TxnSails: Achieving Serializable Transaction Scheduling with Self-Adaptive Isolation Level Selection

Paper No: 1446

ABSTRACT

Serializable isolation level is regarded as the gold standard for transaction processing; however, it incurs significant overhead in guaranteeing serializable transaction scheduling. Recent studies reveal that scheduling entire transaction workloads under low isolation levels can still achieve serializability by adjusting specific query patterns, such as promoting reads to writes in certain SQL queries. However, selecting the optimal isolation level for such adaptations remains a complex and challenging task. Moreover, these modifications are often outperformed by simply configuring the database to serializable isolation. In this paper, we present TxnSails, a middle-tier solution designed to ensure serializability by strategically selecting between serializable and low isolation levels for dynamic workloads. First, TxnSails incorporates a unified concurrency control algorithm that achieves serializability at lower isolation levels without introducing additional write conflicts. Second, TxnSails employs a graph-based model that characterizes dynamic workloads and uses deep learning techniques to predict the optimal isolation level. Finally, TxnSails implements a cross-isolation validation mechanism to ensure serializability during real-time transitions between different isolation levels. Extensive experiments show that TxnSails outperforms state-of-the-art solutions by up to 26.7x and PostgreSQL's serializable isolation by up to 4.8x.

ACM Reference Format:

Paper No: 1446. 2024. TxnSails: Achieving Serializable Transaction Scheduling with Self-Adaptive Isolation Level Selection. In *Proceedings of the 2025 International Conference on Management of Data (SIGMOD '25).* ACM, New York, NY, USA, 14 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Serializable isolation level (SER) is regarded as the gold standard for transaction processing due to its ability to prevent all forms of anomalies. SER is essential in mission-critical applications, such as banking systems in finance and air traffic control systems in transportation, which require their data to be 100% correct. However, it incurs expensive coordination overhead by configuring the RDBMS to SER. Despite significant efforts to alleviate this overhead [34, 41, 48], maintaining a serial order of transactions to be scheduled still remains a fundamental performance bottleneck.

SIGMOD '25, June 22-27, 2025, Berlin, Germany

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

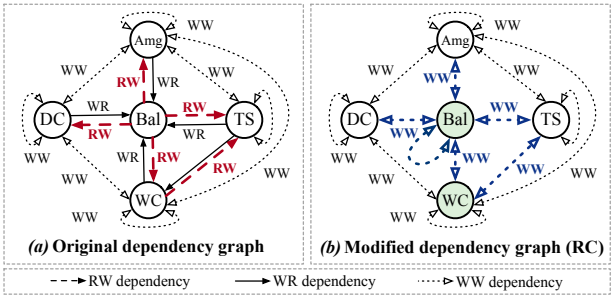


Figure 1: Static dependency graphs of Smallbank

In recent years, many studies have explored achieving SER by modifying applications while configuring the RDBMS to a low isolation level. This approach is driven by two key reasons. First, some RDBMSs, such as Oracle 11g, do not natively support SER, requiring application logic modifications to enforce it. A more comprehensive list of RDBMSs and their supported isolation levels can be found in [10]. Second, RDBMSs typically offer better performance at lower isolation levels, such as read committed (RC) and snapshot isolation (SI), due to their more relaxed ordering requirements. Modifying applications to achieve SER while using a lower isolation level may result in better performance compared to directly setting the RDBMS to SER [8, 9, 42].

The main idea of existing works that achieve SER by modifying application logic follows three steps: ❶ Configure the RDBMS to a low isolation level and build a *static dependency graph* by analyzing the transaction templates of the workload. Since users typically submit transactions through forms or programs, these transactions can be abstracted into templates. They model the transaction templates as a static dependency graph, where each template is represented by a vertex. The dependencies between templates, which can be write-write (WW), write-read (WR), or read-write (RW), are represented as edges. ❷ Analyze the graph to identify all *dangerous structures* that are permissible under the low isolation level but prohibited by SER. Dangerous structures under different isolation levels are defined separately. For example, under SI, a dangerous structure is characterized by two consecutive RW dependencies [23, 24], while under RC, a single RW dependency constitutes the dangerous structure [9, 42]. ❸ Eliminate dangerous structures by modifying application logic, e.g., promoting reads to writes for certain SQL queries so that the RW dependencies are eliminated, and thus ensures SER under low isolation levels.

Example 1. *Consider the SmallBank benchmark [8], which consists of five transaction templates: amalgamate (Amg), balance (Bal), depositChecking (DC), transactSavings (TS), and writeCheck (WC). Suppose the RDBMS is configured to RC. As outlined in step ❶, the benchmark is modeled into a static dependency graph (shown in Figure 1(a)). The graph contains 16 WW dependencies, 5 WR dependencies, and 5 RW dependencies. At step ❷, five dangerous structures*

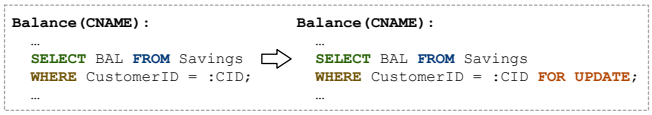


Figure 2: Modification of Bal transaction template

(highlighted by red dashed arrows) are identified. These include the dependency from WC to TS and 4 dependencies from Bal to the other four templates. At step ❸, extra writes are introduced to eliminate the dangerous structures by converting RW dependencies into WW dependencies. To achieve this, certain SELECT statements are modified to SELECT … FOR UPDATE statements. Figure 2 illustrates the modification of Bal. For reference, the complete modified dependency graph, based on the original in Figure 1(a), is shown in Figure 1(b). □

Nevertheless, simply converting reads to writes can significantly degrade system performance for two reasons. First, when the workload of modified transactions becomes intensive, promoting reads to writes can block the execution of a large number of transactions. This is because RDBMSs often adopt MVCC, where read and write operations on the same data item do not cause conflicts, but two write operations on the same data item do. Second, determining the optimal isolation level for dynamic workloads is non-trivial. Consider Example 1. When the workload mainly consists of DC, Amg, and TS transactions, configuring the RDBMS to RC is preferable as it allows greater concurrency with low overhead for dangerous structure prevention. However, if Bal and WC transactions become intensive, the additional writes may lead to significant overhead, making SER a more efficient option. Thus, constantly configuring the RDBMS to a low isolation level does not guarantee optimal performance.

In this paper, we present TxnSails, a middle-tier solution designed to meet two key requirements: ❶ It must be efficient to handle dangerous structures under various lower isolation levels while ensuring SER. ❷ It must adaptively select the optimal isolation level to maximize performance in response to dynamic workloads. The first requirement reduces the overhead caused by RW-to-WW conversions, while the second addresses the challenge of selecting the optimal isolation level. However, implementing TxnSails poses three key challenges. First, developing a new approach without introducing additional writes that can promote various isolation levels to SER while minimizing promotion overhead is not trivial. Second, even with such a method, determining the optimal isolation level for dynamic workloads remains a complex task. Third, as workloads fluctuate, the system's optimal isolation level may need to change. Designing an efficient and correct transition between isolation levels is crucial. To address these challenges, we propose the following key techniques.

(1) Middle-tier concurrency control algorithm ensuring SER for each low isolation level (Section 4.1). We propose a dynamic, fine-grained approach that operates on individual transactions rather than transaction templates, ensuring that the execution of transactions meets the requirements of SER. This approach is inspired by the theorem that a schedule is serializable if it does not contain two transactions, T_i and T_j, where T_j commits before T_i, but there is an RW dependency from T_i to T_j [9]. Building on this theorem, we propose a unified concurrency control algorithm to ensure SER. The algorithm tracks the transactions with their templates involved in RW dependency within a static dependency graph. Upon committing any transaction, it checks if the commit order aligns with the dependency order. If they align, the transaction is committed; otherwise, it is aborted or blocked. Additionally, several optimizations are proposed to reduce the overhead of monitoring dependencies between transactions.

(2) Self-adaptive isolation level selection mechanism (Section 4.2). We observe that the optimal isolation level for achieving peak performance in different workloads is closely tied to two key characteristics: the data access dependencies between transactions and the data access distribution within transactions. Based on this insight, we propose a graph embedding prediction model for dynamic workloads, which predicts the optimal isolation level using real-time workload features. Specifically, we use a graph model to represent workload features, where vertices represent transaction individual features and edges represent data access dependencies between transactions. Using this graph model as inputs, TxnSails employs a learned model equipped with graph neural networks [14] and message-passing [27] techniques to predict the optimal isolation level for the workload. To the best of our knowledge, *TxnSails is the first work to enable self-adaptive isolation level selection for dynamic workloads.*

(3) Cross-isolation validation mechanism that enables efficient transitions and serializable scheduling (Section 4.3). The optimal isolation level should adapt as the workload evolves. When the RDBMS decides to change the isolation level, new transactions must be executed under this updated isolation. Although existing approaches can achieve SER when all transactions use a unified low isolation level, they fail to ensure SER when transactions operate under different isolation levels. This is because varying isolation levels can introduce new dangerous structures that may violate the requirements of SER. To address this issue, we identify dangerous structures across different isolation levels and propose a cross-isolation validation mechanism that can prevent these structures during transitions without causing significant system downtime. We prove the correctness of the cross-isolation validation mechanism in Section 5.2.

We have conducted extensive evaluations on SmallBank [8], TPC-C [4], and YCSB+T [17] benchmarks. The results show that TxnSails can adaptively select the optimal isolation level for dynamic workloads, achieving up to a 26.7x performance improvement over other state-of-the-art methods and up to a 4.8x performance boost compared to SER provided by PostgreSQL.

