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

Qiyu Zhuang†, Wei Lu†, Shuang Liu†, Yuxing Chen‡, Xinyue Shi†
Zhanhao Zhao†, Yipeng Sun†, Anqun Pan‡, Xiaoyong Du†

†Renmin University of China ‡Tencent Inc.

1.Introduction

Serializable isolation level is regarded as the gold standard for transaction processing due to its ability to prevent all anomalies. However, it also incurs expensive coordination overhead.
Many studies have explored achieving SER by operating at lower isolation levels while modifying specific query patterns within a workload. This approach is driven by two key reasons. First, some RDBMSs cannot strictly guarantee SER, requiring application logic modifications to enforce it. Second, RDBMSs typically offer better performance at lower isolation levels due to their more relaxed ordering requirements.

Current approaches:

1. Build a static dependency graph from transaction templates.
2. Configure the database to a low isolation level and then identify anomaly structures.
3. Eliminate anomaly structures by modifying application logic, e.g., promoting reads to writes for certain SQL statements.

Fig.1 Static Dependency Graph over SmallBank

Problems & Challenges

- Static modification of query patterns is inefficient. Designing an approach that elevates various isolation levels to SER ***without introducing additional writes*** is a complex task.
- Current approaches fail to address the key trade-off between the performance gains and the additional overhead under lower isolation levels. Determining the optimal isolation level requires accurately ***modeling the trade-offs*** is challenging.
- As workloads evolve, the optimal isolation level may adapt over time, making it essential to design ***an efficient and reliable mechanism for transitioning between isolation levels***.

Preliminaries

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.

Vulnerable dependency (Runtime)

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

For any vulnerable dependency $T_i \xrightarrow{rw} T_j$, if T_i ***commits before*** T_j, then the scheduling achieves SER.

2.Overview

TxnSails works in the middle tier between the application tier and database tier, it comprises three main components: ***Analyzer***, ***Executor***, and ***Adapter***.

Analyzer

It builds the static dependency graph for the transaction templates and identifies all the static vulnerable dependencies for each low isolation level.

Executor

It ensures SER when transactions operate at a single low isolation level or during the transition. *Isolation Level Manager* stores the static vulnerable dependencies. Before any transaction *T* starts, it identifies whether involves any static vulnerable dependencies.
Dependency Detector monitors the read and write set, detecting the runtime vulnerable dependencies.
Transaction Scheduler guarantees the consistent between commit order and dependency order.
During the isolation level transition, *Transition Governor* follows a **new corollary**, which extends the single isolation level *Theorem*.

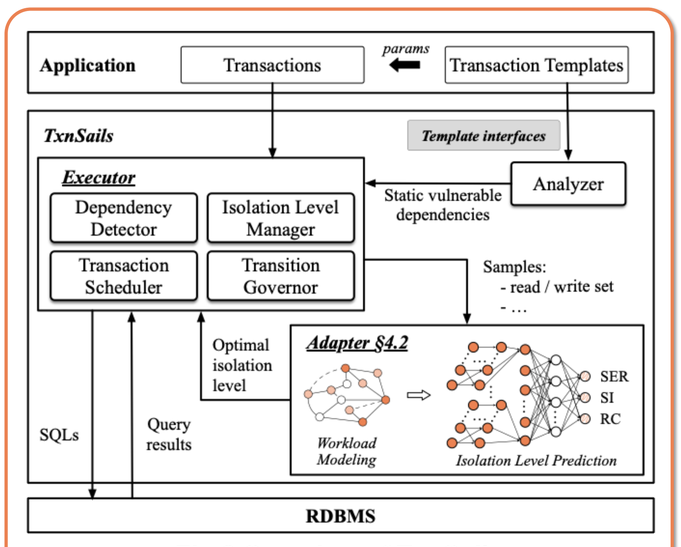

Fig.2 Overview of TxnSails

Adaptor

It samples the real-time transactions and collects their characteristic. Then, it employs a graph-based model to predict the optimal isolation level.

3.Technology Details

Unified middle-tier concurrency control mechanism

TxnSails introduces a middle-tier concurrency control algorithm, which dynamically validates runtime dependencies and schedules their commit order. In particular, it focuses exclusively on vulnerable dependencies identified by the Analyzer and employs a lightweight validation mechanism to further mitigate overhead.

Transaction lifecycle

I. 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. Following the RDBMS transaction execution, TxnSails stores the read/write data items in the thread-local buffer that may induce the vulnerable dependencies;
II. In the validation phase, TxnSails acquires validation locks for data items stored in the buffer. Then, it detects the dependencies among them and aims to schedule the commit order consistent with the identified dependency order;
III. In the commit phase, TxnSails applies modifications to the database and subsequently releases the validation locks.

Example

In the execution phase, after the RDBMS execution ❶, T_{wc} stores the data item *x* in its vread_set and T_{ts} stores *x* in its vwrite_set ❷. In the validation phase of T_{wc}, it acquires the shared validation lock on *x* ❸ and retrieves the latest version of x from either VLT or the RDBMS ❹. 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* ❺.

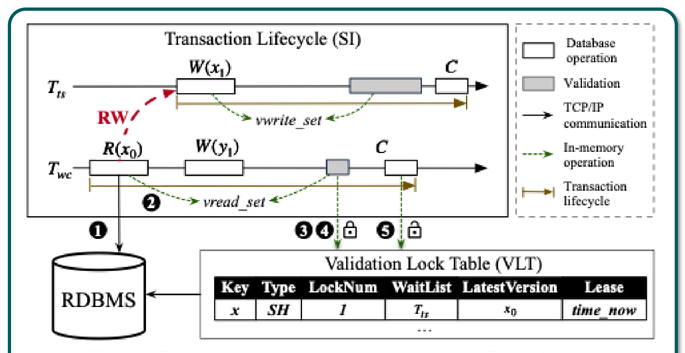

Fig.3 Transaction processing in TxnSails

Self-adaptive isolation level selection

TxnSails adopts transaction dependency graphs to capture workload features and adopts a graph classification model to perform self-adaptive isolation level selection.

