Brook-2PL: Tolerating High Contention Workloads with A Deadlock-Free Two-Phase Locking Protocol

FARZAD HABIBI, University of California, Irvine, USA
JUNCHENG FANG, University of California, Irvine, USA
TANIA LORIDO-BOTRAN, Roblox, Northeastern University, USA
FAISAL NAWAB, University of California, Irvine, USA

The problem of hotspots remains a critical challenge in high-contention workloads for concurrency control (CC) protocols. Traditional concurrency control approaches encounter significant difficulties under high contention, resulting in excessive transaction aborts and deadlocks. In this paper, we propose *Brook-2PL*, a novel two-phase locking (2PL) protocol that (1) introduces *SLW-Graph* for deadlock-free transaction execution, and (2) proposes *partial transaction chopping* for early lock release. Previous methods suffer from transaction aborts that lead to wasted work and can further burden the system due to their cascading effects. *Brook-2PL* addresses this limitation by statically analyzing a new graph-based dependency structure called *SLW-Graph*, enabling deadlock-free two-phase locking through predetermined lock acquisition. *Brook-2PL* also reduces contention by enabling early lock release using partial transaction chopping and static transaction analysis. We overcome the inherent limitations of traditional transaction chopping by providing a more flexible chopping method. Evaluation using both our synthetic online game store workload and the TPC-C benchmark shows that *Brook-2PL* significantly outperforms state-of-the-art CC protocols. *Brook-2PL* achieves an average speed-up of 2.86× while reducing tail latency (p95) by 48% in the TPC-C benchmark.

CCS Concepts: • **Information systems → Database transaction processing**.

Additional Key Words and Phrases: Concurrency Control, Two-Phase Locking, Deadlock-Free, OLTP

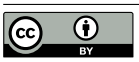
ACM Reference Format:
Farzad Habibi, Juncheng Fang, Tania Lorido-Botran, and Faisal Nawab. 2025. Brook-2PL: Tolerating High Contention Workloads with A Deadlock-Free Two-Phase Locking Protocol. *Proc. ACM Manag. Data* 3, 6 (SIGMOD), Article 302 (December 2025), 27 pages. https://doi.org/10.1145/3769767

1 Introduction

The rapid growth of cloud computing provides an excellent infrastructure for data management, offering scalable and flexible resources to meet the growing demands of modern applications [28]. Many cloud data management systems have been proposed and widely adopted [4, 11, 31, 34]. However, achieving both scalability and consistency under a high contention workload remains challenging [2, 65].

Transaction processing systems use advanced concurrency control protocols to leverage parallelism. However, these protocols still suffer from conflicts that reduce concurrency and throughput [6, 9, 21, 23, 55, 62]. In highly contentious workloads, hotspots—frequently accessed small sets

Authors' Contact Information: Farzad Habibi, University of California, Irvine, USA, habibif@uci.edu; Juncheng Fang, University of California, Irvine, USA, junchf1@uci.edu; Tania Lorido-Botran, Roblox, Northeastern University, USA, tbotran@roblox.com, t.loridobotran@northeastern.edu; Faisal Nawab, University of California, Irvine, USA, nawabf@uci.edu.



ACM 2836-6573/2025/12-ART302
https://doi.org/10.1145/3769767

of database records—pose significant challenges [2, 65]. To maintain strong isolation and serializability, concurrency control protocols must serialize transactions around these hotspots, even though they represent only a small fraction of total transactions. Under traditional concurrency protocols, transactions encountering hotspots may be forced to either wait or abort and restart, further impacting system performance [23, 58].

To improve performance, prior production systems [14, 18, 35] and research efforts [15, 20, 22, 23, 32, 62] have explored early write visibility through various methods. For instance, Bamboo [23] modifies pessimistic locking in the 2PL protocol to allow locks to be released earlier, enhancing parallelism. Their solution involves tracking dirty reads and cascadingly aborting transactions if one transaction aborts. However, this approach continues to experience transaction aborts under high contention, resulting in wasted work and degraded performance. In fact, cascading aborts become especially problematic in heavily contentious workloads, generating additional wasted work and further straining system resources.

Transaction chopping approaches [54], such as IC3 [62], also enable early write visibility through static analysis of transactions to improve performance. However, transaction chopping is rigid in that isolated chops of a transaction follow their own concurrency control protocol and do not interact. This limits the flexibility of acceptable chopping configurations and thus makes them impractical in real applications. These approaches also continue to experience transaction aborts under high contention.

In this work, we propose *Brook-2PL*, a deadlock-free 2PL protocol, to address the problem of hotspots in high-contention workloads by eliminating transaction aborts and enabling earlier lock release.

Brook-2PL statically analyzes a highly contentious set of transactions using a specific dependency graph called *SLW-Graph* (Serializable Wait For Lock Graph). *Brook-2PL* eliminates potential deadlocks by enforcing a predefined locking order within transactions and adjusting certain lock acquisitions to occur earlier in the transaction sequence. Additionally, we show that static analysis of transactions and the removal of cyclic dependencies (deadlocks) can facilitate the early release of locks through a concept we call *partial transaction chopping*. Traditional transaction chopping mechanisms [54, 67] are rigid and impose strict conditions that limit their flexibility in chopping transactions. We relax these assumptions, enabling more flexible and serializable choppings. The main insight of our mechanism is to allow chopping transactions in a *partial*—rather than exclusive—manner.

In summary, this paper makes the following contributions:

- We develop *Brook-2PL*, a new deadlock-free concurrency control protocol that eliminates transaction aborts and significantly reduces wasted work, improving the parallelism of transactions.
- We propose *SLW-Graph*, a graph-based representation of transactions, and introduce lock manipulation techniques to eliminate potential deadlocks.
- We combine *SLW-Graph* with transaction chopping to enable early lock release, enhancing performance by increasing parallelism. Our approach to write visibility surpasses others by identifying more opportunities for early write visibility based on preprocessing and transaction analysis. Transaction chopping has inherent limitations, which we overcome by enabling a greater number of transactions to be chopped via *partial* transaction chopping.
- We design a static analysis algorithm that leverages *SLW-Graph* and *partial transaction chopping* to analyze a set of highly contentious transactions, generating optimized and deadlock-free transactions. *Brook-2PL* generalizes to all workloads by enabling dynamic transaction execution.
- We evaluate *Brook-2PL* under both a synthetic online game store workload and the TPC-C benchmark [57], comparing it with our baseline consisting of state-of-the-art concurrency control

protocols. *Brook-2PL* demonstrates an average speed-up of 2.86× while reducing latency by 48% in the TPC-C benchmark.

Paper Organization. The remainder of this paper is organized as follows: Section 2 provides motivation and background information on database transactions, two-phase locking (2PL), and transaction chopping. Section 3 introduces the design of *Brook-2PL*, including static analysis of transactions using *SLW-Graph* and our proposed *partial transaction chopping* method, as well as our approach to handle dynamic transactions. Section 4 presents experimental results evaluating *Brook-2PL*. Section 5 reviews relevant literature, and Section 6 concludes the paper.

2 Background and Motivation

2.1 Database Transactions

A database transaction is a sequence of operations executed as a single unit of work on a database, ensuring the ACID (atomicity, consistency, isolation, and durability) properties. A transaction T can be defined as an ordered set of operations: $T = \{o_1, o_2, \ldots, o_n\}$, where each o_i represents an operation (such as read or write) on data items within the database.

Serializability in a database system ensures concurrent execution of transactions by enforcing rules that prevent unexpected or incorrect results. Each transaction is treated as if it is the only transaction running, thereby maintaining data integrity. A concurrent schedule is considered serializable if it is equivalent to some sequential execution of its transactions. A stricter form, conflict serializability, holds when a schedule is conflict equivalent to a serial schedule—meaning it involves the same transactions and maintains the relative order of all conflicting operations.

Two-Phase Locking (2PL). Two-phase locking (2PL) is the most widely used class of concurrency control in database systems. In 2PL, reads and writes are synchronized through explicit locks: shared mode for read operations and exclusive mode for write operations. A transaction can operate on a resource only if it acquires the necessary locks. To maintain serializability, 2PL enforces two key rules [6]: (1) Conflicting locks are not allowed at the same time on the same data. Two locks conflict if they both lock the same resource and at least one of them is in exclusive mode. (2) A transaction cannot acquire additional locks once it has released any.

Two-phase locking enforces a growing-then-shrinking lock acquisition rule to ensure serializability, but this can lead to deadlocks, mostly resolved through detection or prevention strategies. In deadlock detection, the system maintains a central wait-for graph that tracks dependencies and periodically checks for cycles during runtime. However, maintaining this graph can create a scalability bottleneck [65]. In deadlock prevention, the system allows only a subset of transactions to wait on locks based on specific criteria. Wound-Wait and Wait-Die are two popular deadlock prevention strategies within 2PL [51]. In both deadlock handling mechanisms, the database system aborts a transaction to break the deadlock cycle, resulting in wasted CPU resources and failing to prevent the same deadlock scenario from reoccurring.

2.2 Transaction Chopping

Transaction chopping breaks down a transaction type into smaller subtransactions, enabling the independent execution of each subtransaction. This independence allows locks to be held only during the execution of each piece. As a result, locks are released earlier, reducing contention and improving concurrency. This approach, referred to as piecewise execution, ensures that locks are held only during the execution of each subtransaction.

However, breaking down a transaction can pose challenges for maintaining serializability. To ensure serializability, transaction chopping relies on a graph called the SC-Graph. This graph

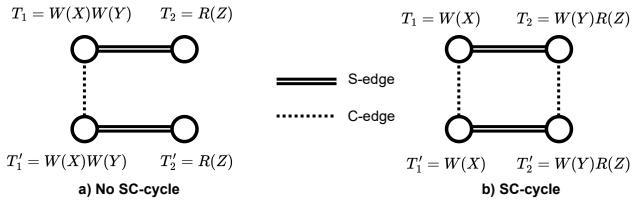


Fig. 1. SC-Graph of two different choppings for $T = W(X)W(Y)R(Z)$: (a) Chopping into $T_1 = W(X)W(Y)$ and $T_2 = R(Z)$, where the SC-Graph has no SC-cycle and is serializable; (b) Chopping into $T_1 = W(X)$ and $T_2 = W(Y)R(Z)$, where the SC-Graph contains an SC-cycle and is not serializable.

verifies that the chopping and piecewise execution of subtransactions are valid and maintain a serializable execution order.

Example. Consider a transaction type $T = W(X)W(Y)R(Z)$ that accesses three different object types (e.g. tables): X, Y, and Z. This transaction type can be chopped into two subtransactions: $T_1 = W(X)W(Y)$ and $T_2 = R(Z)$, allowing each subtransaction to release its locks after its execution is finished. To process the chain of subtransactions, the system will first execute T_1, followed by T_2.

SC-Graph Representation. In an SC-Graph, vertices represent subtransactions, and edges represent the relationships between these subtransactions. Vertices within the same chain are connected by (sibling) S-edges, while vertices in different chains that access the same data item, with at least one being a write operation, are connected by (conflict) C-edges. In an SC-Graph, two instances of each transaction are included. This is necessary because the SC-Graph must capture scenarios where two separate, concurrent instances of the same transaction type conflict with each other. Including two instances in the SC-Graph accurately models these potential self-conflicts.

It has been proven that an SC-Graph without an SC-cycle (a cycle containing both S-edges and C-edges) guarantees serializability, even with piecewise execution [54]. Figure 1a shows the SC-Graph for subtransactions $T_1 = W(X)W(Y)$ and $T_2 = R(Z)$. This chopping is serializable and valid as it does not create an SC-cycle. However, if the initial transaction T were instead chopped into $T_1 = W(X)$ and $T_2 = W(Y)R(Z)$, an SC-cycle would be created, violating serializability (Figure 1b).