2 PRELIMINARIES

RDBMSs typically offer several isolation levels; in this paper, we focus on the three most commonly used: serializable (SER), snapshot isolation (SI), and read committed (RC). In this section, we first discuss transactions and transaction templates. We then present the dangerous structures under SI and RC, respectively. We finally discuss a new dependency that builds the foundation of our approach.

2.1 Transactions and Transaction Templates

A transaction is a sequence of read/write operations on data items that execute atomically. This means that either all operations are

successful, resulting in a commit of the transaction, or none of the operations succeed, leading to an abort.

A transaction template is a block of business logic code within the application that consists of predefined SQL statements with parameter placeholders. Take the Amalgamate (Amg) template as an example, which transfers funds from one customer to another. This template first reads the balances of the checking and savings accounts of customer N_1, sets both balances to zero, and subsequently credits the checking balance for N_2 by the sum of N_1's prior balances. In this context, N_1 and N_2 serve as parameter placeholders. This modular structure ensures flexibility, allowing the transaction template to be reused across various contexts while ensuring data consistency. When a customer initiates a transaction at runtime, the application fills the placeholders with actual data, and the complete transaction is executed in the RDBMS, finalizing the business logic.

For better clarity, we use $\mathcal{T}_i$ to denote a transaction template and T_i to denote a transaction generated by $\mathcal{T}_i$.

2.2 Template Dangerous Structures

The dependencies between two transaction templates, $\mathcal{T}_i$ and $\mathcal{T}_j$, are defined as follows: (1) $\mathcal{T}_i \xrightarrow{ww} \mathcal{T}_j$ if $\mathcal{T}_i$ and $\mathcal{T}_j$ write the same data set (e.g., relation) in sequence; (2) $\mathcal{T}_i \xrightarrow{wr} \mathcal{T}_j$ if $\mathcal{T}_i$ writes and $\mathcal{T}_j$ reads the same data set in sequence; (3) $\mathcal{T}_i \xrightarrow{rw} \mathcal{T}_j$ if $\mathcal{T}_i$ reads and $\mathcal{T}_j$ writes the same data set in sequence.

Definition 1 (Static SI dangerous structure [15]). *In a static dependency graph, two consecutive edges $\mathcal{T}_i \xrightarrow{rw} \mathcal{T}_j, \mathcal{T}_j \xrightarrow{rw} \mathcal{T}_k$ are deemed to constitute a static SI dangerous structure.* □

Definition 2 (Static RC dangerous structure [9, 43]). *In a static dependency graph, an edge $\mathcal{T}_i \xrightarrow{rw} \mathcal{T}_j$ is deemed to constitute a static RC dangerous structure.* □

Theorem 1. *[6] If a static dependency graph contains no SI (resp. RC) static dangerous structures, then scheduling the transactions generated by the corresponding transaction templates achieves SER when the RDBMS is configured to SI (resp. RC).* □

Theorem 1 serves as the foundation for existing approaches to achieving SER while the RDBMS is configured to SI/RC. However, these approaches are static and coarse-grained, leading to the incorrect identification of many non-cyclic schedules. This, in turn, causes a significant number of unnecessary transaction rollbacks.

2.3 Transaction Dangerous Structures

The dependencies between two concurrent transactions, T_i and T_j, operating on the same item x, are classified as follows.

- $T_i \xrightarrow{ww} T_j$: T_i writes a version of data item x, and T_j writes a later version of x.
- $T_i \xrightarrow{wr} T_j$: T_i writes a version of data item x, and T_j reads either the version written by T_i or a later version of x.
- $T_i \xrightarrow{rw} T_j$: T_i reads a version of data item x, and T_j writes a later version of x.

Definition 3 (SI dangerous structure [37]). *Under SI, two consecutive RW dependencies: $T_i \xrightarrow{rw} T_j \xrightarrow{rw} T_k$ are considered as an SI dangerous structure, where T_i and T_j, T_j and T_k are concurrent transactions, respectively.* □

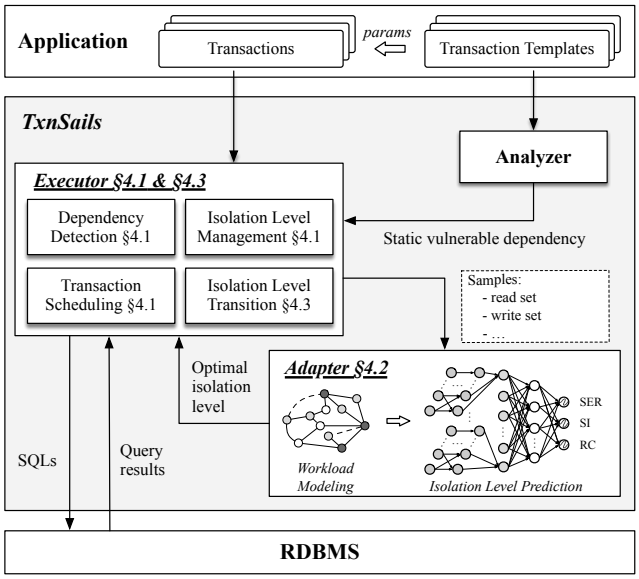


Figure 3: An overview of TxnSails

Definition 4 (RC dangerous structure [9, 26]). *Under RC, an RW dependency: $T_i \xrightarrow{rw} T_j$ is considered as an RC dangerous structure, where T_i and T_j are concurrent transactions.* □

2.4 Vulnerable Dependency

Definition 5 (Static vulnerable dependency). *The static vulnerable dependency is defined as $\mathcal{T}_j \xrightarrow{rw} \mathcal{T}_k$ in chain $\mathcal{T}_i \xrightarrow{rw} \mathcal{T}_j \xrightarrow{rw} \mathcal{T}_k$ under SI, and $\mathcal{T}_i \xrightarrow{rw} \mathcal{T}_j$ under RC, respectively.* □

Definition 6 (Vulnerable dependency). *The vulnerable dependency is defined as $T_j \xrightarrow{rw} T_k$ in chain $T_i \xrightarrow{rw} T_j \xrightarrow{rw} T_k$ under SI, and $T_i \xrightarrow{rw} T_j$ under RC, respectively.* □

Theorem 2 ([9]). *For any vulnerable dependency $T_i \xrightarrow{rw} T_j$, if T_i commits before T_j, then the transaction scheduling achieves SER.* □

Theorem 2 forms the basis of our dynamic, fine-grained approach to achieving SER. Compared to existing approaches, our approach neither introduces unnecessary writes nor misjudges cyclic schedules, thus preventing unwarranted transaction rollbacks.

3 OVERVIEW OF TxnSails

TxnSails works in the middle tier between the application tier and the database tier, designed to ❶ ensure SER when transactions operate under a low isolation level without introducing additional writes; ❷ select the optimal isolation level for dynamic workloads adaptively; ❸ constantly keep SER during the isolation level transition. An overview of TxnSails is depicted in Figure 3. It comprises three main components: *Analyzer*, *Executor*, and *Adapter*.

Analyzer. Before TxnSails starts to receive any transaction from the application, *Analyzer* builds the static dependency graph for the transaction templates and identifies all the static vulnerable dependencies for each low isolation level according to Definition 5. It then sends the static vulnerable dependencies to *Executor*.

Executor. *Executor* is responsible for transaction execution, ensuring SER when transactions operate under a single low isolation level and during the isolation level transition. There are four core modules: *Isolation Level Management(ILM)*, *Dependency Detection(DD)*, *Transaction Scheduling(TS)*, and *Isolation Level Transition(ILT)*. (1) ILM stores the static vulnerable dependencies and before any transaction T starts, it identifies whether T involves any static vulnerable dependencies. If the template of T does not involve static vulnerable dependencies, *Executor* sends T directly to the RDBMS for execution; otherwise, ILM triggers DD that identifies vulnerable dependencies of T. (2) DD monitors the reads and writes of transaction T, detecting any vulnerable dependencies between T and other transactions at runtime. If T is involved in any vulnerable dependencies, TS is triggered. (3) TS attempts to ensure that the commit order and the order of vulnerable dependencies remain consistent between T and other transactions. If this consistency cannot be guaranteed, T is blocked or aborted; otherwise, T proceeds to commit. (4) ILT is responsible for ensuring SER during the transition between two isolation levels. It follows a new corollary, which extends Theorem 2 to any two transactions, T_i and T_j, executing under different isolation levels. If $T_i \xrightarrow{rw} T_j$ and T_i commits before T_j, the transaction scheduling during the transition achieves SER. The proof ensuring SER during the isolation level transition is provided in detail in Section 5.2.

Adapter. *Adapter* is designed to predict the optimal isolation level and achieve the best performance when the workloads evolve. Initially, a dedicated thread is introduced to continuously sample aborted/committed transactions using Monte Carlo sampling [52], capturing transaction statuses and read/write data items. After collecting a batch of transaction samples, *Adapter* predicts the optimal isolation level for future workloads based on the characteristics of the batch. The prediction process consists of two parts: *Workload Modeling (WM)* and *Isolation Level Prediction (ILP)*. WM extracts performance-related features and models the workload as a graph, where the vertices represent the runtime transactions, and the edges are the operation dependencies between two transactions. The feature of vertices represents the individual features of transactions, including the number of data items in the read and write set. The feature of edges represents the operation dependencies, including the RW and WW dependencies. ILP first embeds the workload graph into a high-dimension vector using graph neural network [14] and message passing techniques [27], and then translates the high-dimension vector into three possible labels: RC, SI, or SER. The label with the highest value, as determined by our model, indicates the optimal isolation level.

4 DESIGN OF TxnSails

In this section, we provide the detailed design of TxnSails. We first introduce the middle-tier concurrency control algorithm that guarantees serializability when the underlying RDBMS is configured to a low isolation level (Section 4.1). Then, we present a self-adaptive isolation level selection approach (Section 4.2), which dynamically predicts the optimal future isolation level based on transaction dependency information. Lastly, we introduce the cross-isolation validation mechanism that ensures serializability during the self-adaptive isolation level switching (Section 4.3).