Graph construction

TxnSails proposes a graph-structured workload model, where each transaction is mapped to a vertex v_i, and its feature vector v_i is generated by extracting the number of data items in its read and write set. For each edge (v_i, v_j), TxnSails extracts the data dependency type and the involved relations to generate its feature.

Graph embedding and isolation level prediction

The graph model comprises two parts. First, we use a *Graph Embedding Network* to learn and aggregate both vertex and edge features, producing node-level embedded matrix that encodes the local structure and attribute information of the graph. Second, to predict the optimal isolation level, we use a *Graph Classification Network* that learns the mapping from the embedded matrix to perform the end-to-end graph classification to predict the optimal isolation level.

Cross-isolation level validation mechanism

If the predicted optimal isolation level changes, TxnSails will adapt from the previous isolation level I_{old} to the optimal isolation level I_{new}. TxnSails employs a cross-isolation validation (CIV) mechanism that ensures serializability and allows for non-blocking transaction execution.

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 T_j commits after the transition starts.

Corollary

For any cross-isolation vulnerable dependency $T_j \xrightarrow{rw} T_k$, if T_j ***commits before*** T_k , the scheduling achieves SER.

Transition procedure

I. TxnSails blocks new transactions from entering the validation phase until all transactions that have entered the validation phase before the transition commit or abort.
II. Validation locks are required according to the stricter locking method of either I_{old} or I_{new} to ensure that all cross-isolation vulnerable dependencies can be detected.
III. After acquiring validation locks, transaction first detects vulnerable dependencies in its original isolation level. Then, it detects cross-isolation vulnerable dependencies by checking whether a committed transaction modifies its read set.

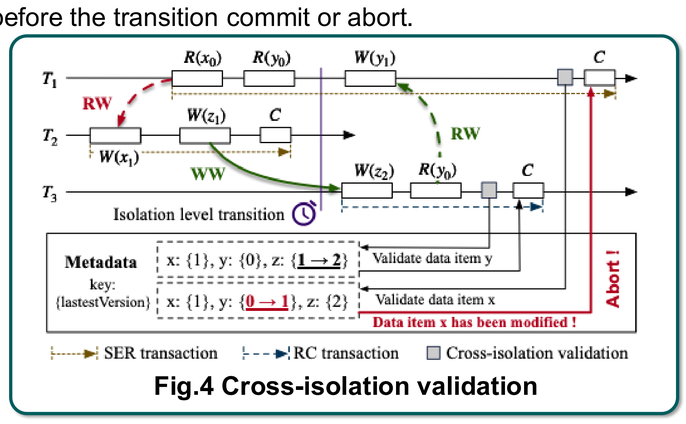

Fig.4 Cross-isolation validation

Once all transactions executed under I_{old} are committed or aborted, the transition stage ends. Then, transactions do not need to undergo the cross-isolation validation.

4.Evaluations

Self-adaptive isolation level selection

We first evaluate selfadaptive isolation level selection by varying the workload every 10s across six distinct scenarios. The experimental results are illustrated in Fig. 5.

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 suitable in high-skew scenarios with little writes (D,F). RC excels in high skew scenarios with more writes (B). ***TxnSails can choose the optimal isolation level across all tested scenarios.***

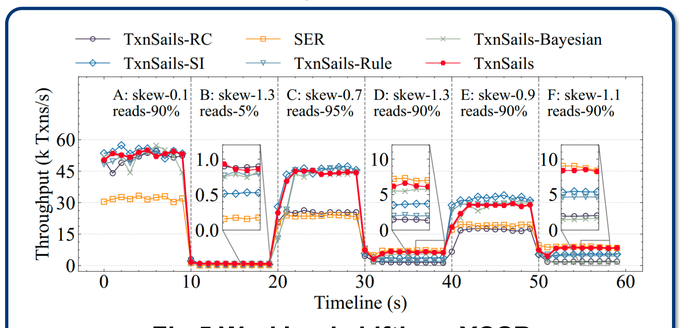

Fig.5 Workload shifting - YCSB

Impact of templates percentages

In complex workloads like SmallBank and TPC-C, only certain transaction templates lead to 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.

As the ratio of Balance transactions increases, performance of all solutions improves. TxnSails transitions to SI in these workloads and achieves up to 6.2× performance gain. As the ratio of WriteCheck transactions increases, the advantage of TxnSails becomes more pronounced as no extra WW conflicts are introduced, outperforming up to 2.3× the performance of SER.

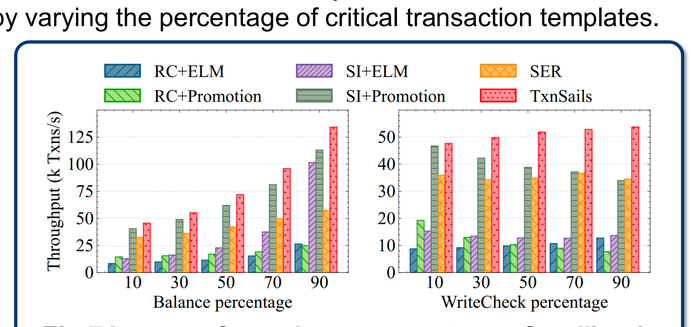

Fig.7 Impact of templates percentage - Smallbank