The SC-graph of all involved transactions must have no SC-cycles, which limits the practicality of transaction chopping. This is because any practical application with more than a few transactions would typically contain an SC-cycle. One reason for this is that to have no SC-cycles, any pair of a decomposed transaction can only conflict on a single subtransaction.

2.3 Why Deadlocks are Deadly?

Deadlocks are problematic in the system for the following reasons.

Deadlocks are Difficult to Handle. Systems typically handle deadlocks through either prevention or detection. In deadlock prevention, systems abort transactions that could potentially create a deadlock, breaking the waiting cycle between transactions. However, this often results in aborting more transactions than necessary, leading to wasted computational resources and a loss of useful work. Additionally, most prevention mechanisms abort transactions mid-execution, which further wastes the progress made up to that point.

On the other hand, deadlock detection mechanisms present their own challenges, as they require the system to maintain a wait-for graph between transactions. This graph is costly to manage and imposes additional overhead on the system.

Transaction Retrials are Problematic. Deadlocks and transaction retrials can trigger a cascading effect, which exacerbates system instability and can lead to prolonged or even complete service

outages [7, 25–27]. In a system with a continuous load of incoming transactions, retrials can accumulate, forming an artificial feedback loop that generates additional conflicts and further transaction retrials. This cycle intensifies contention within the system, resulting in degraded performance and work amplification.

3 *Brook-2PL* Architecture

In this section, we introduce the design of *Brook-2PL*, which leverages our proposed graph representation, *SLW-Graph* (Section 3.3), for the static analysis of transactions. Using *SLW-Graph*, *Brook-2PL* achieves deadlock-free two-phase locking by preprocessing transactions, detecting potential sources of deadlocks, and providing strategies to eliminate them during execution (Section 3.5). Additionally, *Brook-2PL* leverages a flexible transaction chopping [54] concept that we proposed, called *partial transaction chopping* (Section 3.6), to enable the early release of locks without violating serializability. *Brook-2PL* provides a middleware algorithm (Section 3.7) to preprocess a static set of highly contentious transaction types. In *Brook-2PL*, preprocessed transactions follow a basic 2PL protocol without requiring any deadlock-prevention mechanism. *Brook-2PL* also supports the execution of dynamic transactions—transactions that have not been statically analyzed (Section 3.8).

3.1 Intuition

Assume there are only two types of transactions allowed in the system: $T_1 = \mathrm{W}(A)\mathrm{W}(B)$ and $T_2 = \mathrm{W}(B)\mathrm{W}(A)$. A possible deadlock scenario arises when an instance of transaction T_1 locks a row of table A while, concurrently, an instance of T_2 locks a row of table B to perform their respective write operations. In this case, a deadlock may occur if both transactions attempt to access the same row held by the other. This happens because each transaction is waiting for the other to release its lock. One solution is to abort one of the transactions, but this leads to wasted CPU resources and does not prevent the same deadlock scenario from reoccurring.

A more effective solution is to ensure both transactions follow the same locking order for tables A and B. This approach eliminates the possibility of deadlocks and unnecessary aborts.

Additionally, deadlock-freedom enables early lock release, since transactions are guaranteed to commit without aborting from deadlocks. For example, in the scenario above, T_1 can release its lock on row of table A after the last operation on it, once it is certain the transaction will successfully commit.

However, in real-world systems, tracking all locks across many transactions is challenging. To address this, we propose *Brook-2PL*, which enables the static analysis of transactions as a preprocessing step. By analyzing the *SLW-Graph*, potential deadlocks are identified, and elimination strategies are applied during a preprocessing phase. This "abort-free" property further allows for the early release of locks, reducing contention. *Brook-2PL* introduces *partial transaction chopping* to fully exploit the benefits of early lock release. This technique builds upon the concept of early lock release and extends its potential by identifying additional opportunities within a transaction where locks can be safely released.

3.2 *Brook-2PL*

Brook-2PL introduces a static pre-processing step for the two-phase locking (2PL) concurrency protocol. It converts a predefined set of transactions into a graph structure that is then analyzed prior to execution. This pre-processing analysis ensures deadlock-free operation during runtime, eliminating the need for transaction retrials and allowing for the early release of locks.

All statically analyzed transactions follow the 2PL protocol: locks are acquired during the growing phase, operations are executed, and locks are released during the shrinking phase. *Brook-2PL* does not alter the core 2PL protocol for static transactions; rather, the changes lie in the order and timing

of lock acquisition and release. Systems can maintain their existing 2PL implementations while simply adjusting the order and timing in which locks are acquired and released. This allows for the seamless integration of our approach without modifying the underlying 2PL mechanism.

When dynamic transactions are enabled, they follow a modified 2PL to safely coexist with statically analyzed transactions. If only dynamic transactions are present, *Brook-2PL* falls back to Wound-Wait 2PL. This design enables systems to retain their existing 2PL implementations while incorporating our minimal changes.

3.3 *SLW-Graph*

SLW-Graph represents transactions in a structured manner. Each node in the graph corresponds either to an operation or to a lock associated with an operation (referred to as a hop). Typically, a locking node precedes the operation nodes that modify the value of the locked variable. Within the same transaction, hops are linked sequentially by directed sibling edges (s-edges). When two locking hops from different chains conflict by accessing the same table, they are connected by undirected wait-for edges (w-edges).

A transaction is encoded as a sequence of hops ($[h_1 \cdots h_n]$), forming chains connected by s-edges. Additional transactions contribute further chains to the graph.

Representation. To formalize the *SLW-Graph* in graph-theoretic terms, for a set of transactions T, the corresponding *SLW-Graph* is defined as: $G = (V, E_s, E_w)$. Here, V is the set of vertices or nodes, E_s represents the set of directed sibling edges, and E_w represents the set of undirected wait-for edges.

Hops. Each vertex in the *SLW-Graph* represents a specific component of a transaction. The set of vertices V is composed of four types of nodes:

(1) Operation Nodes (V_O): Representing read/write operations within a transaction.
(2) Shared Locking Nodes (V_{LS}): Representing shared locking operations that precede read operations.
(3) Exclusive Locking Nodes (V_{LE}): Representing exclusive locking operations that precede write operations.
(4) Unlock Nodes (V_U): Representing the nodes responsible for unlocking a set of locks held by the transaction.

Thus, the set of all locking nodes is defined as $V_L = V_{LE} \cup V_{LS}$, and the set of all vertices is defined as: $V = V_L \cup V_O \cup V_U$.

Sibling Edges. Sibling edges (E_s) connect two nodes within the same transaction. If two hops n and n' are part of the same transaction, and one follows the other in execution order, they are connected via a directed s-edge.

Wait-for Edges. Wait-for edges (E_w) represent potential conflicts in resource allocation between different transactions. These edges connect two conflicting locking nodes associated with the same resource, where at least one of them is an exclusive lock. An undirected w-edge indicates a potential situation where one transaction waits for another to release a lock.

Static analysis cannot determine exactly what data items a hop accesses (which rows); therefore, we add a w-edge between two locking hops of different chains if the hops access the ***same table***. Therefore, *Brook-2PL* static analysis operates at the table level. In *Brook-2PL* implementation, programmers can provide annotations on the operations to specify that two nodes accessing the same table do not conflict, in which case the w-edge is removed. An example of such annotations is for operations that commute and, therefore, would not conflict even if accessing the same table.

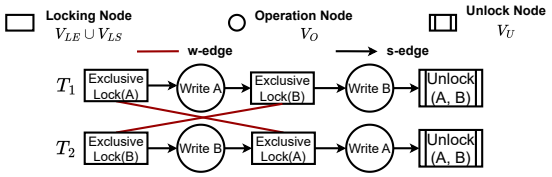


Fig. 2. *SLW-Graph* representation of $T_1 = W(A)W(B)$ and $T_2 = W(B)W(A)$

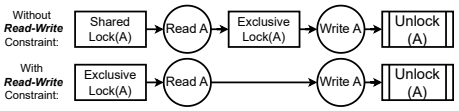


Fig. 3. The effect of the read-write constraint in *SLW-Graph*

Chains. A chain is a sequence of hops $[h_1 \cdots h_n]$ (where $h_i \in V$), connected by s-edges and representing a transaction. Chains are interconnected via w-edges in the *SLW-Graph*.

SLW-cycle (Deadlocks). In the *SLW-Graph*, an SLW-cycle is defined as a *directed* cycle that includes s-*edges* (E_s), *locking hops* (V_L), and *w-edges*, with the constraint that no two w-edges can be adjacent. We prove (see Section 3.4) that if no SLW-cycle exists in the *SLW-Graph*, deadlocks are impossible in runtime. Therefore, by eliminating SLW-cycles, we can guarantee a deadlock-free 2PL.

Example. Figure 2 shows an *SLW-Graph* representation of two transactions, $T_1 = W(A)W(B)$ and $T_2 = W(B)W(A)$. In this graph representation, there is an SLW-cycle, indicating a potential deadlock between these transactions where the lock for resource A is held by T_1 and the lock for resource B is held by T_2.

Read-Write Constraint. Since *Brook-2PL* performs static analysis on the *SLW-Graph* of transactions and can identify when a write follows a read, it provides an opportunity to combine both locks into a single node, reducing potential deadlocks caused by lock upgrades. In each transaction chain, if both read and write operations target the same resource (same table) within a single transaction type, *Brook-2PL* applies a single exclusive lock for both in the *SLW-Graph*. This approach simplifies the graph structure and reduces the risk of deadlocks by avoiding shared lock conflicts.

Figure 3 shows the effect of this constraint in the *SLW-Graph* of a transaction type $T = \mathrm{R}(A)\mathrm{W}(A)$. This transaction first performs a read on table A, followed by a write to the same table. In a typical 2PL execution, an instance of this transaction type would acquire a shared lock for the read and then upgrade to an exclusive lock before the write, often causing deadlocks. However, our static analysis detects this `R(A)→W(A)` access pattern replaces the locks with a single exclusive lock. As a result, with this constraint applied, each instance of this transaction type acquires an exclusive lock on the resource before both the read and write operations.

3.4 Correctness Proof

Theorem 3.1. *A set of transactions is deadlock-free if, in their* SLW-Graph*, no SLW-cycle exists.*

Proof. We will proceed by contradiction. Suppose that, despite the absence of an SLW-cycle, a deadlock occurs. Our goal is to show that this assumption leads to a contradiction, thus confirming that the transactions are deadlock-free if no SLW-cycle exists.

Let $\{T_1, T_2, \ldots, T_n\}$ be the set of transactions involved in the deadlock. Under the 2PL protocol, a deadlock implies that these transactions form a cycle (a wait-for-cycle) in the wait-for graph, which can be described as:

$$\text{wait-for-cycle} = T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \rightarrow T_1,$$

where each transaction T_i is waiting for a lock held by T_{i+1}, with $T_{n+1} = T_1$. Note that $n \geq 2$, as a single transaction cannot deadlock with itself—each transaction only locks a resource once (due to the read-write constraint).