4.1 Middle-tier Concurrency Control

After receiving the transaction templates, TxnSails initially analyzes to identify all static vulnerable dependencies at each low isolation level and the corresponding transaction templates involved based on Definition 5. To prevent the non-serializable scheduling, TxnSails proposes the middle-tier concurrency control algorithm, which introduces a validation phase into the lifecycle of transactions derived from these templates. In the validation phase, TxnSails detects vulnerable dependencies between transactions derived from identified templates and schedules the commit order to make it consistent with the dependency order.

4.1.1 Transaction lifecycle. A transaction life is divided into three phases, i.e., the execution phase, the validation phase, and the commit phase. (1) In the execution phase, TxnSails establishes a database connection with a specific isolation level, which is not adjusted until the transaction is committed or aborted. Then, after the normal transaction execution, TxnSails stores the read/write data items in the thread-local buffer that may induce the vulnerable dependencies according to Definition 5; (2) In the validation phase, TxnSails acquires validation locks for data items stored in the buffer. Then, it detects the dependencies between them and intends to schedule the commit order consistent with the dependency order. A more detailed description of the validation phase will be given in Section 4.1.2; (3) In the commit phase, TxnSails applies the modification into the database and releases the validation locks obtained in the validation phase.

4.1.2 Validation. TxnSails performs two key tasks in the validation phase: (1) detecting vulnerable dependencies; (2) scheduling the commit order consistent with dependency order. To achieve this, we explicitly add a *version* column to the schema, which is incremented after every update. We trace the operation order and detect the dependencies by comparing the version of read/write data items. Algorithm 1 shows the detailed algorithm.

Detecting Vulnerable Dependencies. We detect vulnerable dependencies based on those defined in Definition 6 and Theorem 2. For both the RC and SI isolation levels, we should detect the vulnerable dependency $T_i \xrightarrow{rw} T_j$. We achieve this detection by introducing a Validation Lock Table (VLT). Before entering the validation, the transaction first requests *Shared* locks for items in the read set and *Exclusive* locks for items in the write set (lines 2-7). There are two steps to validate each transaction T_i. (1) The first step is to check T_i's read set, TxnSails detects whether its read set is modified by a committed transaction T_j, which results in an RW dependency $T_i \xrightarrow{rw} T_j$. (2) The second step is to check transaction T_i's write set, TxnSails detects whether older versions of its write set have been read by some other transaction T_j, in which case an RW dependency $T_j \xrightarrow{rw} T_i$ occurs.

During a transaction T_i's execution phase, it stores the version of the corresponding read/write data item in its thread-local buffer. When detecting vulnerable dependencies, T_i first traverses all read items and compares the version of each read item in the thread-local buffer with the item's latest version in the RDBMS (line 9-15 in Algorithm 1). If a version mismatch is found, indicating an RW dependency from the current transaction T_i to a committed

Algorithm 1: Middle-tier concurrency control algorithm

```
1  Function Validate(T, conn):
       Input: T, transaction requiring validation;
              conn, a connection under RC
       // Acquire the validation locks on data items
2      for r in T.vread_set ∪ T.vwrite_set do
3          res := TryValidationLock(r.key, T.tid, r.type)
4          while res is WAIT do
5              res := TryValidationLock(r.key, T.tid, r.type)
6          if res is ERROR then
7              return ERROR
       // Check the version of data items in the read set
8      for r in T.vread_set do
9          version := 0
10         entry := HVC.get_lock_entry(r.key)
11         if entry.version > 0 then
               // get the latest version from version cache
12             version := entry.version
13         else
               // fetch the lastest version from DBMS
14             version := conn.get_version(r.key)
15             entry.version := version
16         if version is not r.version then
17             return ERROR
18     return SUCCESS
19 Function Commit(T, sess):
       Input: sess, session for transaction execution;
20     sess.commit(T)
21     for r in T.vwrite_set do
22         entry := HVC.get_lock_entry(r.key)
23         entry.version = r.version
24     for r in T.vread_set ∪ T.vwrite_set do
25         ReleaseValidateLock(r.key)
```

transaction, say T_j, then T_i is aborted to ensure the consistency of commit and dependency orders (line 16-17).

After traversing the read set, T_i checks each data item in its write set to determine if there is any concurrent transaction T_j reading the same data item and is undergoing validation. We achieve this with the validation locks. If any validation lock request fails, indicating such a transaction T_j exists, an RW dependency is detected. In this condition, the failed lock request should be appended in the corresponding item's *WaitList*, making T_i wait until T_j commits, ensuring consistency between dependency and commit orders. If no concurrent transactions in the validation phase are reading the same item, T_i proceeds to commit and create a new data version.

Scheduling the Commit Order. We conduct a two-step assurance process to ensure that the commit order of transactions aligns with their dependency order. Firstly, we ensure the commit order in the middle tier is consistent with the dependency order. Secondly, we ensure the actual commit order in the RDBMS is consistent with the commit order in the middle tier. The middle tier consistency is achieved by aborting or blocking transactions once a vulnerable dependency is detected. We ensure the RDBMS

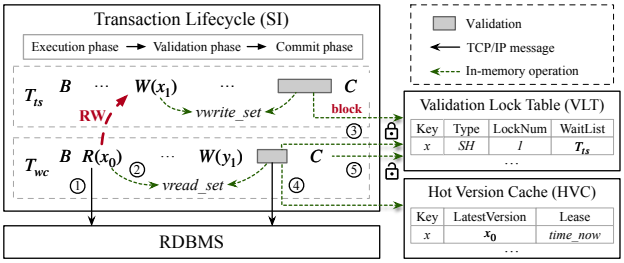


Figure 4: Transaction processing in TxnSails

layer consistency is achieved by releasing validation locks after the transaction is committed in the RDBMS (lines 24-25). Based on this, if two concurrent transactions access the same data item x (with one writing and the other reading or writing), they cannot both enter the validation phase simultaneously. One must complete validation and commit before the other can proceed to validation, ensuring the correct commit order in RDBMS.

Optimization. In the validation phase, TxnSails should compare the data items in the local buffer with the latest one, which introduces additional overhead due to interactions between the middle tier and the RDBMSs. To minimize the overhead, TxnSails is equipped with a hot version cache (HVC) that stores the latest version of frequently accessed data items. Each cache entry k in the HVC has two attributes, i.e., *lastestVersion* and *lease*. The *k.lastestVersion* attribute, initially set to -1, facilitates a fast path to obtain the latest version during validation. To manage memory efficiently, HVC only keeps versions for hot data items. The *k.lease* attribute denotes the expiration time of the cache entry, updated upon access, allowing TxnSails to release lock entries from the VLT when their leases expire.

Example 2. *Take Figure 4, which provides a concise depiction of transaction processing, as an example. Recall that there exists a static vulnerable dependency* $\mathcal{T}_{wc} \xrightarrow{rw} \mathcal{T}_{ts}$ *in Smallbank when the RDBMS is set to SI (Figure 1). Thus, it is necessary to detect the read operation of* T_{wc} *and the write operation of* T_{ts}*. In the execution phase, after the normal transaction execution (①),* T_{wc} *stores the data item x in its vread_set and* T_{ts} *stores x in its vwrite_set (②). In the validation phase of* T_{wc}*, it acquires the shared validation lock on x in the VLT (③) and retrieves the latest version of x from either HVC or the RDBMS (④). While in the validation phase of* T_{ts}*, it requests the exclusive validation lock on x and is blocked until* T_{wc} *releases the lock. Finally, in the commit phase,* T_{wc} *releases the validation lock on x. This ensures that the commit order of the two transactions is consistent with the dependency order, thereby guaranteeing SER when they operate under SI.* □

Discussion. We note that range queries with predicates may potentially introduce phantom reads and violate SER. For phantom reads, the definition of vulnerable dependency remains applicable, which enables TxnSails to detect and prevent this anomaly. The only difference is that we need to implement a larger granule validation lock, such as interval or table locks, to enable detecting the dependencies between predicates. As various coarse-grained locking techniques, such as SIREAD locks in PostgreSQL and gap locks [33], already exist, we opt to implement the coarse-grained validation locks using these methods and exclude locking optimization from our paper to focus on efficient isolation level adaptation.

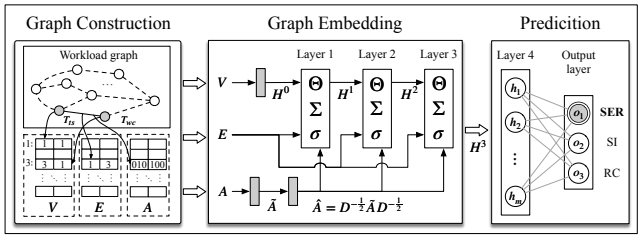


Figure 5: Graph-based isolation level selection model

4.2 Self-adaptive Isolation Level Selection

Selecting optimal isolation levels for all transactions in a workload while maintaining SER is challenging as we need to balance the overhead and performance gain in different isolation levels. Inspired by existing approaches that conduct workload prediction [35, 46, 50] for preemptive database tuning or index creation, which predicts future workloads based on the current workload features, we propose a neural-network-based isolation level prediction approach, which predicts the future optimal isolation level based on the current workload features. The main challenges are effective workload feature selection and representation. Towards this end, TxnSails adopts transaction dependency graphs to capture the workload features and adopts a graph neural network model to facilitate self-adaptive isolation level prediction.

4.2.1 Graph construction. One of the main challenges in workload modeling is the complexity of characterizing concurrent transactions, which can involve diverse transaction features and varying read/write sets. TxnSails utilizes three matrices, i.e., a vertice matrix, an edge index matrix, and an edge attribute matrix, as features to characterize concurrent transactions.

