/79.

[12] Paul Andlinger and Matthias Gelbmann. 2021. PostgreSQL is the DBMS of the Year 2020. https://db-engines.com/en/blog_post/85.

[13] Apache. 2024. Apache Seata 2.1. https://seata.apache.org.

[14] Jason Baker, Chris Bond, James C. Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Fifth Biennial Conference on Innovative Data Systems Research, CIDR 2011, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*. www.cidrdb.org, 223–234. http://cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf

[15] R.K. Batra, M. Rusinkiewicz, and D. Georgakopoulos. 1992. A decentralized deadlock-free concurrency control method for multidatabase transactions. In *[1992] Proceedings of the 12th International Conference on Distributed Computing Systems*. 72–79. https://doi.org/10.1109/ICDCS.1992.235053

[16] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. *SIGMOD Rec.* 24, 2 (May 1995), 1–10. https://doi.org/10.1145/568271.223785

[17] Josh Berkus. 2009. Elephant Roads: a tour of Postgres forks. https://www.slideshare.net/slideshow/elephant-roads-a-tour-of-postgres-forks/5376286.

[18] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency control and recovery in database systems.* Addison-Wesley Longman Publishing Co., Inc., USA.

[19] Yuri Breitbart, Hector Garcia-Molina, and Abraham Silberschatz. 1992. Overview of Multidatabase Transaction Management. *VLDB J.* 1, 2 (1992), 181–239. http://www.vldb.org/journal/VLDBJ1/P181.pdf

[20] Yuri Breitbart and Avi Silberschatz. 1988. Multidatabase update issues. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '88)*. Association for Computing Machinery, New York, NY, USA, 135–142. https://doi.org/10.1145/50202.50217

[21] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2008. Serializable isolation for snapshot databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 729–738. https://doi.org/10.1145/1376616.1376690

[22] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2009. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.* 34, 4, Article 20 (dec 2009), 42 pages. https://doi.org/10.1145/1620585.1620587

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

[24] Youmin Chen, Xiangyao Yu, Paraschos Koutris, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. 2022. Plor: General Transactions with Predictable, Low Tail Latency. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 19–33. https://doi.org/10.1145/3514221.3517879

[25] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, USA, 251–264.

[26] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3, Article 8 (Aug. 2013), 22 pages. https://doi.org/10.1145/2491245

[27] Umur Cubukcu, Ozgun Erdogan, Sumedh Pathak, Sudhakar Sannakkayala, and Marco Slot. 2021. Citus: Distributed PostgreSQL for Data-Intensive Applications. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2490–2502. https://doi.org/10.1145/3448016.3457551

[28] Akon Dey, Alan Fekete, and Uwe Röhm. 2013. Scalable transactions across heterogeneous NoSQL key-value data stores. *Proc. VLDB Endow.* 6, 12 (Aug. 2013), 1434–1439. https://doi.org/10.14778/2536274.2536331

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

[30] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB* 7, 4 (2013), 277–288. http://www.vldb.org/pvldb/vol7/p277-difallah.pdf

[31] Bailu Ding, Lucja Kot, and Johannes Gehrke. 2018. Improving optimistic concurrency control through transaction batching and operation reordering. *Proc. VLDB Endow.* 12, 2 (Oct. 2018), 169–182. https://doi.org/10.14778/3282495.3282502

[32] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. 1976. The notions of consistency and predicate locks in a database system. *Commun. ACM* 19, 11 (Nov. 1976), 624–633. https://doi.org/10.1145/360363.360369

[33] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. 2005. Making snapshot isolation serializable. *ACM Trans. Database Syst.* 30, 2 (jun 2005), 492–528. https://doi.org/10.1145/1071610.1071615

[34] Hector Garcia-Molina and Kenneth Salem. 1987. Sagas. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '87)*. Association for Computing Machinery, New York, NY, USA, 249–259. https://doi.org/10.1145/38713.38742

[35] Matthias Gelbmann and Paul Andlinger. 2024. PostgreSQL is the DBMS of the Year 2023. https://db-engines.com/en/blog_post/106.

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

[37] Goetz Graefe, Mark Lillibridge, Harumi Kuno, Joseph Tucek, and Alistair Veitch. 2013. Controlled lock violation. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 85–96. https://doi.org/10.1145/2463676.2465325

[38] Jim Gray. 1978. Notes on Data Base Operating Systems. In *Operating Systems, An Advanced Course*. Springer-Verlag, Berlin, Heidelberg, 393–481.