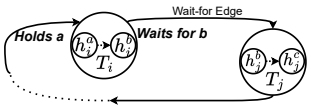
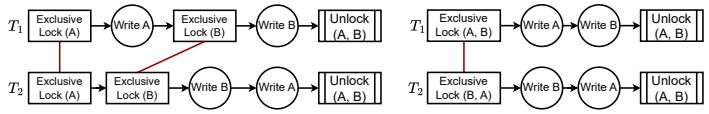


Fig. 4. Breakdown of a potential wait-for-cycle resulting in an SLW-cycle



(a) Lock movement strategy (b) Lock combination strategy

Fig. 5. Lock manipulation techniques for eliminating SLW-cycles and preventing deadlocks

In the *SLW-Graph*, each transaction T_i is represented as a sequence of hops connected by directed sibling edges (s-edges), reflecting the order of operations within the transaction. Let $h_i^{s_1}, h_i^{s_2}, \ldots, h_i^{s_n}$ be hops of T_i, connected by s-edges:

$$h_i^{s_1} \xrightarrow{\text{s-edge}} h_i^{s_2} \xrightarrow{\text{s-edge}} \ldots \xrightarrow{\text{s-edge}} h_i^{s_n}.$$

We can construct an SLW-cycle using the transactions involved in the deadlock as follows:

(1) Each wait-for edge in the wait-for-cycle corresponds to a w-edge in the *SLW-Graph*. In other words, if a wait-for edge exists between Transactions T_i and T_j, then a **w-edge** exists between these transactions, connecting two locking hops (h_i^b and h_j^b) that conflict on the same resource, with h_i^b waiting for a lock held by h_j^b.

(2) For each vertex T_i in the wait-for-cycle, there exist two distinct locking hops h_i^a and h_i^b, connected by a directed **s-edge** (and possibly intermediate hops), where h_i^a occurs before h_i^b. The hop h_i^a exists because a transaction is waiting for T_i, and h_i^b exists as T_i is waiting for the next transaction in the wait-for-cycle. Additionally, due to the existence of h_i^a and h_i^b, w-edges are always separated by s-edges.

Thus, an SLW-cycle is formed using this configuration, containing at least one **s-edge** (from step 2) and at least one **w-edge** (from step 1) connected by hops that are locking nodes (V_L). It is important to note that this cycle is directed, as each s-edge consistently leads from the hop that acquired the lock to the hop that is waiting for the lock. Additionally, in this cycle, no two w-edges are adjacent as they are always separated by s-edges (step 2).

This contradicts the initial assumption that no SLW-cycle exists in the *SLW-Graph*. Therefore, the assumption must be false. Hence, if no SLW-cycle exists in the *SLW-Graph*, the set of transactions is deadlock-free under the 2PL protocol. Figure 4 shows a breakdown of a wait-for-cycle leading to an SLW-cycle in the *SLW-Graph*. □

3.5 Lock Manipulation Techniques

In 2PL, a lock must be acquired before its corresponding operation to ensure the serializability of transactions. However, there are no restrictions on the overall order of locks. *Brook-2PL* utilizes this flexibility to eliminate deadlocks. This section discusses the lock manipulation techniques that *Brook-2PL* employs in its static analysis to eliminate potential deadlocks represented by an SLW-cycle. This is achieved by repositioning certain locking nodes earlier in the transaction to create an ordered sequence of locks, merging locking nodes into atomic locking nodes, and annotating commutativity.

Moving Lock Nodes Earlier. An effective deadlock elimination technique involves repositioning locking nodes earlier in the transaction sequence. Adjusting the timing of lock operations creates an ordered sequence for acquiring locks, transforming potential SLW-cycles into undirected cycles. Unlike SLW-cycles, undirected cycles do not lead to deadlocks.

Figure 5a demonstrates the removal of an SLW-cycle by moving a lock in T_2 earlier, thereby creating an ordered sequence of locks for resources A and B. Even though there is a cycle in this *SLW-Graph*, it is not an SLW-cycle and will not lead to a deadlock.

Following the natural ordering of tables in the data model, particularly based on how foreign keys are defined in the relationships between tables, creates a consistent locking order for future transactions, ensuring that new transactions can also follow this ordering. However, there is no strict constraint on how locks should be ordered. As long as the SLW-cycle is eliminated, deadlocks will not occur during execution. In our static analysis, *Brook-2PL* explores various locking orders and selects the one that minimizes contention across transactions.

Combining Nodes. One strategy involves combining multiple nodes in the *SLW-Graph*. This approach reduces the complexity of the graph by merging nodes, which leads to the removal of s-edges. By decreasing the number of s-edges, we eliminate the formation of SLW-cycles. This method is particularly useful when lock acquisition of an operation is related to a previous operation, and it is not feasible to move any of the locks earlier. The atomic locking operation can be implemented using a secondary lock. Figure 5b illustrates a possible combination of exclusive locks in transactions T_1 and T_2. To eliminate the sibling edge in the SLW-cycle, the locking of resource B in T_1 and resource A in T_2 are combined into an atomic locking node. This results in the removal of four sibling edges and resolves the cycle.

Brook-2PL combines all locks on a table into a single locking node in the *SLW-Graph* for each transaction. Moving locks establishes a global deadlock-free locking order across tables, while atomic locks eliminate deadlocks within a single table.

Commutativity. A w-edge between two resources can be removed if the resulting operations of those locks are known to be commutative. For instance, in many cases, two insert/delete operations are commutative, allowing the corresponding w-edge between these operations to be safely removed. Additionally, with application-level knowledge, certain operations can be explicitly annotated as commutative. This prevents the addition of a w-edge between their locking nodes, as they are non-conflicting.

3.6 Partial Transaction Chopping

By releasing locks early, the level of parallelization increases. Knowing that transactions will not abort due to deadlocks enables other transactions to read resources written by uncommitted transactions, providing early write visibility. This parallelism is particularly beneficial for high-contention workloads where read-only transactions conflict with other transactions.

Releasing locks early for a resource divides a transaction into subtransactions. To maintain serializability, an SC-Graph must be created from these new subtransactions following lock release. This SC-Graph aligns with the transaction chopping concepts discussed in Section 2. However, traditional transaction chopping requires each sub-transaction to be fully **independent**—acquiring all locks at the start and releasing them upon completion—which often leads to SC-cycles and limits practicality. Our partial chopping relaxes this by allowing locks to be held across sub-transaction boundaries: a lock acquired early can be reused and released later in another sub-transaction. To enable this, we (1) redefine C-edges in the SC-Graph to connect conflicting *locking nodes* (instead of operations), even if the operations occur across sub-transactions; (2) redefine sub-transactions by splitting at each unlock node. This approach overcomes the inherent limitations of transaction chopping by providing a more flexible chopping mechanism without requiring alterations to the ordering of transaction operations.

In the following section, we discuss our approach for early lock release. We present our method for generating the SC-Graph, which enables conflict-serializable execution of subtransactions

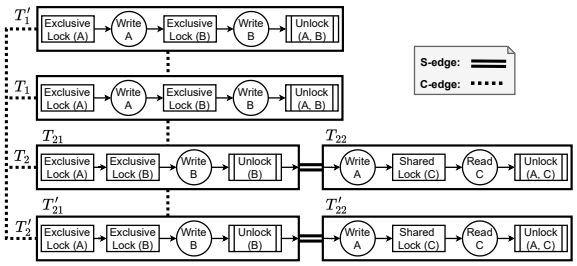


Fig. 6. The resulting SC-graph after chopping T_2 into two sub-transactions: $T_{21} = W(B)$ and $T_{22} = W(A)R(C)$

and differs from conventional SC-Graph generation in Transaction Chopping. We then prove the correctness of our method in ensuring serializability. For clarity, we retain the terminology of transaction chopping with some modifications in the SC-Graph generation.

SC-Graph Generation. The SC-Graph is generated by chopping the transaction at each release of a lock, splitting the transaction into smaller sub-transactions based on unlock nodes. Subtransactions are connected using sibling edges (S-edge), and a subtransaction is connected to another with a conflict edge (C-edge) if they both have a conflicting locking node. As demonstrated in Section 3.6, conflicts between locking nodes are sufficient to ensure correctness, even if the operations occur in separate subtransactions.

To model potential self-conflicts, the SC-Graph includes two instances of each transaction type. This is necessary because, during execution, concurrent instances of the same transaction chain may access conflicting resources. Including two instances ensures that the SC-Graph captures these self-conflicts.

Example. Assume the system has the two transactions $T_1 = W(A)W(B)$ and $T_2 = W(B)W(A)R(C)$, and we have already analyzed the transactions using *SLW-Graph*. Following the analysis, we decided to move the lock operation for resource A earlier, before all locking nodes. By moving the lock earlier, we guarantee that no deadlock will occur during execution.

In the next step, to improve concurrency, we decide to release the lock on resource B immediately after the write operation is completed, allowing other transactions to read its value. Figure 6 illustrates the resulting SC-graph after chopping T_2 into two sub-transactions: T_{21} and T_{22}. Note that in this graph, two instances of each sub-transaction are included, as instances of the same chain might conflict. In this SC-graph, no SC-cycle exists, ensuring that the corresponding chopping remains serializable.

Correctness. *Brook-2PL* enforces a specific ordering of operations through locks, meaning that subtransactions consistently follow the same order for conflicting locking nodes rather than the order of operations. This ensures that, in the Serialization Graph (SG), transactions (subtransactions) respect the direction of conflicts between locks rather than the sequence of operations. As a result, the SG between subtransactions is constructed based on resource locking rather than on individual operations. Using this definition of SG, we will prove the following by contradiction.

Theorem 3.2. *Any execution of chopping where the SC-graph contains no SC-cycles yields an acyclic serialization graph on the given transaction instances $T_1, T_2, \ldots, T_n$ and, therefore, is equivalent to a serial execution of committed transaction instances.*

Proof. We proceed by contradiction. Consider an execution of chopping where the SC-graph contains no SC-cycle of $T_1, T_2, \ldots, T_n$. Suppose that there exists a cycle in the serialization graph of $T_1, T_2, \ldots, T_n$ resulting from this execution. That is, a cycle such as $T_l \rightarrow T_j \rightarrow \cdots \rightarrow T_l$. Identify

the subtransaction of the chopping associated with each transaction instance involved in this cycle: $ST_1 \rightarrow ST_2, \cdots \rightarrow ST_n$, where both ST_1 and ST_n belong to the same transaction instance T_l.

(1) Sub transactions ST_1 and ST_n cannot be identical since each piece adheres to two-phase locking by the execution rules, and it is a known fact that the serialization graph of a set of committed two-phase locked transaction instances is acyclic. Since ST_1 and ST_n are distinct pieces within the same transaction instance T_l, there is an S-edge between them in the chopping graph.
(2) Every directed edge in the serialization graph cycle corresponds to a C-edge in the SC-graph, reflecting a conflict. Therefore, the cycle in the serialization graph implies the existence of an SC-cycle in the chopping graph, which contradicts our assumption of SC-acyclic execution.

Thus, the execution of chopping with no SC-cycle in their SC-Graph guarantees conflict-serializability, making it equivalent to a serial execution of the transaction instances.

□

Self-Conflicts. Self-conflicts may arise among subtransactions, leading to the creation of an SC-cycle between a transaction and a second instance of itself. An approach to eliminate self-conflict edges in later subtransactions is to move the conflicting locks to an earlier point, specifically to the first subtransaction. This adjustment removes the SC-cycle by preventing conflicts in later stages. In *Brook-2PL*'s static analysis, we consistently move conflicting locks earlier to eliminate self-conflicting SC-cycles.

3.7 Static Analysis

To statically analyze transactions, we propose an algorithm that takes a set of input transactions and produces a new set of transactions with modified lock ordering, while preserving the original operation ordering. The transformed transactions adhere to an SLW-cycle-free *SLW-Graph* and are further optimized using partial transaction chopping to release locks as early as possible. The algorithm systematically explores all feasible lock acquisition strategies and selects the best one based on a contention ranking. The goal is to minimize contention among transactions, which requires a quantitative metric to evaluate and compare different strategies.