Formally, the vertex matrix, $V \in \mathbb{N}^{N\times 2}$, captures the features of N transactions. Without losing generality, we use the number of data items in the read set r_cnt and write set w_cnt as the transaction feature, which can be represented as a two-dimensional vector $\langle r\_cnt, w\_cnt\rangle$.

The edge matrix, $E \in \mathbb{N}^{2\times M}$, represents the dependencies between transactions. Different transactions may have such dependencies if they access the same data items. In TxnSails, we characterize workload features by using the intersection of read/write sets to define these dependencies. The edge matrix records the indices of the source and destination transactions of the dependency.

The attribute matrix, $A \in \mathbb{N}^{M\times 2}$, encapsulates the dependency type and the involved relations of M edges. The dependency type can be either RR, RW/WR, or WW. For example, if two transactions' read sets intersect, there is an RR dependency between them. Given fixed relations and dependency types, we use one-hot encoding to represent these dependencies d_type and the involved relations d_rel. Thus, edge attributes in matric A are two-dimensional vectors: $\langle d\_type, d\_rel\rangle$.

In summary, the matrix V allows us to characterize the size of the read/write set of each transaction in the workload. Meanwhile, the matrices A and E capture the key distribution and identify dependencies that need to be handled either in the database, in the middle tier, or both.

4.2.2 Graph embedding and isolation prediction. We use the vertice matrix (V), edge matrix (E), and attribute matrix (A) to predict the optimal strategy for specific workloads. One primary challenge is the presence of features with both vertices and edges, requiring the propagation of information across edges to effectively learn the graph's features. Traditional models such as CNNs [14] are not readily applicable due to the complex, non-Euclidean structure of our inputs. Inspired by MPNN [27] and Zhou et. al [51], we divide the isolation level prediction process into two distinct phases: the embedding and prediction, as shown in Figure 5.

The embedding network is designed to embed the workload graph into a high-dimension vector. It adheres to a layer-wise propagation rule [14], incorporating three embedding layers. Each layer takes the embedding vector produced by the previous layer and the edge matrices as input and outputs the updated embedding vector. The edge network serves as input for all layers in the embedding network to reinforce the transaction dependency information in each layer. Dependency information is propagated throughout the network through multiple iterations, enabling global contextual information integration in the output layer.

The operation of one embedding layer with symmetric normalization can be mathematically described as follows [25, 51]

$$\begin{cases} D_i = [A_i^T A_{:,i}]H^{(l-1)} \\ H^{l+1} = \sigma^l(D^{-\frac{1}{2}}\tilde{A}D^{-\frac{1}{2}}W^l H^l) \end{cases} \tag{1}$$

Here, $\sigma^l(*)$ represents the activation function [20] used to facilitate diverse metric learning. A_i^T represents the transposed edge features associated with node v_i, $A_{:,i}$ represents all edge features related to node v_i according to the index matric E; H^l represents the node feature matrix in layer l; In this way, this model has the learning capability on graph-structured data.

The prediction network takes the high-dimensional vector produced by the embedding network as input. It outputs a three-dimensional vector, with each field representing the probability of the isolation level being optimal. We finally chose the isolation level with the highest probability. In the prediction network, we adopt a two-layer perceptron with one hidden layer and an output layer, which is particularly effective at extracting performance-related features. The hidden layer conducts data abstraction on H^{t_0} and outputs an abstracted matrix H^{t_1}. Then, the output layer employs a log-softmax function to obtain the log probabilities for each class based on H^{t_1}.

4.2.3 Data collection and labeling. Our modeling approach is somewhat general and not designed specifically for specific workloads. However, in practice, we train the model separately for each type of benchmark for efficiency considerations. Taking YCSB+T as an example, we generate 600 random workloads with varying read/write ratios and key distributions. Each workload is executed under each isolation level for 10 seconds, with sampling intervals of 1 second, and the optimal isolation level is labeled based on throughput. We follow the same process for data collection and labeling in Smallbank.

4.2.4 Model training. In TxnSails, we train the embedding and prediction network together and use cross-entropy loss for multi-class classification. Backpropagation involves calculating the gradients of the loss function concerning the parameters of the graph model. First, the gradient is computed for the output layer. Then,

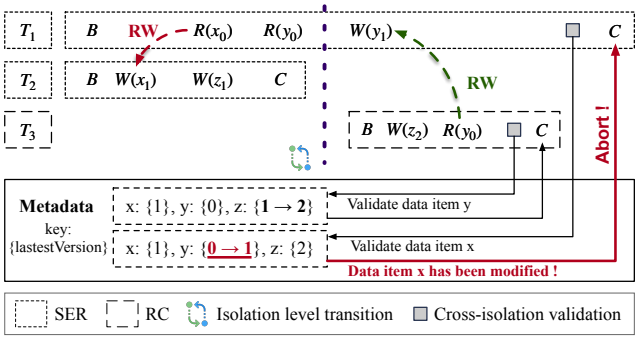


Figure 6: Cross-isolation validation

using the chain rule, these gradients are propagated backwards through the whole network, updating the parameters of each layer. For embedding layers, this process includes computing gradients for both vertex features and transformation matrices derived from edge attributes.

4.3 Cross-isolation Validation

If the predicted optimal isolation level changes, TxnSails will adapt from the previous isolation level I_{old} to the predicted isolation level I_{new}. Non-serializable scheduling may occur during the isolation level transition process. We provide a formal proof of this in Section 5.2.

Example 3. *Figure 6 illustrates non-serializable scheduling during the transition from SER to RC after T_2 commits, making T_1 and T_2 operate under SER while T_3 operates under RC. In this scenario, T_1 is expected to be aborted to ensure SER, however, existing RDBMSs do not handle dependencies between transactions under different isolation levels, allowing T_1 to commit successfully, leading to the non-serializable scheduling. Note that when transactions T_1, T_2, and T_3 are all executed under SER, the concurrency control in RDBMS prevents such non-serializable scheduling.* □

We need to explicitly consider the situations of cross-isolation transitions to ensure the correct transaction execution during the process. A straightforward approach is to wait for all transactions to complete under the previous isolation level before making the transition. In the example above, this would mean blocking T_3 until T_1 commits. However, it can result in prolonged system downtime, especially when there are long-running uncommitted transactions. Another possible approach is to abort these uncommitted transactions and retry them after the transition, which leads to a high abort rate. To mitigate these negative impacts, TxnSails employs a cross-isolation validation (CIV) mechanism that ensures serializability and allows for non-blocking transaction execution without a significant increase in aborts. Specifically, we extend the vulnerable dependency under the single isolation level in Definition 6 to the cross-isolation vulnerable dependency, defined as follows:

Definition 7 (Cross-isolation vulnerable dependency). *The cross-isolation vulnerable dependency is defined as $T_j \xrightarrow{rw} T_k$ in chain $T_i \xrightarrow{rw} T_j \xrightarrow{rw} T_k$ where three transactions are executed under two different isolation levels.* □

We generalize Theorem 2 to obtain corollary 1 and prove it in Section 5.2.

Corollary 1. *For any cross-isolation vulnerable dependency $T_i \xrightarrow{rw} T_j$, if T_i commits before T_j, then the transaction scheduling during isolation transition is serializable.* □

Based on Corollary 1, we implement our CIV mechanism by detecting all cross-isolation vulnerable dependencies during the isolation-level transition and ensuring the consistency of the commit and dependency orders. The CIV mechanism includes three steps. (1) When the system transitions from the current isolation I_{old} to the optimal isolation level I_{new}, the middle tier blocks new transactions from entering the validation phase until all transactions that have entered the validation phase before the transition commit or abort. Importantly, we only block transactions to enter the validation phase. Transactions can execute normally without blocking. (2) After that, the transaction that has completed the execution phase enters the cross-isolation validation phase. During the cross-isolation validation phase, transactions request validation locks according to the stricter locking method of either I_{old} or I_{new} to ensure that all cross-isolation vulnerable dependencies can be detected. For example, when transitioning from SI to RC, the transaction in the cross-isolation validation phase requests validation locks following RC's validation locking method, regardless of whether it is executed under SI or RC. (3) After acquiring validation locks, transaction T_i first detects vulnerable dependencies of its original isolation level. Then it detects cross-isolation vulnerable dependencies by checking whether a committed transaction modifies its read set (using the same detection method as that in Section 4.1.2). If such modifications are detected, T_i is aborted to ensure the consistency of the commit and dependency orders.

Once all transactions executed under I_{old} are committed/aborted, the transition process ends and new transactions will operate under I_{new}, which are validated in the process described in Section 4.1.2.

5 CORRECTNESS

5.1 Serilizability under Low Isolation Levels

Non-serializable scheduling under each low isolation level accommodates certain specific vulnerable dependencies. According to Theorem 2, a necessary condition for non-serializability is the presence of inconsistent dependencies and commit orders among these vulnerable dependencies. TxnSails identifies static vulnerable dependencies from the transaction templates and ensures that, in transactions involving these dependencies, the commit order aligns with the dependency order as specified in Algorithm 1. This approach maintains SER even when the RDBMS is configured to low isolation levels.

5.2 Serializability under Cross-isolation Levels

In this subsection, we prove that if we can ensure the commit order of T_j and T_k in the cross-isolation vulnerable dependency $T_i \xrightarrow{rw} T_j \xrightarrow{rw} T_k$ aligns with the dependency order, then transactions can achieve SER. We prove this in the following two steps.