[39] Jim Gray. 1981. The transaction concept: virtues and limitations (invited paper). In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7* (Cannes, France) *(VLDB '81)*. VLDB Endowment, 144–154.

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

[41] Theo Haerder and Andreas Reuter. 1983. Principles of transaction-oriented database recovery. *ACM Comput. Surv.* 15, 4 (Dec. 1983), 287–317. https://doi.org/10.1145/289.291

[42] Major Hayden. 2024. MySQLTuner 2.6.0. https://github.com/major/MySQLTuner-perl.

[43] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. https://doi.org/10.1145/78969.78972

[44] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2020. Opportunities for optimism in contended main-memory multicore transactions. *Proc. VLDB Endow.* 13, 5 (Jan. 2020), 629–642. https://doi.org/10.14778/3377369.3377373

[45] Darby Huye, Yuri Shkuro, and Raja R. Sambasivan. 2023. Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 419–432. https://www.usenix.org/conference/atc23/presentation/huye

[46] Peter Kraft, Qian Li, Xinjing Zhou, Peter Bailis, Michael Stonebraker, Matei Zaharia, and Xiangyao Yu. 2023. Epoxy: ACID Transactions across Diverse Data Stores. *Proc. VLDB Endow.* 16, 11 (July 2023), 2742–2754. https://doi.org/10.14778/3611479.3611484

[47] Rodrigo Laigner, Yongluan Zhou, Marcos Antonio Vaz Salles, Yijian Liu, and Marcos Kalinowski. 2021. Data management in microservices: state of the practice, challenges, and research directions. *Proc. VLDB Endow.* 14, 13 (Sept.

2021), 3348–3361. https://doi.org/10.14778/3484224.3484232

[48] Butler W. Lampson. 1979. *Crash recovery in a distributed data storage system.* Technical Report. Xerox Palo Alto Research Center.

[49] James Lewis and Martin Fowler. 2014. Microservices: a definition of this new architectural term. https://martinfowler.com/articles/microservices.html.

[50] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 412–426. https://doi.org/10.1145/3472883.3487003

[51] Sharad Mehrotra, Rajeev Rastogi, Yuri Breitbart, Henry F. Korth, and Avi Silberschatz. 1992. The concurrency control problem in multidatabases: characteristics and solutions. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data* (San Diego, California, USA) *(SIGMOD '92)*. Association for Computing Machinery, New York, NY, USA, 288–297. https://doi.org/10.1145/130283.130327

[52] Sam Newman. 2021. *Building Microservices, 2nd Edition.* O'Reilly Media, Inc.

[53] Guy Pardon. 2009. Try-Cancel/Confirm: Transactions for (Web) Services. https://web.archive.org/web/20090106020843/http://www.atomikos.com/Publications/TryCancelConfirm.

[54] Daniel Peng and Frank Dabek. 2010. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, USA, 251–264.

[55] Dan R. K. Ports and Kevin Grittner. 2012. Serializable snapshot isolation in PostgreSQL. *Proc. VLDB Endow.* 5, 12 (aug 2012), 1850–1861. https://doi.org/10.14778/2367502.2367523

[56] Guna Prasaad, Alvin Cheung, and Dan Suciu. 2020. Handling Highly Contended OLTP Workloads Using Fast Dynamic Partitioning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 527–542. https://doi.org/10.1145/3318464.3389764

[57] Yoav Raz. 1992. The Principle of Commitment Ordering, or Guaranteeing Serializability in a Heterogeneous Environment of Multiple Autonomous Resource Mangers Using Atomic Commitment. In *Proceedings of the 18th International Conference on Very Large Data Bases (VLDB '92)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 292–312.