Contention Score. To compare different *SLW-Graph*s of transactions, we define a contention score that measures the locking contention caused by each transaction. The contention score is computed based on how long (how many operations) locks are held within a transaction, reflecting the potential blocking time for other transactions. To account for the importance of each lock, we assign higher weights to locks on smaller tables, as they are considered more contentious. We define the contention score as shown in Equation (1), where higher contention scores indicate higher levels of contention within a given set of transactions.

$$\text{Contention Score} = \sum_{\text{chain} \in \textit{SLW-Graph}} \left\lfloor \sum_{v \in V_L(\text{chain})} \frac{\text{contention}(v)}{|\text{table}(v)|} \right\rfloor \tag{1}$$

Equation (1) computes a contention score for the entire graph by summing, for each chain (representing a transaction) in the *SLW-Graph*, the contention levels of all locking nodes. The inner summation iterates over all locking nodes $v \in V_L(\text{chain})$, where $V_L(\text{chain})$ denotes the set of locking nodes in the given chain. The function $\text{contention}(v)$ calculates the number of operations between the locking node v and its corresponding unlocking node within the same transaction, implying that maintaining locks for longer periods increases the contention score. This value is then divided by $|\text{table}(v)|$, the population (i.e., number of records) of the table associated with v, accounting for the relative importance of each locking node. Tables with smaller populations create higher contention, thus increasing the overall contention score.

Static Analysis Algorithm. Algorithm 1 provides an overview of *Brook-2PL* static analysis of transactions.

Algorithm 1 Brook2PL Static Analysis

1: **Input:** Set of transactions $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$
2: **Output:** Deadlock-free transaction set $\mathcal{T}' = \{T'_1, T'_2, \ldots, T'_n\}$
3: **procedure** BROOK2PL-ANALYZE($\mathcal{T}$)
4: $G_{\text{init}} \leftarrow$ BuildInitialSLWGraph($\mathcal{T}$) ▷ Build initial *SLW-Graph* including read-write constraints and commutativity—may contain SLW-cycles
5: $\mathcal{G}_{\text{deadlock-free}} \leftarrow$ GenerateCycleFree(G_{init}) ▷ Enumerate all SLW-acyclic graphs by reordering conflicting locking nodes using backtracking
6: $\mathcal{G}_{\text{partially-chopped}} \leftarrow$ GreedyEarlyLockRelease($\mathcal{G}_{\text{deadlock-free}}$) ▷ Greedily release locks earlier in *SLW-Graph*
7: $G_{\text{Brook2PL}} \leftarrow$ SelectBestGraph($\mathcal{G}_{\text{partially-chopped}}$) ▷ Select the best *SLW-Graph* based on the contention score
8: $\mathcal{T}' \leftarrow G_{\text{Brook2PL}}$.ExtractTransactions()
9: **return** $\mathcal{T}'$
10: **end procedure**

First, Algorithm 1 constructs an initial *SLW-Graph* of the input set of transactions in Line 4 (G_{init}). This *SLW-Graph* follows the representation discussed in Section 3.3, representing locking and operations as nodes and relations as edges. It also incorporates the read-write constraint of *SLW-Graph* and annotated commutativity to remove unnecessary w-edges. Note that the initial *SLW-Graph* of transactions might not be cycle free, thus leading to a potential deadlock during the execution.

Subsequently, *Brook-2PL* employs a backtracking procedure in Line 5 to generate all feasible SLW-acyclic graphs by reordering conflicting locks earlier ($\mathcal{G}_{\text{deadlock-free}}$). This procedure works as follows: for each chain in *SLW-Graph* representing a transaction, the locking nodes are moved earlier in every possible way, generating all possible orderings of locking nodes within the chain. These combinations from all chains are then merged using the Cartesian product to generate every possible *SLW-Graph* that includes all feasible lock combinations. Note that only locks that conflict between chains (which might create a deadlock) are moved. The state space is also pruned by leveraging dependencies between operations: some operations (and their associated locks) depend on earlier operations in the chain due to transaction logic, and locking nodes cannot be moved earlier than their corresponding operations. Finally, only SLW-acyclic *SLW-Graph*s are returned as the set $\mathcal{G}_{\text{deadlock-free}}$.

After exploring all possible *SLW-Graph*s, Line 6 applies a greedy algorithm to release locks earlier and partially chop transactions, resulting in the set $\mathcal{G}_{\text{partially-chopped}}$. In each iteration, the algorithm moves an unlock node one position earlier in the transaction chain. During this process, we verify conditions for correctly partially chopping a transaction, as outlined in Section 3.6. The greedy algorithm also attempts to optimize the graph based on the contention score in Equation 1. If moving the unlock node one position earlier does not reduce this score, we stop moving the unlock node.

Finally, *Brook-2PL* selects the best *SLW-Graph* with the lowest contention score (Line 7) and reconstructs the new deadlock-free transactions with modified lock ordering (Line 8) as the output. *Brook-2PL* handles transactions with conditional logic by building a "supergraph" containing all possible execution paths (chains) of a transaction. The lock reordering algorithm then searches for all possible lock acquisition orders that are deadlock-free across all paths. If a deadlock-free ordering for a transaction cannot be found, that transaction type is designated as "dynamic" and handled according to Section 3.8.

Time Complexity. The overall time complexity of Algorithm 1 is dominated by the backtracking step (Line 5), which explores all valid reorderings of conflicting locking nodes across transactions. For each transaction T_i with k_i movable locks, there are up to $k_i!$ valid reorderings, leading to a worst-case complexity of $O\left(\prod_{i=1}^{n} k_i!\right)$ for generating and validating all SLW-acyclic *SLW-Graph*s (where n is the total number of all transactions). The subsequent greedy optimization (Line 6) attempts to move each unlock node earlier, taking up to $O(g \cdot u)$ time, where g is the number of generated acyclic *SLW-Graph*s and u is the number of unlocks per *SLW-Graph*. Finally, the *SLW-Graph* selection contributes $O(g \cdot \log g)$ time as we sort *SLW-Graph*s based on contention score. In practice, the search space is significantly reduced through pruning based on dependency constraints, making the approach tractable for typical workloads; moreover, the algorithm is highly parallelizable.

Example. We provide a concrete example of how transactions are converted into *SLW-Graph* and how static analysis, reorder, and combine locks to provide a deadlock-free version in Section 4.1.

3.8 Dynamic Transactions

To handle dynamic transactions, *Brook-2PL* categorizes transactions into two sets: the *Static* set, which includes contentious transactions that have been statically analyzed and are guaranteed never to abort, and the *Dynamic* set, which includes dynamic transactions that may abort in case of a conflict. Our static analysis guarantees that transactions within the *Static* set cannot cause deadlocks among themselves. However, conflicting transactions involving the *Dynamic* set—either internally within the *Dynamic* set or between the *Dynamic* and *Static* sets—may still result in deadlocks.

In *Brook-2PL*, we enable the optional execution of dynamic transactions by permitting aborts exclusively for those in the *Dynamic* set. Our design assumes contentious transactions will be statically analyzed and included in the *Static* set. Consequently, introducing aborts only for dynamic transactions does not significantly increase abort rates, thereby avoiding degradation of system performance.

If dynamic transactions are enabled, we apply the following deadlock prevention protocol. When a transaction T_i requests a lock currently held by transaction T_j; depending on the sets to which T_i and T_j belong, we follow one of these rules:

- **Both** $T_i \in Static$ **and** $T_j \in Static$**:**
  - No deadlock prevention is needed since this scenario does not lead to deadlocks.
- **Both** $T_i \in Dynamic$ **and** $T_j \in Dynamic$**:**
  - If T_i is older than T_j, abort T_j; otherwise, T_i waits.
- $T_i \in Dynamic$ **and** $T_j \in Static$**:**
  - If T_i is older than T_j, abort T_i; otherwise, T_i waits.
- $T_i \in Static$ **and** $T_j \in Dynamic$**:**
  - If T_i is older than T_j, abort T_j; otherwise, T_i waits.

This protocol ensures a consistent timestamp-based ordering among all transactions. Note that *Brook-2PL* falls back to Wound-Wait 2PL when only dynamic transactions are used. However, by statically analyzing transactions, we assign them higher priority over dynamic transactions in case of a conflict.

Additionally, *Brook-2PL* maintains correctness in case of uncommitted data reads when dynamic transactions are enabled. This is because static transactions are guaranteed not to have concurrency-control-related aborts. If a dynamic transaction reads uncommitted data from a static transaction and subsequently aborts, it will simply roll back its own changes with no effect on the static transaction, preserving correctness.

3.9 Limitations

Brook-2PL imposes certain limitations on the system, which we discuss in this section.

Preprocessing. Similar to other preprocessing methods, our approach introduces some limitations due to the requirement of a preprocessing phase for transaction analysis. However, static analysis is performed offline as an optimization and is only necessary when new contentious transaction types are introduced to the system. It is not required during execution. *Brook-2PL*'s dynamic transaction handling approach is designed to support all workloads without the need for continuous static analysis. *Brook-2PL* prevents deadlocks in statically analyzed transactions. Deadlocks in dynamic transactions are unpredictable, but *Brook-2PL* handles them to ensure the system remains correct.

Brook-2PL targets workloads that require prior knowledge of static transaction types. This aligns with real-world systems where many transactions are implemented as stored procedures or can be converted into static transactions with minimal changes [43].

User and Programmatic Aborts. The early release of locks restricts the ability to perform aborts after locks are released in a static transaction. Once a lock on a resource is released, the resource may be read by other concurrent transactions, making it impossible to abort the previous operation without affecting serializability. In *Brook-2PL*, user-initiated aborts are only permitted before any locks are released in the static transaction. If a user wishes to abort a transaction after release of locks, they must provide a compensating transaction. For workloads with frequent user-initiated or data-dependent aborts, *Brook-2PL* (and any protocol with early write visibility) faces limitations.

However, if user aborts are necessary, besides disabling the early release of locks, a similar approach to previous works [23] can be implemented by tracking dirty reads and performing cascading aborts in cases of inconsistency, to benefit from early release of locks. For workloads with a low rate of user aborts, this method will not be inefficient or impose a significant overhead on *Brook-2PL*. In all cases, *Brook-2PL* guarantees deadlock-freedom. We omit the details of this method, as it is beyond the scope of this paper.

4 Evaluation

To evaluate *Brook-2PL*, we perform several experiments comparing its performance against the state-of-the-art concurrency control protocol Bamboo [23] as well as a basic 2PL protocol. The transactions used for evaluation fall into two categories. First, we use a synthetic transactional workload that simulates an online store for buying and selling game items. This workload is designed to highlight various aspects of *Brook-2PL* and to walk through the process of statically analyzing transactions. Second, we evaluate *Brook-2PL* using the TPC-C [57] benchmark under high-contention conditions, following the setup of prior works [23, 46, 50].

In the rest of this section, we first discuss the two workloads and their transactions used in our experiments and how *Brook-2PL* statically analyzes them. Next, we describe the implementation of the transaction layer and our experimental setup. Finally, we present and analyze the results of various experiments focusing on transaction performance and tail latency.