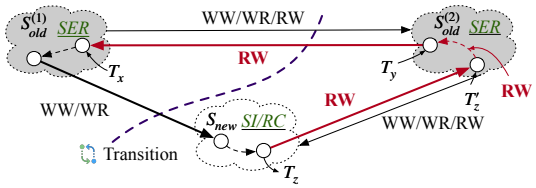


Figure 7: Transition from SER to SI or RC

First, we prove that ❶ *if there is a non-serializable scheduling during the transition, there must be a structure containing two consecutive RW dependencies, $T_i \xrightarrow{rw} T_j \xrightarrow{rw} T_k$, where T_k commits first in this cycle and T_j operates under SER.* Consider T_k as the first transaction to commit in the cycle. This implies the existence of an RW edge from T_j to T_k, as T_k commits first. However, by identifying the static vulnerable dependency at RC, the middle-tier concurrency control algorithm in Section 4.1 ensures T_j commits before T_k for any RW dependency, meaning that T_j operates under SI or SER. If the dependency from T_i to T_j is either a WW or WR dependency, implying that T_i commits before T_j starts. Since T_j is concurrent with T_k by an RW dependency, T_k commits after T_j starts. Thus, T_k must commit before T_j commits, which contradicts the assumption that T_k is the first transaction to commit in the cycle. Therefore, we conclude that if there is a cross-isolation level dependency cycle, there must contain two consecutive RW dependencies $T_i \xrightarrow{rw} T_j \xrightarrow{rw} T_k$, where T_k commits first. Furthermore, if T_i operates under SI, the middle-tier concurrency control algorithm in Section 4.1 can also maintain the commit order between T_j and T_k consistent with their dependency order. Therefore, T_j operates under SER.

We then prove that ❷ *if there is non-serializable scheduling during the transition, there must be at least one cross-isolation vulnerable dependency, $T_i \xrightarrow{rw} T_j$, where 1. T_j commits before T_i commits; 2. T_i operates under SER; 3. T_i commits after the transition starts.* We only consider the transitions involving SER because the transitions between SI and RC are safe.

(1) Transitions from SI/RC to SER. According to the proof in ❶, if there is non-serializable scheduling during the transition, there must be a structure containing two consecutive RW dependencies, $T_i \xrightarrow{rw} T_j \xrightarrow{rw} T_k$, where T_k commits first in this cycle and T_j operates under SER. In this structure, T_j operates under the new isolation level; thus, T_j commits after the transition starts. Thus, T_j and T_k constitute a cross-isolation level vulnerable dependency.

(2) Transitions from SER to SI/RC. For clarity, we categorize the transactions during the transition into three discrete sets:

- $S^{(1)}_{old}$: The set of transactions that operate under I_{old} and have been committed when the transition occurs.
- $S^{(2)}_{old}$: The set of transactions that operate under I_{old} and remain unfinished when the transition occurs.
- S_{new}: The set of transactions that starts after the transition occurs and operate under I_{new}.

The partial orders between transaction sets are depicted in Figure 7. Non-serializable scheduling implies dependency cycle in the dependency graph, which can be classified into two kinds: (a) scheduling involves only transactions in $S^{(2)}_{old}$ and S_{new}; (b) scheduling involves transactions in $S^{(1)}_{old}$, $S^{(2)}_{old}$ and S_{new}. In the first case, it is similar to the proof of the transition from SI/RC to SER because all transactions are committed after the transition starts.

In the second case, we need to prove that there is at least a cross-isolation vulnerable dependency, $T_j \xrightarrow{rw} T_k$, where $T_j \in S^{(2)}_{old}$. As highlighted by red arrows in Figure 7, (a) transactions in $S^{(2)}_{old}$ operate under SER and start before those in S_{new} start, thus the dependency from transactions in S_{new} to those in $S^{(2)}_{old}$ are RW dependencies. (b) the dependencies between transactions within $S^{(2)}_{old}$ can only be RW dependencies because they are concurrent transactions. (c) transactions in $S^{(1)}_{old}$ commits before those in $S^{(2)}_{old}$, thus dependencies from transactions in $S^{(2)}_{old}$ to those in $S^{(1)}_{old}$ must be RW dependencies.

Consider a counter-example: all T_j in cross-isolation vulnerable dependencies, $T_j \xrightarrow{rw} T_k$, has committed before the transition starts, i.e., $T_j \in S^{(1)}_{old}$. In this case, for any dependency $T_y \xrightarrow{rw} T_z$, where $T_y \in S^{(2)}_{old}$ and $T_z \in S^{(1)}_{old}$, the precede dependency can not be RW, which means $T_x \xrightarrow{ww/wr} T_y$, thus $T_x \in S^{(1)}_{old}$. In this case, a dependency cycle cannot contain transactions in all three transaction sets. Therefore, the counter-example does not exist; at least a precede RW dependency of T_y must exist. Then, there is a structure with two consecutive RW dependencies, $T_x \xrightarrow{rw} T_y \xrightarrow{rw} T_z$, where $T_z \in S^{(1)}_{old}$ commits before $T_y \in S^{(2)}_{old}$ and T_y commits after the transition starts. Thus, T_y and T_z constitute a cross-isolation level vulnerable dependency.

Therefore, we conclude that for transitions from SER to SI/RC or from SI/RC to SER, if there is non-serializable scheduling, there is at least one cross-isolation vulnerable dependency, $T_j \xrightarrow{rw} T_k$, in two consecutive RW dependencies, transaction T_j operates under the SER and commits after the transition starts.

Serializability. CIV ensures that the transaction T_j involving the cross-isolation vulnerable dependency, $T_j \xrightarrow{rw} T_k$, commits before T_k commits. Thus, the transaction scheduling during the isolation level transition achieves SER.

6 EVALUATIONS

In this section, we evaluate TxnSails's performance compared to state-of-the-art solutions. Our goal is to validate two critical aspects empirically: (1) TxnSails's effectiveness in adaptively selecting the appropriate isolation level for dynamic workloads (Section 6.2); and (2) TxnSails's performance superiority over state-of-the-art solutions across a variety of scenarios (Section 6.3).

6.1 Setup

We conducted our experiments on a server equipped with an AMD EPYC 7K62 Processor, including 16 cores, 64 GB of DRAM, and a 1 TB SSD. The operation system was CentOS Linux release 7.9.

6.1.1 Implementation. TxnSails was implemented on the codebase of BenchBase [19]. Our graph learning model was trained

to utilize the *torch_geometric* library. To integrate these components, approximately 4k lines of Java (3.5k+ lines) and Python (0.4k+ lines) code were modified, with the complete codebase available at [5]. To ensure cross-platform compatibility and efficiency, the Python and Java components communicate via *sockets* using a predefined message format.

6.1.2 Default Configuration. We deployed PostgreSQL 15.2 [1] as the database engine, which employs MVCC to implement three distinct isolation levels: Read Committed (RC), Snapshot isolation (SI), and Serializable (SER) (by SSI [15]). Under RC, the system can read the most recently committed version, while both SI and SER maintain a view of the data as it existed at the start of the transaction, thereby observing the committed version from that point in time. To prevent dirty writes, write locks are enforced at all isolation levels. For our database configuration, we allocated a buffer pool size of 24GB, limited the maximum number of connections to 2000, and established a lock wait timeout of 100 ms. By default, the experiments were conducted using 128 client terminals. To eliminate network-related variables from affecting the results, both TxnSails and PostgreSQL were deployed on the same server.

6.1.3 Baselines. For an apples-to-apples comparison, we implemented existing approaches in the same framework as TxnSails.

*(1) Serializable (**SER**).* This baseline executes the workload under a SER level by RDBMS.

*(2) & (3) Conflict materialization (**RC+ELM** [9] and **SI+ELM** [8]).* This approach employs an external lock manager. Workloads are modified to include an extra update operation on the lock manager before other operations. This modification is applied under RC and SI levels, denoting RC+ELM and SI+ELM, respectively.

*(4) & (5) Promotion (**RC+Promotion** [43] and **SI+Promotion** [8]).* This approach promotes a read into a write operation, altering an RW into a WW dependency. This cannot be achieved in common databases (e.g., PostgreSQL) by simply replacing *SELECT ... FOR UPDATE* with *SELECT*, as the former only prevents some but not all interleavings that could result in non-serializable scheduling [8]. Instead, we convert the *SELECT* statement into a non-modifying *UPDATE*. It applies similarly to both RC and SI, denoting RC+Promotion and SI+Promotion, respectively.

It is worth noting that several modifications of the same approach exist to ensure serializability and we adopted the most effective modification based on the findings of Alomari et al. [9]. For instance, in the case of the Promotion strategy within the SmallBank benchmark under SI, modifying the *WriteCheck* template rather than the *Balance* or *TransactSavings* templates yields the best performance. Also, we have implemented our validation-based concurrency control (Section 4.1) for both RC and SI levels, denoted as **TxnSails-RC** and **TxnSails-SI**, respectively.

6.1.4 Benchmarks. Three benchmarks were conducted as follows.

SmallBank [8]. This benchmark populates the database with 400k accounts, each having associated checking and savings accounts. Transactions are selected by each client using a uniform distribution. To simulate transactional access skew, we employ a Zipfian distribution with a default *skew factor* of 0.7.

YCSB+T [18]. This benchmark generates synthetic workloads emulating large-scale Internet applications. In our setup, the *usertable* consists of 4 million records, each 1KB in size, totaling 4GB. The *skew factor*, set by default to 0.7, controls the distribution of accessed data items, with higher values increasing data contention. Each default transaction involves 10 operations, with a 90% probability of being a read and a 10% probability of being a write.