[58] Y. Raz. 1993. Commitment ordering based distributed concurrency control for bridging single and multi version resources. In *Proceedings RIDE-IMS '93: Third International Workshop on Research Issues in Data Engineering: Interoperability in Multidatabase Systems*. 189–198. https://doi.org/10.1109/RIDE.1993.281924

[59] Yoav Raz. 1993. Extended commitment ordering, or guaranteeing global serializability by applying commitment order selectively to global transactions. In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Washington, D.C., USA) *(PODS '93)*. Association for Computing Machinery, New York, NY, USA, 83–96. https://doi.org/10.1145/153850.153858

[60] Stephen Revilak, Patrick O'Neil, and Elizabeth O'Neil. 2011. Precisely Serializable Snapshot Isolation (PSSI). In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE '11)*. IEEE Computer Society, USA, 482–493. https://doi.org/10.1109/ICDE.2011.5767853

[61] Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis. 1978. System level concurrency control for distributed database systems. *ACM Trans. Database Syst.* 3, 2 (June 1978), 178–198. https://doi.org/10.1145/320251.320260

[62] Kenneth Salem, Hector Garcia-Molina, and Rafael Alonso. 1989. Altruistic locking: A strategy for coping with long lived transactions. In *High Performance Transaction Systems*, Dieter Gawlick, Mark Haynie, and Andreas Reuter (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 175–199.

[63] Korakit Seemakhupt, Brent E. Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C. Snoeren, Arvind Krishnamurthy, David E. Culler, and Henry M. Levy. 2023. A Cloud-Scale Characterization of Remote Procedure Calls. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 498–514. https://doi.org/10.1145/3600006.3613156

[64] Ohad Shacham, Yonatan Gottesman, Aran Bergman, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. 2018. Taking omid to the clouds: fast, scalable transactions for real-time cloud analytics. *Proc. VLDB Endow.* 11, 12 (Aug. 2018), 1795–1808. https://doi.org/10.14778/3229863.3229868

[65] Ohad Shacham, Francisco Perez-Sorrosal, Edward Bortnikov, Eshcar Hillel, Idit Keidar, Ivan Kelly, Matthieu Morel, and Sameer Paranjpye. 2017. Omid, reloaded: scalable and highly-available transaction processing. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies* (Santa clara, CA, USA) *(FAST'17)*. USENIX Association, USA, 167–180.

[66] Amit P. Sheth and James A. Larson. 1990. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comput. Surv.* 22, 3 (Sept. 1990), 183–236. https://doi.org/10.1145/96602.96604

[67] Michael Stonebraker. 2010. Why Enterprises Are Uninterested in NoSQL. BLOG-CACM.

[68] Michael Stonebraker and Andrew Pavlo. 2024. What Goes Around Comes Around... And Around... *SIGMOD Rec.* 53, 2 (July 2024), 21–37. https://doi.org/10.1145/3685980.3685984

[69] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1493–1509. https://doi.org/10.1145/3318464.3386134

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

[71] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 18–32. https://doi.org/10.1145/2517349.2522713

[72] Oleksii Vasyliev. 2024. PgTune. https://pgtune.leopard.in.ua.

[73] Jiachen Wang, Ding Ding, Huan Wang, Conrad Christensen, Zhaoguo Wang, Haibo Chen, and Jinyang Li. 2021. Polyjuice: High-Performance Transactions via Learned Concurrency Control. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 198–216. https://www.usenix.org/conference/osdi21/presentation/wang-jiachen

[74] Tianzheng Wang, Ryan Johnson, Alan Fekete, and Ippokratis Pandis. 2017. Efficiently making (almost) any concurrency control mechanism serializable. *The VLDB Journal* 26, 4 (Aug. 2017), 537–562. https://doi.org/10.1007/s00778-017-0463-8

[75] Zhaoguo Wang, Shuai Mu, Yang Cui, Han Yi, Haibo Chen, and Jinyang Li. 2016. Scaling Multicore Databases via Constrained Parallel Execution. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1643–1658. https://doi.org/10.1145/2882903.2882934

[76] Zhaoguo Wang, Chuzhe Tang, Xiaodong Zhang, Qianmian Yu, Binyu Zang, Haibing Guan, and Haibo Chen. 2024. Ad Hoc Transactions through the Looking Glass: An Empirical Study of Application-Level Transactions in Web Applications. *ACM Trans. Database Syst.* 49, 1, Article 3 (Feb. 2024), 43 pages. https://doi.org/10.1145/3638553

[77] X/Open Company Limited. 1991. Distributed Transaction Processing: The XA Specification.

[78] Hiroyuki Yamada, Toshihiro Suzuki, Yuji Ito, and Jun Nemoto. 2023. ScalarDB: Universal Transaction Manager for Polystores. *Proc. VLDB Endow.* 16, 12 (Aug. 2023), 3768–3780. https://doi.org/10.14778/3611540.3611563