4.1 Online Game Store Workload

The store workload represents an online store for game items. The design of the data tables in this online game store includes three tables: ***Players***, ***Items***, and ***Listings***. The **Players** table contains information about individual players, including `PId` (a primary key), `name` (the player's name), and `cash` (the player's available balance). The **Items** table represents game items, with each record containing `IId` (a primary key), `name` (the item name), and `owner` (a foreign key referencing `PId` in the Players table to denote ownership). Finally, the **Listings** table tracks items available for

```
1 Transaction AddListing(PId, IId, price):
2 item = Read Item(IId)  // Check item exists and verify its owner
3 player = Read Player(PId)  // Check player exists
4 Insert Listing(IId, price)
```

Transaction 1. Add Listing Transaction

```
1 Transaction BuyListing(PId, LId):
2 listing = Read(Listings, LId)  // Check listing exists
3 buyer = Read(Players, PId) // Check player & funds
4 item = Read(Items, listing.item) // Retrieve Item ID from Listing and verify item existence
5 owner = Read(Players, item.owner)  // Check seller exists
6 Delete(Listings, LId)  // Remove the listing after checks
7 Write(Items, listing.item)  // Transfer ownership
8 Write(Players, PId) // Deduct buyer
9 Write(Players, item.owner)  // Credit seller
```

Transaction 2. Buy Listing Transaction

```
1 Transaction ReadItems(items):
2 for IId in items:
3     Read Item(IId)
```

Transaction 3. Read Items Transaction

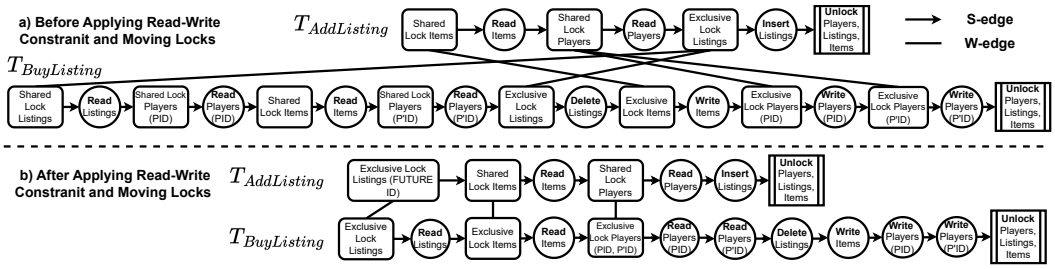


Fig. 7. *SLW-Graph* representation of the store workload: (a) *SLW-Graph* without applying the read-write constraint or moving locks earlier; (b) *SLW-Graph* after applying the read-write constraint and moving locks earlier to ensure an SLW-cycle-free *SLW-Graph*. This execution will be utilized in *Brook-2PL*.

sale, including `LId` (a primary key), `item` (a foreign key referencing `IId` from the Items table), and `price` (the listed sale price).

Transactions. Three transactions are implemented in this online game store. The first transaction, represented in Transaction 1, shows how a player lists an item for sale with a specific price. In this transaction, after verifying the ownership of the item (line 2) and ensuring the existence of the player (line 3), a new listing is inserted with the desired price.

The second transaction involves buying an already listed item, as shown in Transaction 2. In this transaction, we first perform initial checks to verify the existence of the listing and the player and ensure that the buyer has sufficient funds for the purchase (Lines 2–5). Next, the listing is deleted (Line 6), and the money is transferred from the buyer to the seller (Lines 7–9).

The third transaction, shown in Transaction 3, is a read-only transaction that retrieves a set of requested items. This transaction represents the potential read-only transactions that might be executed in a highly contentious system. Read-only transactions are significant because reads dominate real-world workloads [8, 38].

Static Analysis. We implement the static analysis algorithm shown in Algorithm 1 to parse transactions and produce deadlock-free transactions. Figure 7 shows the *SLW-Graph* representation for transactions in the store workload. In Figure 7a, the *SLW-Graph* is shown prior to applying the read-write constraint or moving any locks earlier—in other words, before *Brook-2PL* static analysis.

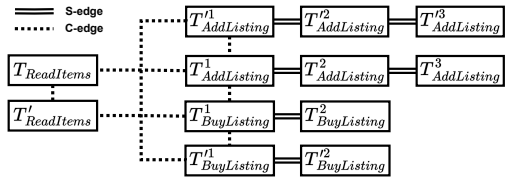


Fig. 8. SC-Graph representation of the store workload after the early release of locks

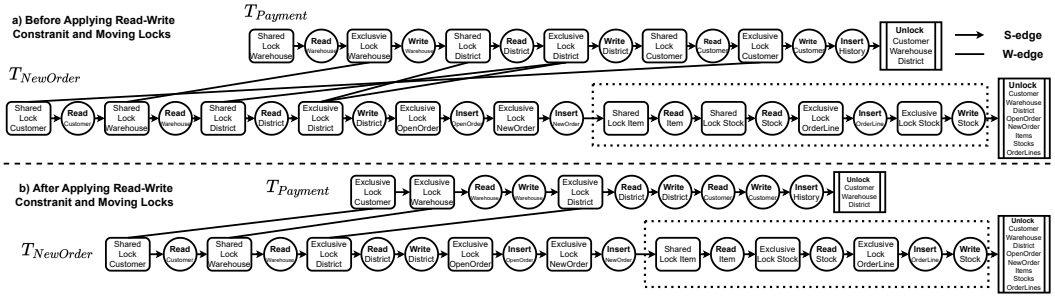


Fig. 9. *SLW-Graph* representation of TPC-C workload transactions: (a) *SLW-Graph* without applying the read-write constraint or moving locks earlier; (b) *SLW-Graph* after applying the read-write constraint and moving locks earlier.

Figure 7b shows the *SLW-Graph* after applying the read-write constraint and performing a lock movement. The read-write constraint reduces the locks used for Listings, Players, and Items into exclusive locks. Subsequently, by moving the Listings lock in the *Add Listing* transaction to the beginning of the transaction, the SLW-cycle is eliminated, ensuring the deadlock-free execution of these transactions. It is important to note that the *Read Items* transaction is not shown in this figure, as the inclusion of a read-only transaction chain does not create any SLW-cycle in the *SLW-Graph*.

Partial Transaction Chopping. Our static analysis algorithm improves concurrency by releasing the lock on *Items* immediately after the final write operation in the *Buy Listing* transaction. Additionally, it releases the locks on *Items* and *Players*, respectively, after the lock and read operations on *Players* in the *Add Listing* transaction. The corresponding SC-Graph for this partial transaction chopping is shown in Figure 8. In this chopping, the *Buy Listing* transaction is divided into two subtransactions, and the *Add Listing* transaction into three. This SC-Graph does not contain any SC-cycles, ensuring the serializability of the final execution.

4.2 TPC-C Workload

The TPC-C benchmark [57] is widely used to evaluate concurrency control protocols under high-contention workloads. It simulates a wholesale supplier managing orders and payments across multiple warehouses in a dynamic and complex database environment.

Following previous works [23, 46, 50], we restrict our evaluation to TPC-C's *New Order* and *Payment* transactions, as they constitute the majority of the benchmark (approximately 45% and 43%, respectively). Additionally, these transactions are short update transactions that place greater stress on concurrency control. Therefore, in our evaluations, we conducted experiments with a workload comprising 50% *New Order* transactions and 50% *Payment* transactions. In accordance with the benchmark specifications, 1% of these transactions were aborted to simulate user aborts. In this benchmark, warehouses serve as hotspots for the workload, and contention increases as the number of warehouses decreases.

Static Analysis. Figure 9 illustrates the *SLW-Graph* representation for *New Order* and *Payment* transactions. After applying the read-write constraint to the *SLW-Graph*, potential deadlocks are eliminated by moving the Customer lock earlier in the *Payment* transaction.

Partial Transaction Chopping. *Brook-2PL*'s static analysis releases the locks on *District* and *Warehouse* after the last write on *District* in the *Payment* transaction, resulting in two subtransactions. Similarly, in the *New Order* transaction, the locks on *Customer*, *District*, and *Warehouse* are released after the last write on *District*, by partially chopping the transaction into two subtransactions. To remove self-conflicts between the second subtransaction of the *New Order* transaction, the *Stock* lock is moved to the first subtransaction. Our partial transaction chopping method, unlike traditional transaction chopping, enables the creation of an SC-acyclic SC-Graph for TPC-C transactions, thereby reducing contention, while ensuring that the final execution is serializable.

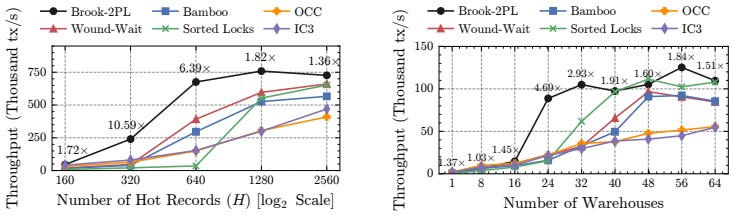
4.3 Implementation and Setup

System Implementation. *Brook-2PL* static analysis (Algorithm 1) is implemented in Java. This implementation parses the stored-procedure transactions—using the transaction representation language used in Section 4.1—and produces deadlock-free transactions as a preprocessing step. In our experiments, transactions, including their locks and operations, are written and executed as one-shot stored procedures, a common practice to evaluate concurrency control under high contention [3, 23, 46, 50]. However, this does not preclude *Brook-2PL* from handling interactive transactions, as dynamic transactions are supported.

Our 2PL implementation in Java uses a hash table as the lock table to store Lock objects. We also implement an in-memory database with Write-Ahead Logging (WAL), using a hash table to store the data. To enhance scalability, we use Java's ConcurrentHashMap [44], which employs per-bucket locking that is highly optimized for concurrent execution. Additionally, our implementation acquires fine-grained (row-level) logical locks on records. Consequently, latch contention only occurs when multiple threads simultaneously acquire or release the logical locks of the same record. Our implementation follows a basic 2PL that does not require conflict handling for static transactions, as they are designed to be deadlock-free. However, if dynamic transactions are enabled, conflicts are managed as described in Section 3.8.

Comparison Baseline. We implement wound-wait 2PL [6], Bamboo [23], optimistic concurrency control (OCC) [33], IC3 [62], and a basic lock sorting approach as comparison baselines.

- **Wound-Wait:** The wound-wait variant of the 2PL protocol is a deadlock prevention mechanism. This mechanism ensures deadlock prevention while prioritizing older transactions.
- **Bamboo:** Bamboo is a state-of-the-art 2PL protocol that allows transactions to read dirty data during the execution phase (thereby violating 2PL) while still ensuring serializability. This protocol, which is a variant of wound-wait 2PL, permits a transaction to release its lock on a record after its last update, enabling other transactions to access the data. To enforce serializability, Bamboo tracks the dependencies of dirty reads and cascadingly aborts transactions when these dependencies are violated. Bamboo includes a latch-reduction optimization unrelated to concurrency control. Our implementation omits this to ensure a fair comparison focused solely on concurrency control logic.
- **Sorted Locks:** This 2PL baseline assumes known read/write sets (unlike *Brook-2PL*) and acquires locks in sorted order at transaction start, ensuring deadlock-free execution.
- **OCC:** Optimistic concurrency control allows transactions to execute without locking resources, assuming conflicts are rare, and checks for conflicts before committing.



(a) Store Wrokload (b) TPC-C Workload

Fig. 10. Performance comparison while varying the number of hot records with $P_{HotRecord}$ = 100%

- **IC3:** IC3 is a state-of-the-art transaction chopping method that statically analyzes transactions and executes sub-transactions using OCC. It provides an approach to execute deadlock-prone SC-Cycles in SC-Graphs. We statically analyze our workloads using this method. For a fair comparison, we apply our definition of conflicting operations in the static analysis: two operations conflict if they access the same row of a table, regardless of the column.

Experimental Setup. The experiments are conducted on *cascadelake* node of Chameleon Cloud [30]. This machine has 2 *Intel Xeon Gold 6242 CPUs* at *2.8 GHz* with *96* virtual threads and is also equipped with 187 GB RAM and 10 Gigabit Ethernet.