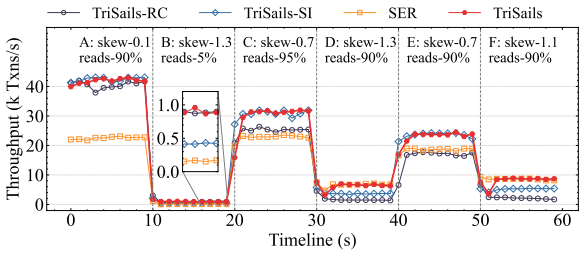


Figure 8: Workload shifting by YCSB

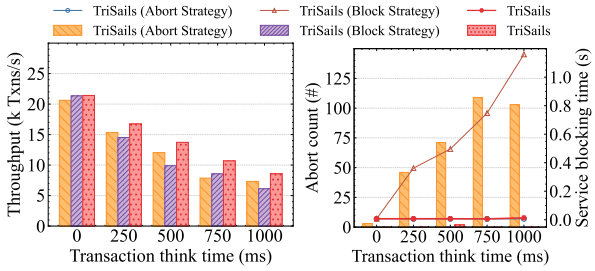


Figure 9: Comparasion of transition mechanisms by YCSB

TPC-C [4, 42]. We use the TPC-C benchmark, which modified the schema and templates to convert all predicate reads into key-based accesses. It includes 5 transaction templates: NewOrder, Payment, OrderStatus, Delivery, and StockLevel. Our tests host 32 warehouses, with each containing about 100MB of data. Following previous works [28, 47], we exclude user data errors that cause about 1% of NewOrder transactions to abort.

6.2 Ablation Study

In this part, we evaluate the effectiveness of the self-adaptive isolation level selection and isolation transition in TxnSails.

6.2.1 Self-Adaptive Isolation Level Selection. We first evaluate the selection of self-adaptive isolation level by varying the workload every 10 seconds across six distinct scenarios. The experimental results are illustrated in Figure 8. We sample the workload at 1-second intervals. The results demonstrate that different isolation levels perform variably under different workloads: SI performs well in low-skew scenarios (A, C, E), SER is more effective in high-skew scenarios with a lower percentage of write operations (D, F), and RC excels in high skew scenarios with a high percentage of writes (B). Across all tested dynamic scenarios, TxnSails successfully adapts to optimal isolation level. Specifically, the graph learning model in TxnSails identifies that SI is suitable for scenarios with fewer conflicts due to its higher concurrency and lower data access overhead (i.e., one-time timestamp acquisition). Conversely, RC is ideal for scenarios with higher conflict rates and more write operations, as it efficiently handles concurrent writes (SI aborts concurrent writes, while RC allows them to commit).

6.2.2 Cross-Isolation Level Validation. We evaluate TxnSails using various transition mechanisms mentioned in Section 4.3 by

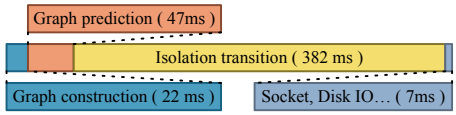
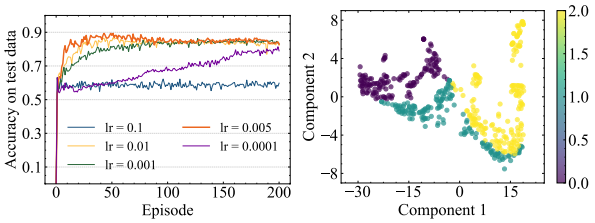
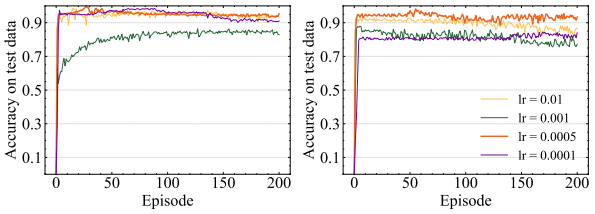


Figure 10: Breakdown during the transition by YCSB



(a) Accuarcy **(b) Extracted features**

Figure 11: Model training metrics by YCSB

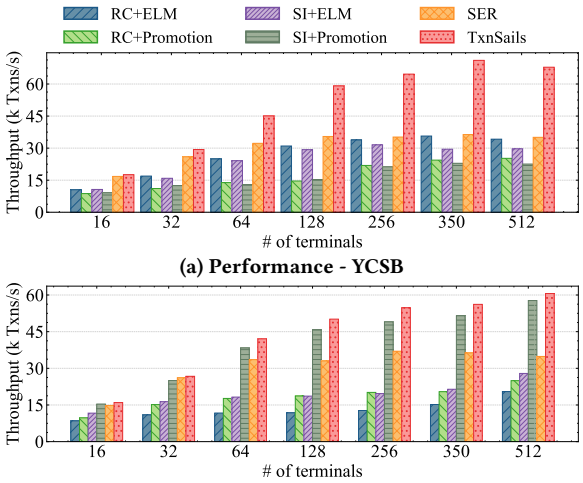


(a) Accuarcy by smallbank **(b) Accuarcy by TPC-C**

Figure 12: Model training metrics

YCSB. To simulate different execution latencies, we adjust the "think time" parameter and transition the workload from D to E. We record the performance metrics, including abort count and blocked time, for each algorithm during the transition. The result is shown in Figure 9. As think time increases, transaction execution latency also rises, and the abort strategy results in a higher number of aborted transactions, while the blocking time associated with the blocking strategy also increases accordingly. In contrast, the cross-isolation validation mechanism outperforms the other two mechanisms by up to 35.9% and 40.3%, while maintaining minimal transaction aborts and negligible blocking time. This enhanced performance can be attributed to TxnSails's ability to avoid actively blocking or rolling back incoming transactions, then the impact on normal transaction execution is minimized while maintaining serializable scheduling.

6.2.3 Graph Model: Construction, Training, and Prediction. Figure 10 illustrates the overhead of workload transition, which takes approximately 450 milliseconds—negligible for longer workloads. Specifically, graph construction and prediction require 22 milliseconds and 47 milliseconds, respectively, while over 80% of the time is spent on transition, from initiating the transition to all connections adopting the new isolation level, closely tied to the largest transaction execution latency. Notably, the prediction in Figure 8 was inaccurate for 1 or 2 seconds at the 30-second and 50-second marks due to the collector sampling transactions from the previous workload during the transition. However, the model successfully transitions to the optimal isolation level in subsequent prediction cycles. The overhead of the learned model is minimal, with less than a 2.5% difference in throughput between using the graph model and not using it.



(a) Performance - YCSB

(b) Performance - SmallBank

Figure 13: Impact of client terminal numbers

We also compare the training process at various learning rates. As shown in Figure 11a, a small learning rate (0.0001) results in slow training due to minimal weight updates, while a large learning rate (0.1 or greater) can lead to poor accuracy. We found that a learning rate of 0.005 quickly achieves approximately 86% accuracy on test workloads. To visualize the high-dimensional vectors produced by our model, we use t-SNE [3] for nonlinear dimensionality reduction, mapping them into two dimensions and plotting them with their true labels in Figure 11b. Most workloads are accurately distinguished, with errors primarily occurring at the boundaries between isolation levels, where performance similarities can lead to incorrect predictions that do not significantly impact overall performance. We further illustrate the training accuracy for Smallbank and TPC-C in Figure 12. During the training process, we noticed an imbalance among the three types of labels. For example, in the Smallbank benchmark, TxnSails-SI consistently outperformed the other two in various scenarios. Inspired by downsampling techniques, we decided to reduce the training data for certain classifications to balance the dataset. We set the model's learning rate to 0.0005. After 10 rounds of training, the accuracy stabilized at 95.1% for Smallbank and 91.7% for TPC-C.

Insight. One isolation level does not fits all workloads. In low-skew scenarios, SI outperforms RC; in high-skew scenarios with fewer writes, SER is the most effective; and in high-skew scenarios with intensive writes, RC is more suitable. TxnSails efficiently adapts isolation levels to optimize performance for dynamic workloads using the proposed fast isolation level transition technique.

6.3 Comparision to State-of-the-art Solutions

We evaluate TxnSails against state-of-the-art solutions over YCSB, SmallBank, and TPC-C benchmarks.

6.3.1 Scalability. This part evaluates the scalability under various numbers of client terminals. The results are shown in Figure 13. TxnSails outperforms other solutions by up to 4.04x and 4.29x by YCSB and SmallBank, respectively. As client terminals increase, TxnSails consistently outperforms, primarily due to the benefits

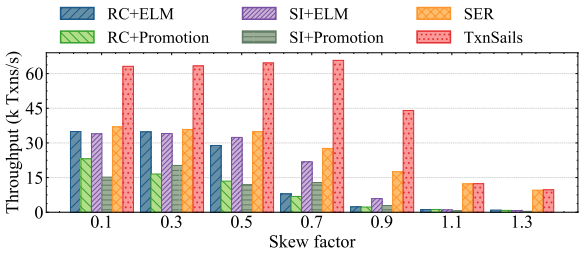
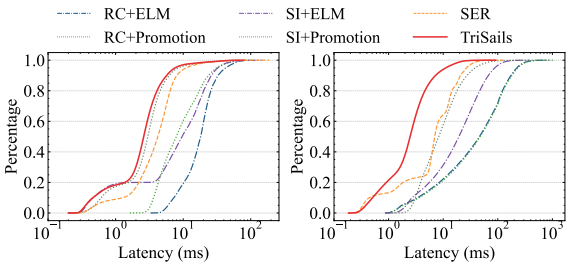


Figure 14: Impact of data contention by YCSB



(a) *Skew factor* is 0.1 - YCSB **(b) *Skew factor* is 0.9 - YCSB**

Figure 15: Analysis of latency CDF by YCSB

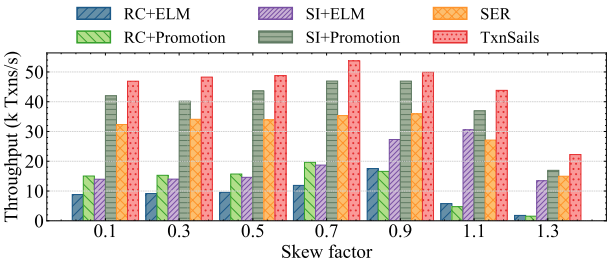
of configuring the database to a lower isolation level, which outweighs the overhead associated with external concurrency control management. In contrast, SER outperforms most other solutions, as these solutions often introduce additional write operations that restrict concurrency and require updating extensive locking information. The notable exception occurs in the SmallBank workload, where the SI+Promotion method surpasses SER. This improvement can be largely attributed to the modification of a limited number of transaction templates within SmallBank.

6.3.2 Impact of Data Contention. This part studies the impact of data contention. We vary the *skew factor* from 0.1 to 1.3 to simulate different levels of data contention in YCSB, and vary *hotspot probability* from 10% to 90% and fix the number of hotspots to 10 and 100 to simulate data contention in SmallBnak.

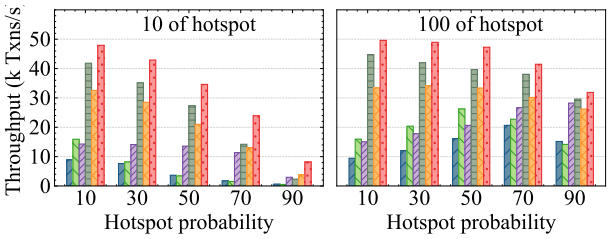
In YCSB, TxnSails outperforms other solutions by up to 26.7x and 2.5x better than the second-best solution SER, as depicted in Figure 14. In high contention scenarios, TxnSails becomes less efficient, causing the validation costs to outweigh the benefits of using a lower isolation level. We further analyze the latency distribution of transactions with skew factors of 0.1 and 0.9 using cumulative distribution function (CDF) plots, as shown in Figure 15. In all scenarios, TxnSails reduces the latency of transactions.

In SmallBank, TxnSails consistently outperforms other solutions by up to a 15.61x improvement and a 2.2x better than the second-best solution SER, as depicted in Figure 16. Unlike YCSB, SmallBank only needs to validate a small portion of read and write operations. As the skew factor increases, TxnSails maintains its advantage over SER by employing a lower isolation level. As hotspot size increases, the reduced conflicts between transactions make the performance advantage of TxnSails less pronounced.

6.3.3 Impact of Write/Read Ratios. This part evaluates the performance of TxnSails by varying the percentage of write operations with YCSB, using the *skew factors* of 0.1 and 0.7. In read-write scenarios in Figure 17a and 17b, TxnSails can still outperform other

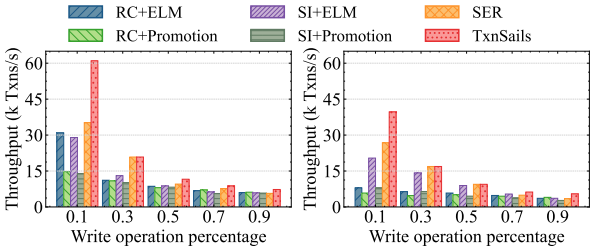


(a) Performance with *skew factors*

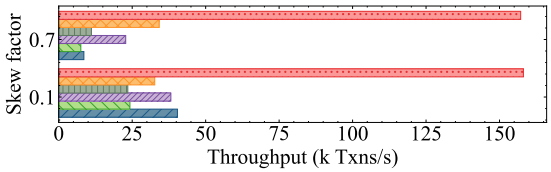


(b) Performance with fixed number of hotspots

Figure 16: Impact of data contention by SmallBank



(a) Skew factor is 0.1 - YCSB **(b) Skew factor is 0.7 - YCSB**



(c) Read only transactions - YCSB

Figure 17: Impact of write/read ratio by YCSB

solutions up to 6.82x. As the percentage of write operations increases, the performance gap narrows as verification overhead at lower isolation levels increases. TxnSails transitions from using the SI to RC, as the FCW strategy raises the abort rate in scenarios with a high percentage of write operations.

We also evaluated the performance in read-only scenarios in Figure 17c. TxnSails achieves performance up to 4.8x higher than SER and up to 20.9x higher than others. TxnSails adopts to SI level as its in-memory validation is nearly costless and rarely fails. Other solutions convert read operations to write operations, thereby restricting concurrency. This also highlights that when a database is configured to be serializable, there is a significant performance loss compared to SI, particularly in read-only scenarios, despite the utilization of snapshots without RW conflicts.

6.3.4 Impact of Templates Percentages. In complex workloads like SmallBank and TPC-C, only certain transaction templates lead to

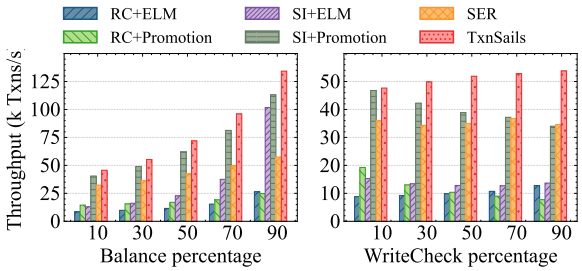


Figure 18: Impact of templates percentage by SmallBank

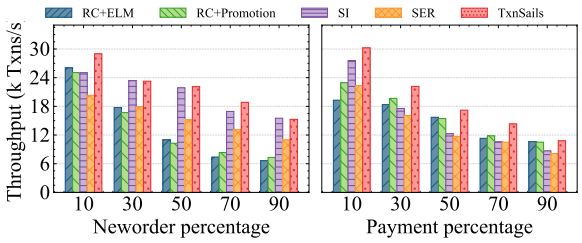


Figure 19: Impact of templates percentage by TPC-C

data anomalies, so modifying these templates can ensure serializability under low isolation levels. This part compares different solutions by varying the percentage of critical transaction templates.

In SmallBank, we evaluate the proportions of the read-only *Balance* and write transaction *WriteCheck*, as shown in Figure 18. As the proportion of *Balance* transactions increases, performance improves; however, RC+ELM and RC+Promotion introduce additional writes in *Balance*, leading to increased WW conflicts. In contrast, SI+ELM and SI+Promotion perform better since they do not modify *Balance* transactions. TxnSails-RC must detect RW dependencies from *Balance*, increasing overhead as their proportion rises. Thus, TxnSails transitions to SI in this scenario, achieving up to a 6.2x performance gain over RC+ELM and RC+Promotion. Conversely, as the proportion of *WriteCheck* transactions increases, concurrency decreases, leading to worse performance for SI+ELM and SI+Promotion. However, TxnSails's performance advantage becomes more pronounced, as it maintains consistent commit and dependency orders through validation without modifying the workload. At 90% *WriteCheck* transactions, TxnSails improves performance by 58.1% compared to SI+Promotion and achieves 2.3x the performance of SER.

In the TPC-C, it executes serializably under SI, eliminating the need for validation in SI. The critical *NewOrder* and *Payment* transactions require modifications by RC+ELM and RC+Promotion, increasing write conflicts on *NewOrder*, resulting in a performance disadvantage compared to TxnSails, which can achieve up to 2.54x their performance. Due to high contention on the warehouse relation, validation overheads are generally higher, except when the proportion of *NewOrder* is 0.1, where TxnSails shows a 15% improvement over SI. In other scenarios, TxnSails adapts to SI. *Payment* transactions are more write-intensive, prompting TxnSails to set the database to RC, which achieves a 40.2% performance improvement over SI. Compared to other solutions, TxnSails achieves up to a 56.7% performance improvement.

Insight. Current research often limits concurrency and scalability by replacing read locks with write locks, whereas TxnSails employs validation-based concurrency control to enhance this. In low to medium contention scenarios, TxnSails performs better at lower isolation levels, but in high contention situations, it benefits from SER due to increased transaction conflicts. Although TxnSails excels at the SI in read-only scenarios, it adapts to the RC as write operations increase to reduce the impact on concurrent writes. Additionally, the structure and proportions of transaction templates are vital for determining the optimal isolation level and performance. Ideally, it is possible to achieve serializability without requiring a serializable isolation level.

7 RELATED WORK

Concurrency control to gurantee SER. SER is regarded as the gold standard for transaction processing as its execution aligns with a serial execution. Much of the existing literature on serializability has explored a variety of algorithms, including 2PL and its variants [12, 13], OCC and its variants [31, 32, 41, 48, 49], timestamp ordering and its variants [13], and MVCC [21, 22, 36]. While these algorithms effectively mitigate all forms of anomalies associated with concurrent transaction execution, they often incur significant costs [12, 28, 29, 38, 45, 47] . In this paper, TxnSails considers a broader range of isolation levels and employs middle-tier concurrency control algorithms to ensure SER under low isolation levels, thereby enhancing efficiency.

Serializable scheduling under low isolation levels. Recent studies have revealed that scheduling entire transaction workloads under low isolation levels can still achieve serializability by adjusting specific query patterns. For example, Fekete provides necessary and sufficient conditions for SI to achieve serializable scheduling [7, 23]. A notable example is the TPC-C benchmark, which remains serializable even when executed under the SI level. Ketsman [30] investigated the characteristics of non-serializable scheduling under Read Committed (RC) and Read Uncommitted isolation levels. This theoretical framework has been further refined with functional constraints by Vandevoort et al. [43]. Building on these theoretical foundations, it has been shown that a workload can achieve a serializable schedule through modifications to the application logic via transaction templates, thereby enhancing performance [42]. In this context, TxnSails leverages these theoretical insights to implement serializable scheduling accurately and efficiently across various isolation levels, all without modification to application logic.