The database size for our store workload is initialized with 500, 000 players in *Players* table, each owning 5 items, resulting in an *Items* table containing 2, 500, 000 rows. Additionally, the initial database includes 100, 000 pre-existing listings in the *Listings* table. We vary the database size in the TPC-C benchmark by increasing the number of warehouses.

To execute transactions, they are dispatched from a benchmark layer using 64 threads. Each benchmark runs for 60 seconds, excluding the warm-up period. In the event of a transaction abort, the transaction is immediately retried with its initial Timestamp ID.

Controlling Contention. To control the level of contention in our experiments, we use two metrics:

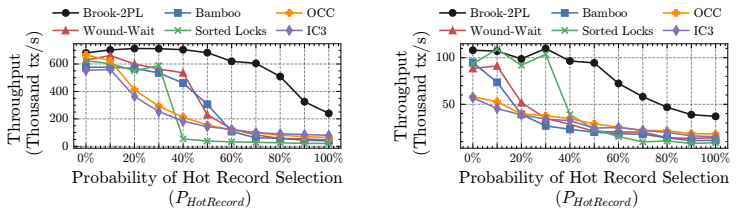
- *Number of Hot records (H)* specifies the number of records in the hotspot. The hotspot records are selected from the tables frequently accessed by all transactions. For the store workload, H represents the number of Items and their associated Players chosen as hotspots. In the TPC-C workload, H warehouses are selected as hotspot records.
- *Probability of hot record selection ($P_{HotRecord}$)* defines the likelihood of selecting a hotspot record for the executing transaction. For instance, if the probability of hot record selection is 10%, there is a 10% chance that the transaction's inputs will be chosen from hotspot records.

These metrics create a controllable contention profile that allows us to systematically vary contention—unlike prior work, which often relies on fixed contention models [23, 62].

4.4 Performance Comparison

Varying Number of Hot Records (H). We conducted experiments on the store workload by varying the number of hot records (H) to control the level of contention. In this experiment, the probability of selecting a hot record ($P_{HotRecord}$) is set to 100%. Figure 10a illustrates the performance of the system using different 2PL protocols to execute two transactions: *Add Listing Transaction* (Algorithm 1) and *Buy Listing Transaction* (Algorithm 2) in equal proportions. Across all hotspot configurations, *Brook-2PL* consistently outperforms competing protocols. *Brook-2PL* achieves an average speed-up in transaction processing of 4.3× compared to others.

Varying Number of Warehouses. In this experiment on the TPC-C workload, all warehouses are treated as hot records ($P_{HotRecord}$ = 100%), and the contention is increased by varying the



(a) Store Workload ($H = 320$) (b) TPC-C Workload ($H = 16$)

Fig. 11. Performance comparison while varying the probability of hot record selection with a fixed number of hot records

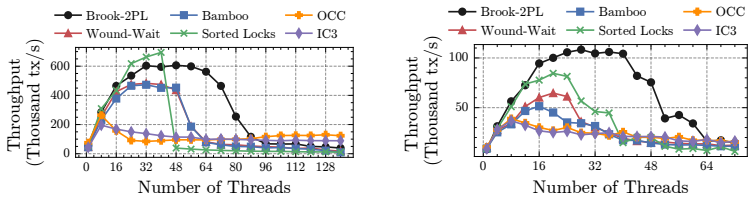
number of warehouses (H). Figure 10b shows that under high contention (fewer warehouses—$H \leq 16$), all methods experience low performance, with *Brook-2PL* outperforming both Bamboo and Wound-Wait. As the number of warehouses exceeds 24, *Brook-2PL* approaches the system's maximum capacity, continuing to outperform other retrial-based approaches, as many of their transactions are aborted. For workloads with $H \geq 40$ and lower contention, our approach performs comparably to Sorted Locks. This figure highlights the average speed-up of *Brook-2PL* over Bamboo and Wound-Wait across different warehouse counts.

Varying Probability of Hot Record Selection ($P_{HotRecord}$). This experiment adjusts the contention of the workload by varying $P_{HotRecord}$ while keeping the number of hot records (H) fixed. Figure 11a shows the performance of comparing 2PL protocols as $P_{HotRecord}$ varies from 0% (no contention) to 100% (all records are selected from hot records) for the *Add Listing* and *Buy Listing* transactions. In this experiment, 320 *Item* records and their corresponding *Players* are selected as hotspot records.

The results show that when there is no contention, all protocols perform similarly. Bamboo performs slightly worse than Wound-Wait in this scenario due to its additional overhead of tracking dirty reads. IC3 does not provide any performance gain over OCC, as its static analysis of the Store workload cannot chop transactions into multiple hops due to deadlocks (each transaction is combined into a single hop). Additionally, IC3 incurs overhead from writing to a local stash for each hop and introduces extra waiting, making it less efficient. As contention increases, the performance of other protocols begins to decline, whereas *Brook-2PL* maintains stable performance under lower levels of contention. For $P_{HotRecord} < 40\%$, the throughput of the Sorted Locks approach does not drop significantly. However, for higher contention levels, performance drops drastically due to long waiting times. A simple sorting approach can create bottlenecks under high contention by forcing all transactions to acquire the same hot locks first. Also, moving all lock acquisitions to the beginning of the transaction forces locks to be held for a longer duration, reducing concurrency.

Interestingly, when the probability of hot records is low ($P_{HotRecord} < 50\%$), *Brook-2PL* shows a slight performance improvement. This is because hot records fit within the cache, resulting in a higher cache hit rate and reduced access latency. *Brook-2PL* tolerates contention up to 50% without a performance drop. Overall, *Brook-2PL* shows less performance degradation than the comparison methods. In this experiment, *Brook-2PL* achieves an average 5.2× speed-up over the other methods.

A similar experiment is conducted on the TPC-C workload (Figure 11b). In this experiment, the database contains 64 warehouses, with 16 warehouses selected as hot records ($H = 16$). Similar to the experiments on the store workload, *Brook-2PL* outperforms the comparison methods and shows lower performance degradation. The throughput of the Sorted Locks approach declines sharply under high contention ($P_{HotRecord} > 30\%$), whereas it remains comparable to *Brook-2PL* at lower contention levels. Bamboo and Wound-Wait perform similarly under high contention,



(a) Store [$P_{HotRecord} = 70\%$, $H = 320$] (b) TPC-C [$P_{HotRecord} = 100\%$, $H = 16$]

Fig. 12. Performance comparison while varying the number of threads with fixed H and $P_{HotRecord}$

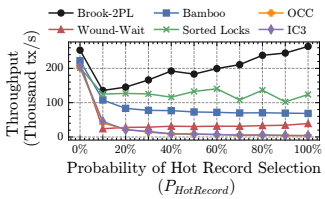
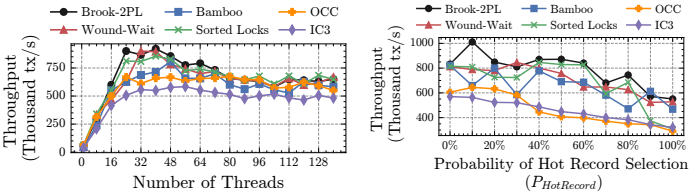


Fig. 13. Performance comparison of read-only transactions while varying the probability of hot record selection ($H = 320$)



(a) Store [$P_{HotRecord} = 0\%$, $H = 320$] (b) Store [32 Threads , $H = 320$]

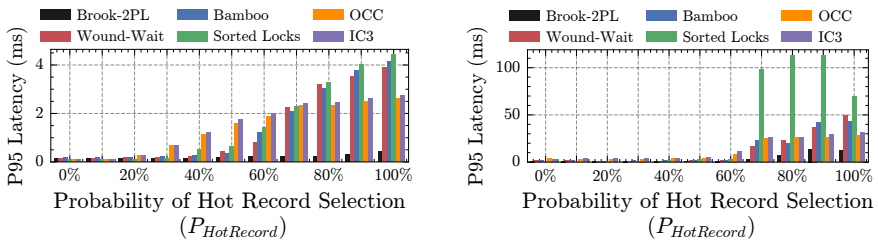
Fig. 14. Performance comparison with lower levels of contention

since Bamboo's lock retirement mechanism only retires exclusive locks and does not provide significant performance gains in the TPC-C workload. OCC and IC3 consistently underperform compared to our method across all contention levels, as many transactions abort due to conflicts. Even with $P_{HotRecord} = 0\%$, conflicts exist between transactions in the TPC-C workload. OCC-based approaches are generally unsuitable for high-contention workloads due to their high abort rates. *Brook-2PL* achieves an average speed-up of 2.86×.

Varying Number of Threads. This experiment evaluates scalability by increasing the number of threads under a high contention setting. Figure 12a shows the performance for the store workload with high contention ($P_{HotRecord} = 70\%$, $H = 320$). While all methods perform similarly at lower thread counts, Bamboo, Wound-Wait, and Sorted Locks suffer performance drops after 48 threads. OCC and IC3 fail to scale under highly contended workloads, reaching their peak performance with only 8 threads. *Brook-2PL* maintains stable and higher throughput, showcasing better scalability under high contention. Figure 12b shows results for TPC-C with high contention ($P_{HotRecord} = 100\%$, $H = 16$). As the number of threads increases, all methods initially improve in throughput, but OCC and IC3 degrade beyond 8 threads, Bamboo and Wound-Wait beyond 16 threads, and Sorted Locks after 20 threads. In contrast, *Brook-2PL* continues to scale up to 40 threads, sustaining higher throughput and maintaining scalability.

Read-Only Transactions. To evaluate the effect of early release of locks on transactions, we execute the *Read Items* transaction–which reads 20 hot items in the store workload–in 20 additional threads alongside the *Add Listing* and *Buy Listing* transactions.

Figure 13 shows the performance of read-only transactions under varying $P_{HotRecord}$. *Brook-2PL* demonstrates higher throughput than the other methods across all values of $P_{HotRecord}$. Under no contention, all methods perform similarly. However, Bamboo's and Sorted Locks throughput drops after $P_{HotRecord} = 10\%$, leveling off near 80k and 120k transactions/sec respectively beyond $P_{HotRecord} = 20\%$. In contrast, *Brook-2PL* maintains stable throughput, with at most a 53% drop in the worst case.



(a) Store Workload (H = 320) (b) TPC-C Workload (H = 16)

Fig. 15. Tail Latency comparison while varying the probability of hot record selection with a fixed H

Brook-2PL demonstrates higher throughput than the other methods across all values of $P_{HotRecord}$. Under no contention, all methods perform similarly. However, Bamboo's and Sorted Locks' throughput drops after $P_{HotRecord}$ = 10%, leveling off near 80k and 120k transactions/sec, respectively, beyond $P_{HotRecord}$ = 20%. In contrast, *Brook-2PL* maintains stable throughput, with at most a 53% drop in the worst case. *Brook-2PL*'s throughput increases under higher contention, as more hot records are cached, reducing the average data retrieval time from the in-memory database. The throughput of other methods remains close to zero under high contention, as they lack mechanisms to support long read-only transactions and suffer from high abort rates.

Choice of 64 Threads. We selected 64 threads for our experiments to (1) saturate all cores and (2) match the thread-to-warehouse ratio in TPC-C, following standard practice. Figure 13a confirms that under 0% hot record probability, all protocols scale well to 64 threads without thrashing. Figure 13b shows the performance of protocols under varying $P_{HotRecord}$ with 32 threads. While trends remain similar, performance gaps are less pronounced, supporting the need for higher thread counts to generate sufficient contention and effectively differentiate protocol performance.

4.5 Tail Latency Comparison

Figure 15 shows the tail latency (p95) of transactions as $P_{HotRecord}$ varies. Latency is measured from the time a transaction is submitted to the system until it is committed. For protocols with aborts, transactions are retried immediately after being aborted. In contrast, *Brook-2PL* experiences no aborts due to deadlocks, and each transaction is executed only once.

Figure 15a shows the p95 latency of transactions when H = 320 in the store workload. *Brook-2PL* consistently achieves lower tail latency. Bamboo suffers from cascading aborts, resulting in higher latency than Wound-Wait across most contention levels. IC3 and OCC achieve similar latency; however, IC3 has slightly higher tail latency due to additional overhead. The Sorted Locks approach achieves the lowest latency under low contention but experiences an increase in latency under high contention, ultimately performing the worst. On average, *Brook-2PL* reduces p95 latency by 56% compared to other baselines.

Figure 15b shows a similar experiment for the TPC-C workload with H = 16 out of 64 warehouses. Tail latency for other protocols increases rapidly due to high abort rates; both exhibit similar latency patterns under this setup. In contrast, *Brook-2PL* provides significantly better p95 latency as contention increases. On average, *Brook-2PL* reduces tail latency (p95) by 48% compared to other protocols in the TPC-C workload.

4.6 Execution Time Breakdown

Figure 16 shows the breakdown of CPU time—useful execution, lock wait, and wasted execution—for the *Add Listing* and *Buy Listing* transactions in the store workload as $P_{HotRecord}$ increases. Useful

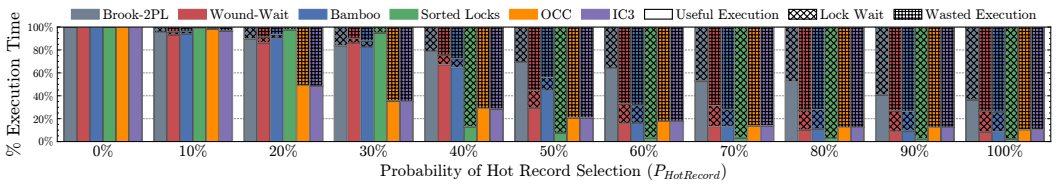
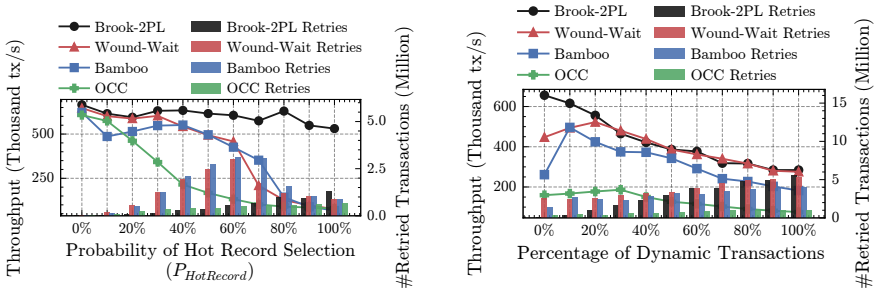


Fig. 16. CPU time breakdown–store workload (H = 320)



(a) 10% Dynamic Transction (b) $P_{HotRecord}$ = 50%

Fig. 17. Performance and Retry Analysis with Dynamic Transactions enabled in the Store Workload

execution represents the proportion of transaction execution time spent performing operations, lock wait indicates the proportion of time spent waiting for locks, and wasted execution corresponds to the proportion of transaction work wasted due to aborts.

Brook-2PL experiences no wasted execution because no transactions are aborted. As contention increases, the proportion of lock wait for *Brook-2PL* increases; however, it remains lower than the wasted execution observed in the comparison methods. The lock wait time in the Sorted Locks approach increases drastically under high contention, consuming nearly the entire execution time. Both Bamboo and Wound-Wait experience significant wasted execution in addition to lock wait. IC3 and OCC experience significantly increased wasted execution time as contention increases. In particular, on average, *Brook-2PL* spends 70% of the execution time performing useful work, while other methods spend 41%.

4.7 Dynamic Transactions

To evaluate dynamic transactions, we enable dynamic transaction support in *Brook-2PL* and introduce transactions that perform 10 read–modify–write operations. Dynamic transaction inputs are selected from database records, with each input drawn from hot records with probability $P_{HotRecord}$. IC3 and Sorted Locks are excluded as they do not support dynamic transactions.

Figure 17a shows the throughput (left axis) and the number of transactions being retried (right axis) as $P_{HotRecord}$ varies for both the store workloads when 10% transactions are dynamic and the rest are from previous static transactions. *Brook-2PL* maintains higher throughput across all contention levels. Although dynamic transactions introduce retries, *Brook-2PL* shows significantly fewer retrials. Bamboo and Wound-Wait suffer from increases in retried transactions, particularly beyond $P_{HotRecord}$ = 40%, leading to noticeable throughput degradation. Beyond $P_{HotRecord}$ = 60%, retry counts decrease not because performance improves but because excessive waiting times prevent many transactions from attempting a retry. For OCC, long dynamic transactions cause most transactions in the system to be continuously retried, significantly reducing overall performance.

In Figure 17b, we vary the share of dynamic transactions from 0% to 100% with $P_{HotRecord}$ = 50%. When only static transactions are present, throughput of *Brook-2PL* outperforms competing methods and experiences no retries, since it never aborts static transactions. Introducing 10% dynamic transactions increases workload diversity and introduces more transaction types, giving Wound-Wait and Bamboo a throughput boost. However, as the share of dynamic transactions increases higher than 10%, throughput steadily declines for all methods. *Brook-2PL* maintains its advantage up to 50% dynamic transactions, beyond which its throughput converges with Wound-Wait's. Bamboo consistently has lower throughput, because it immediately releases the lock after every write in dynamic transactions—without knowing whether a write is the last update—resulting in extra overhead. OCC experiences low throughput across all levels of dynamic transactions due to a high abort rate. *Brook-2PL* achieves an average performance improvement of 95% on static transactions and 84% on dynamic transactions. *Brook-2PL* experiences fewer transaction retries until dynamic transactions reach 60%, after which its retry count is comparable to other methods. Overall, *Brook-2PL* has 21% fewer transactions retried than Bamboo and 33% fewer than Wound-Wait. *Brook-2PL* benefits from a higher percentage of static contentious transactions. Dynamic transactions that become highly contentious within the workload are targeted for static analysis.

5 Related Work

Deadlock and Abort Freedom. Many research efforts have aimed to develop deadlock-free concurrency control protocols, often using deadlock-prevention mechanisms. They inevitably abort and retry conflicting transactions to break dependency cycles [1, 6, 23, 32, 36, 37, 41, 60, 66, 68]. However, these approaches may unnecessarily abort transactions, reducing concurrency and potentially creating a cascading burden on the system.

Deterministic systems, instead of applying concurrency control to individual transactions, process transactions in batches and execute them according to pre-generated schedules. These systems process transactions in a serial order based on their logged read and write sets, which are predetermined to generate optimal solutions. They ensure serializable execution while avoiding concurrency-control-related aborts [13, 20, 48]. QueCC [47] proposes a mechanism to avoid transaction aborts using a priority queue approach to dispatch transactions into different partitions. Similarly, Calvin and Bohm [19, 59] utilize knowledge of transaction conflicts to relax the total order into an equivalent partial order, thereby reducing transaction aborts. Related ideas are applied in other deterministic transaction systems [16, 42, 46]. The concept of analyzing transactions is extended in *Brook-2PL* through static analysis of transaction types. However, in our system, this analysis is applied to transaction types rather than batches of predefined transactions, making the scope of the application distinct from previous approaches.

ORTHRUS [50] is a database system that ensures deadlock avoidance through planned data access. ORTHRUS achieves deadlock-freedom by utilizing the lexicographical ordering of lock acquisition. In contrast, *Brook-2PL* achieves deadlock-freedom through a dynamic ordering of lock acquisition tailored to the transactions. *Brook-2PL* modifies the locking order only when necessary.

Aria [39] is an optimistic, batch-oriented system that avoids requiring a priori knowledge of transaction read/write sets by performing its analysis online after transactions have executed; conflicts are resolved in a commit phase, and transactions may abort as needed. In contrast, *Brook-2PL* performs static analysis on transaction logic before execution, without requiring runtime information, enabling deadlock-free and abort-free 2PL. Aria reorders commit order by transforming RAW to WAR dependencies to minimize aborts. Under high contention, Aria falls back to a Calvin-like protocol using the now-known read/write sets. In contrast, *Brook-2PL* reorders lock acquisitions in advance to break dependency cycles. Other transaction scheduling techniques have been proposed

to address the problem of contention, either by assigning a schedule based on arrival order [5, 12] or by reordering the schedule after transaction commit and/or abort [10, 17].

Early Write Visibility. The authors in [20] propose PWV, which breaks transactions into multiple atomic pieces and builds a dependency graph to schedule and commit transactions with finer granularity. Similar ideas are applied in various other works in the context of deterministic databases [16, 19, 46], where transactions are executed in finer-grained pieces. These protocols rely on the assumption of determinism, where transaction execution is ordered prior to execution. In contrast, *Brook-2PL* does not rely on the assumption of determinism.

Previous research has proposed enabling dirty reads of writes [23, 24, 29, 49, 52]. Authors in [49] propose an early write visibility discipline where a transaction executes with both the pre-image and after-image of reading uncommitted data. In [23], the authors propose Bamboo, which enables early reads of uncommitted writes by keeping track of dirty reads. In these approaches, the problem of cascading aborts persists, and additional runtime overhead is introduced to support early write visibility.

Transaction chopping techniques, such as those proposed in [54, 62, 63, 67], aim to decompose transactions into smaller subtransactions while preserving serializability. However, these methods often suffer from rigid conditions, making it challenging to achieve coarse-grained decomposition. Recent advancements like Lynx [67] and Salt [63] attempt to address this by leveraging application semantics to reduce conflicts among subtransactions. Despite these improvements, they lack a dynamic mechanism to relax transaction chopping conditions. IC3 [62], which introduces early write visibility for transactions through optimistic concurrency control (OCC), inherits the general limitations of transaction chopping. Additionally, IC3 and other OCC approaches [40, 61] are not suitable for high-contention workloads due to the high abort rates inherent to them. *Brook-2PL* introduces partial transaction chopping, which allows flexibility to support more workloads and targets high-contention workloads.

Hybrid Approaches. Prior work has explored combining locking and OCC. For instance, Static and Dynamic OCC [64] switch from OCC to locking after the first retry. MOCC [61] and Hsync [53] are other examples that propose hybrid 2PL–OCC protocols. ACC [56] goes further by adaptively selecting from a broader range of protocols. CCaaLF [45] learns an agent function, implemented as an efficient in-database lookup table, that maps database states to concurrency control actions. However, these methods are generally designed for dynamic workloads and do not directly address hotspots, as they typically delay making writes visible after the transaction is committed.

6 Conclusion

In this paper, we presented *Brook-2PL*, a deadlock-free 2PL protocol designed to tolerate high-contention workloads. *Brook-2PL* statically analyzes transactions using the *SLW-Graph* representation and applies lock manipulation techniques, such as moving locks earlier or combining locking nodes, to eliminate deadlocks. To further reduce contention, *Brook-2PL* introduces a novel partial transaction chopping mechanism that enables the early release of locks, enhancing transaction concurrency. Our evaluation results show that *Brook-2PL* outperforms state-of-the-art concurrency control protocols, achieving a performance improvement of 2.86× and reducing tail latency by 48% in the TPC-C workload.

7 Acknowledgement

This research is partly supported by a gift from Roblox and the NSF under grants 2321121, CNS1815212, and SaTC-2245372.