Another line of research focuses on building concurrency control at the application level, independent of the isolation mechanisms provided by RDBMSs. For instance, some developers emphasize application-level concurrency control using mechanisms such as Java's ReentrantLock or *key-value* stores like Redis [2]. Tang et al. [39, 40, 44] provide insights into ad-hoc transactions, which employ flexible and efficient concurrency control on the application side. Bailis et al. [11] introduce the application-dependent correctness criterion known as *I-confluence*, which assesses whether coordination-free transaction execution preserves application invariants. Conway et al. [16] employ monotonicity analysis to eliminate the need for coordination in distributed applications. TxnSails leverages the isolation levels of RDBMS while additionally implementing concurrency control through workload analysis

between the application and the RDBMS. Furthermore, TxnSails efficiently ensures serializability by adaptively selecting appropriate isolation levels.

8 CONCLUSION

In this paper, we present TxnSails, an efficient middle-tier approach that achieves serializability by strategically selecting between serializable and low isolation levels for dynamic workloads. TxnSails introduces a unified method to enforce the commit order to be consistent with the vulnerable dependency order, ensuring serializability in both single-isolation and cross-isolation scenarios. Moreover, TxnSails adopts a graph neural network model to adaptively predict the optimal isolation levels, achieving further performance improvement. The results show that TxnSails can self-adaptively select the optimal isolation level and significantly outperform the state-of-the-art solutions.

REFERENCES

[1] 2024. PostgreSQL: The World's Most Advanced Open Source Relational Database. https://www.postgresql.org/.

[2] 2024. Redis - The Real-time Data Platform. https://redis.io/.

[3] 2024. t-distributed stochastic neighbor embedding. https://en.wikipedia.org/wiki/T-distributed_stochastic_neighbor_embedding.

[4] 2024. TPC-C: On-Line Transaction Processing Benchmark. http://www.tpc.org/tpcc/.

[5] 2024. TxnSails: Achieving Serializable Transaction Scheduling with Self-Adaptive Isolation Level Selection. https://anonymous.4open.science/r/TxnSail-3C11/README.md.

[6] Mohammad Alomari et al. 2009. Ensuring serializable executions with snapshot isolation dbms. (2009).

[7] Mohammad Alomari, Michael Cahill, Alan Fekete, and Uwe Röhm. 2008. Serializable executions with snapshot isolation: Modifying application code or mixing isolation levels?. In *Database Systems for Advanced Applications: 13th International Conference, DASFAA 2008, New Delhi, India, March 19-21, 2008. Proceedings 13*. Springer, 267–281.

[8] Mohammad Alomari, Michael J. Cahill, Alan D. Fekete, and Uwe Röhm. 2008. The Cost of Serializability on Platforms That Use Snapshot Isolation. In *ICDE*. IEEE Computer Society, 576–585.

[9] Mohammad Alomari and Alan D. Fekete. 2015. Serializable use of Read Committed isolation level. In *AICCSA*. IEEE Computer Society, 1–8.

[10] Peter Bailis, Aaron Davidson, Alan D. Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Highly Available Transactions: Virtues and Limitations. *Proc. VLDB Endow.* 7, 3 (2013), 181–192.

[11] Peter Bailis, Alan D. Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination Avoidance in Database Systems. *Proc. VLDB Endow.* 8, 3 (2014), 185–196.

[12] Claude Barthels, Ingo Müller, Konstantin Taranov, Gustavo Alonso, and Torsten Hoefler. 2019. Strong consistency is not hard to get: Two-Phase Locking and Two-Phase Commit on Thousands of Cores. *Proc. VLDB Endow.* 12, 13 (2019), 2325–2338.

[13] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.* 13, 2 (1981), 185–221.

[14] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. 2014. Spectral Networks and Locally Connected Networks on Graphs. In *ICLR*.

[15] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2008. Serializable isolation for snapshot databases. In *SIGMOD Conference*. ACM, 729–738.

[16] Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. 2012. Logic and lattices for distributed programming. In *SoCC*. ACM, 1.

[17] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SoCC*. ACM, 143–154.

[18] Akon Dey, Alan D. Fekete, Raghunath Nambiar, and Uwe Röhm. 2014. YCSB+T: Benchmarking web-scale transactional databases. In *ICDE Workshops*. IEEE Computer Society, 223–230.

[19] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.* 7, 4 (2013), 277–288.

[20] Shiv Ram Dubey, Satish Kumar Singh, and Bidyut Baran Chaudhuri. 2022. Activation functions in deep learning: A comprehensive survey and benchmark. *Neurocomputing* 503 (2022), 92–108.

[21] Dominik Durner and Thomas Neumann. 2019. No false negatives: Accepting all useful schedules in a fast serializable many-core system. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 734–745.

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

[23] Alan D. Fekete. 2005. Allocating isolation levels to transactions. In *PODS*. ACM, 206–215.

[24] Alan D. Fekete, Elizabeth J. O'Neil, and Patrick E. O'Neil. 2004. A Read-Only Transaction Anomaly Under Snapshot Isolation. *SIGMOD Rec.* 33, 3 (2004), 12–14.

[25] Satoshi Furutani, Toshiki Shibahara, Mitsuaki Akiyama, Kunio Hato, and Masaki Aida. 2019. Graph Signal Processing for Directed Graphs Based on the Hermitian Laplacian. In *ECML/PKDD (1) (Lecture Notes in Computer Science)*, Vol. 11906. Springer, 447–463.

[26] Yifan Gan, Xueyuan Ren, Drew Ripberger, Spyros Blanas, and Yang Wang. 2020. IsoDiff: Debugging Anomalies Caused by Weak Isolation. *Proc. VLDB Endow.* 13, 11 (2020), 2773–2786.

[27] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. In *ICML (Proceedings of Machine Learning Research)*, Vol. 70. PMLR, 1263–1272.

[28] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. 2017. An Evaluation of Distributed Concurrency Control. *Proc. VLDB Endow.* 10, 5 (2017), 553–564.

[29] 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 (2020), 629–642.

[30] Bas Ketsman, Christoph Koch, Frank Neven, and Brecht Vandevoort. 2022. Deciding Robustness for Lower SQL Isolation Levels. *ACM Trans. Database Syst.* 47, 4 (2022), 13:1–13:41.

[31] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *SIGMOD Conference*. ACM, 1675–1687.

[32] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *SIGMOD Conference*. ACM, 21–35.

[33] David B. Lomet. 1993. Key Range Locking Strategies for Improved Concurrency. In *VLDB*. Morgan Kaufmann, 655–664.

[34] David B. Lomet, Alan D. Fekete, Rui Wang, and Peter Ward. 2012. Multi-version Concurrency via Timestamp Range Conflict Management. In *ICDE*. IEEE Computer Society, 714–725.

[35] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. 2018. Query-based Workload Forecasting for Self-Driving Database Management Systems. In *SIGMOD Conference*. ACM, 631–645.

[36] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *SIGMOD Conference*. ACM, 677–689.

[37] Dan R. K. Ports and Kevin Grittner. 2012. Serializable Snapshot Isolation in PostgreSQL. *Proc. VLDB Endow.* 5, 12 (2012), 1850–1861.

[38] Takayuki Tanabe, Takashi Hoshino, Hideyuki Kawashima, and Osamu Tatebe. 2020. An Analysis of Concurrency Control Protocols for In-Memory Database with CCBench. *Proc. VLDB Endow.* 13, 13 (2020), 3531–3544.

[39] 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 *SIGMOD Conference*. ACM, 4–18.

[40] Chuzhe Tang, Zhaoguo Wang, Xiaodong Zhang, Qianmian Yu, Binyu Zang, Haibing Guan, and Haibo Chen. 2023. Ad Hoc Transactions: What They Are and Why We Should Care. *SIGMOD Rec.* 52, 1 (2023), 7–15.

[41] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *SOSP*. ACM, 18–32.

[42] Brecht Vandevoort, Bas Ketsman, Christoph Koch, and Frank Neven. 2021. Robustness against Read Committed for Transaction Templates. *Proc. VLDB Endow.* 14, 11 (2021), 2141–2153.

[43] Brecht Vandevoort, Bas Ketsman, Christoph Koch, and Frank Neven. 2022. Robustness Against Read Committed for Transaction Templates with Functional Constraints. In *ICDT (LIPIcs)*, Vol. 220. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 16:1–16:17.

[44] 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 (2024), 3:1–3:43.

[45] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *Proc. VLDB Endow.* 10, 7 (2017), 781–792.

[46] Tao Yu, Zhaonian Zou, Weihua Sun, and Yu Yan. 2024. Refactoring Index Tuning Process with Benefit Estimation. *Proc. VLDB Endow.* 17, 7 (2024), 1528–1541.

[47] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.* 8, 3 (2014), 209–220.

[48] Xiangyao Yu, Andrew Pavlo, Daniel Sánchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *SIGMOD Conference*. ACM, 1629–1642.

[49] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sánchez, Larry Rudolph, and Srinivas Devadas. 2018. Sundial: Harmonizing Concurrency Control and Caching in a Distributed OLTP Database Management System. *Proc. VLDB Endow.* 11, 10 (2018), 1289–1302.

[50] Qiushi Zheng, Zhanhao Zhao, Wei Lu, Chang Yao, Yuxing Chen, Anqun Pan, and Xiaoyong Du. 2024. Lion: Minimizing Distributed Transactions Through Adaptive Replica Provision. In *ICDE*. IEEE, 2012–2025.

[51] Xuanhe Zhou, Ji Sun, Guoliang Li, and Jianhua Feng. 2020. Query Performance Prediction for Concurrent Queries using Graph Embedding. *Proc. VLDB Endow.* 13, 9 (2020), 1416–1428.

[52] Zeheng Zhou, Ying Jiang, Weifeng Liu, Ruifan Wu, Zerong Li, and Wenchao Guan. 2024. A Fast Algorithm for Estimating Two-Dimensional Sample Entropy Based on an Upper Confidence Bound and Monte Carlo Sampling. *Entropy* 26, 2 (2024), 155.