References

[1] Divyakant Agrawal, Amr El Abbadi, Richard Jeffers, and Lijing Lin. 1995. Ordered shared locks for real-time databases. *The VLDB Journal* 4 (1995), 87–126.

[2] Raja Appuswamy, Angelos C Anadiotis, Danica Porobic, Mustafa K Iman, and Anastasia Ailamaki. 2017. Analyzing the impact of system architecture on the scalability of OLTP engines for high-contention workloads. *Proceedings of the VLDB Endowment* 11, 2 (2017), 121–134.

[3] Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. 2014. Coordination avoidance in database systems. *Proceedings of the VLDB Endowment* 8, 3 (2014), 185–196.

[4] Jason Baker, Chris Bond, James C Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing scalable, highly available storage for interactive services.. In *CIDR*, Vol. 11. 223–234.

[5] Philip A Bernstein and Nathan Goodman. 1983. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)* 8, 4 (1983), 465–483.

[6] Philip A Bernstein, Vassos Hadzilacos, Nathan Goodman, et al. 1987. *Concurrency control and recovery in database systems*. Vol. 370. Addison-wesley Reading.

[7] Nathan Bronson, Abutalib Aghayev, Aleksey Charapko, and Timothy Zhu. 2021. Metastable failures in distributed systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 221–227.

[8] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. 2013. {TAO}:{Facebook's} distributed data store for the social graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 49–60.

[9] Kevin Bruhwiler, Fayzah Alshammari, Farzad Habibi, Juncheng Fang, Yinan Zhou, Abhishek Singh, Ahmad Showail, and Faisal Nawab. 2022. Analyzing soft and hard partitions of global-scale blockchain systems. In *2022 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 304–311.

[10] Matthew Burke, Florian Suri-Payer, Jeffrey Helt, Lorenzo Alvisi, and Natacha Crooks. 2023. Morty: Scaling concurrency control with re-execution. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 687–702.

[11] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.

[12] Audrey Cheng, Aaron Kabcenell, Jason Chan, Xiao Shi, Peter Bailis, Natacha Crooks, and Ion Stoica. 2024. Towards optimal transaction scheduling. *Proceedings of the VLDB Endowment* 17, 11 (2024), 2694–2707.

[13] James Cowling and Barbara Liskov. 2012. Granola:{Low-Overhead} distributed transaction coordination. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 223–235.

[14] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1243–1254.

[15] Bailu Ding, Lucja Kot, and Johannes Gehrke. 2018. Improving optimistic concurrency control through transaction batching and operation reordering. *Proceedings of the VLDB Endowment* 12, 2 (2018), 169–182.

[16] Bailu Ding, Lucja Kot, and Johannes Gehrke. 2018. Improving optimistic concurrency control through transaction batching and operation reordering. *Proceedings of the VLDB Endowment* 12, 2 (2018), 169–182.

[17] Zhiyuan Dong, Zhaoguo Wang, Xiaodong Zhang, Xian Xu, Changgeng Zhao, Haibo Chen, Aurojit Panda, and Jinyang Li. 2023. Fine-grained re-execution for efficient batched commit of distributed transactions. *Proceedings of the VLDB Endowment* 16, 8 (2023), 1930–1943.

[18] Tamer Eldeeb and Philip A Bernstein. 2016. Transactions for Distributed Actors in the Cloud. *Transactions for Distributed Actors in the Cloud* (2016).

[19] Jose M Faleiro and Daniel J Abadi. 2014. Rethinking serializable multiversion concurrency control. *arXiv preprint arXiv:1412.2324* (2014).

[20] Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. 2017. High performance transactions via early write visibility. *Proceedings of the VLDB Endowment* 10, 5 (2017).

[21] Juncheng Fang, Farzad Habibi, Kevin Bruhwiler, Fayzah Alshammari, Abhishek Singh, Yinan Zhou, and Faisal Nawab. 2022. Pelopartition: Improving blockchain resilience to network partitioning. In *2022 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 274–281.

[22] Goetz Graefe, Mark Lillibridge, Harumi Kuno, Joseph Tucek, and Alistair Veitch. 2013. Controlled lock violation. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 85–96.

[23] Zhihan Guo, Kan Wu, Cong Yan, and Xiangyao Yu. 2021. Releasing locks as early as you can: Reducing contention of hotspots by violating two-phase locking. In *Proceedings of the 2021 International Conference on Management of Data*. 658–670.

[24] Ramesh Gupta, Jayant Haritsa, and Krithi Ramamritham. 1997. Revisiting commit processing in distributed database systems. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*. 486–497.

[25] Farzad Habibi. 2024. PhD Forum: Towards Metastable-Failure-Free Distributed Transaction Systems. In *2024 43rd International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 318–321.

[26] Farzad Habibi, Tania Lorido-Botran, Ahmad Showail, Daniel C Sturman, and Faisal Nawab. 2024. MSF-model: Queuing-based analysis and prediction of metastable failures in replicated storage systems. In *2024 43rd International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 12–22.

[27] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. 2022. Metastable Failures in the Wild. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 73–90.

[28] Dean Jacobs and Stefan Aulbach. 2007. Ruminations on multi-tenant databases. (2007).

[29] Evan PC Jones, Daniel J Abadi, and Samuel Madden. 2010. Low overhead concurrency control for partitioned main memory databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 603–614.

[30] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. 2020. Lessons Learned from the Chameleon Testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association.

[31] Ankur Khetrapal and Vinay Ganesh. 2006. HBase and Hypertable for large scale distributed storage systems. *Dept. of Computer Science, Purdue University* 10, 1376616.1376726 (2006).

[32] Hideaki Kimura, Goetz Graefe, and Harumi A Kuno. 2012. Efficient locking techniques for databases on modern hardware.. In *ADMS@ VLDB*. Citeseer, 1–12.

[33] Hsiang-Tsung Kung and John T Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)* 6, 2 (1981), 213–226.

[34] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review* 44, 2 (2010), 35–40.

[35] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M Patel, and Mike Zwilling. 2011. High-performance concurrency control mechanisms for main-memory databases. *arXiv preprint arXiv:1201.0228* (2011).

[36] Justin Levandoski, David Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. 2015. Multi-version range concurrency control in deuteronomy. *Proceedings of the VLDB Endowment* 8, 13 (2015), 2146–2157.

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

[38] Haonan Lu, Siddhartha Sen, and Wyatt Lloyd. 2020. {Performance-Optimal}{Read-Only} Transactions. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 333–349.

[39] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: a fast and practical deterministic OLTP database. (2020).

[40] Hatem A Mahmoud, Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2014. Maat: Effective and scalable coordination of distributed transactions in the cloud. *Proceedings of the VLDB Endowment* 7, 5 (2014), 329–340.

[41] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. 2014. Extracting more concurrency from distributed transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 479–494.

[42] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. 2014. Phase Reconciliation for Contended {In-Memory} Transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 511–524.

[43] Cuong DT Nguyen, Kevin Chen, Christopher DeCarolis, and Daniel J Abadi. 2025. Are Database System Researchers Making Correct Assumptions about Transaction Workloads? *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–26.

[44] Oracle. [n.d.]. ConcurrentHashMap (Java SE 11 and JDK 11). https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ConcurrentHashMap.html. Accessed April 4, 2025.

[45] Hexiang Pan, Shaofeng Cai, Tien Tuan Anh Dinh, Yuncheng Wu, Yeow Meng Chee, Gang Chen, and Beng Chin Ooi. 2025. CCaaLF: Concurrency Control as a Learnable Function. *arXiv preprint arXiv:2503.10036* (2025).

[46] Guna Prasaad, Alvin Cheung, and Dan Suciu. 2020. Handling highly contended OLTP workloads using fast dynamic partitioning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 527–542.

[47] Thamir M Qadah and Mohammad Sadoghi. 2018. Quecc: A queue-oriented, control-free concurrency architecture. In *Proceedings of the 19th International Middleware Conference*. 13–25.

[48] Dai Qin, Angela Demke Brown, and Ashvin Goel. 2021. Caracal: Contention management with deterministic concurrency control. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 180–194.

[49] P Krishna Reddy and Masaru Kitsuregawa. 2004. Speculative locking protocols to improve performance for distributed database systems. *IEEE Transactions on Knowledge and Data Engineering* 16, 2 (2004), 154–169.

[50] Kun Ren, Jose M Faleiro, and Daniel J Abadi. 2016. Design principles for scaling multi-core oltp under high contention. In *Proceedings of the 2016 International Conference on Management of Data*. 1583–1598.

[51] Daniel J Rosenkrantz, Richard E Stearns, and Philip M Lewis. 1978. System level concurrency control for distributed database systems. *ACM Transactions on Database Systems (TODS)* 3, 2 (1978), 178–198.
[52] Mohammad Sadoghi, Mustafa Canim, Bishwaranjan Bhattacharjee, Fabian Nagel, and Kenneth A Ross. 2014. Reducing database locking contention through multi-version concurrency. *Proceedings of the VLDB Endowment* 7, 13 (2014), 1331–1342.
[53] Zechao Shang, Feifei Li, Jeffrey Xu Yu, Zhiwei Zhang, and Hong Cheng. 2016. Graph analytics through fine-grained parallelism. In *Proceedings of the 2016 International Conference on Management of Data*. 463–478.
[54] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. 1995. Transaction chopping: Algorithms and performance studies. *ACM Transactions on Database Systems (TODS)* 20, 3 (1995), 325–363.
[55] Takayuki Tanabe, T. Hoshino, H. Kawashima, and O. Tatebe. 2020. An analysis of concurrency control protocols for in-memory databases with CCBench. *Proceedings of the VLDB Endowment* 13 (2020), 3531 – 3544. https://doi.org/10.14778/3424573.3424575
[56] Dixin Tang, Hao Jiang, and Aaron J Elmore. 2017. Adaptive Concurrency Control: Despite the Looking Glass, One Concurrency Control Does Not Fit All.. In *CIDR*, Vol. 2. 1.
[57] The Transaction Processing Council. 2007. TPC-C Benchmark (Revision 5.9.0).
[58] Alexander Thomasian. 1998. Distributed Optimistic Concurrency Control Methods for High-Performance Transaction Processing. *IEEE Trans. Knowl. Data Eng.* 10 (1998), 173–189. https://api.semanticscholar.org/CorpusID:11852807
[59] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD international conference on management of data*. 1–12.
[60] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 18–32.
[61] Tianzheng Wang and Hideaki Kimura. 2016. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *Proceedings of the VLDB Endowment* 10, 2 (2016), 49–60.
[62] Zhaoguo Wang, Shuai Mu, Yang Cui, Han Yi, Haibo Chen, and Jinyang Li. 2016. Scaling multicore databases via constrained parallel execution. In *Proceedings of the 2016 International Conference on Management of Data*. 1643–1658.
[63] Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan. 2014. Salt: Combining {ACID} and {BASE} in a distributed database. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 495–509.
[64] Philip S. Yu and Daniel M. Dias. 1992. Analysis of hybrid concurrency control schemes for a high data contention environment. *IEEE Transactions on Software Engineering* 18, 2 (1992), 118.
[65] 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. (2014).
[66] Yuan Yuan, Kaibo Wang, Rubao Lee, Xiaoning Ding, Jing Xing, Spyros Blanas, and Xiaodong Zhang. 2016. Bcc: Reducing false aborts in optimistic concurrency control with low cost for in-memory databases. *Proceedings of the VLDB Endowment* 9, 6 (2016), 504–515.
[67] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K Aguilera, and Jinyang Li. 2013. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 276–291.
[68] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. 2014. Fast databases with fast durability and recovery through multicore parallelism. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 465–477.

Received April 2025; revised July 2025; accepted August 2025