

PRDCC: PRedictive Deterministic Concurrency Control under High Contention

YU YAN, Harbin Institute of Technology, China
ZHIYU DAI, Harbin Institute of Technology, China
ZEKAI LV, Harbin Institute of Technology, China
SIJIA CHENG, Harbin Institute of Technology, China
YINGZE LI, Harbin Institute of Technology, China
HONGZHI WANG*, Harbin Institute of Technology, China

Deterministic databases present a promising direction for building scalable OLTP systems by establishing a global transaction order prior to execution, thereby eliminating costly runtime coordination overhead. However, existing methods face significant limitations: the Ordering-Constrained protocols strictly commit transactions in a predetermined TID order, limiting concurrency and scalability; meanwhile, execution-order-independent approaches can incur substantial conflicts, cascading aborts, and retries under high-contention workloads. In this paper, we propose PRDCC, a prediction-driven framework for deterministic concurrency control, designed to mitigate abort-induced throughput collapse under high data contention. PRDCC combines (i) a deterministic conflict predictor for extractable point/range predicates that avoids physical pre-execution and provides a zero-false-negative guarantee within its supported predicate class, and (ii) a deterministic partitioning algorithm that uses predicted dependencies to form an ordered sequence of execution blocks. Experimental results show that PRDCC improves throughput by up to $4.1\times$ over Aria on YCSB and $3.06\times$ on TPC-C under high contention, while maintaining competitive performance in low-contention workloads.

CCS Concepts: • **Software and its engineering** → **Concurrency control**.

Additional Key Words and Phrases: Deterministic Database, Transaction Management, Concurrency Control

ACM Reference Format:

Yu Yan, Zhiyu Dai, Zekai Lv, Sijia Cheng, Yingze Li, and Hongzhi Wang. 2026. PRDCC: PRedictive Deterministic Concurrency Control under High Contention. *Proc. ACM Manag. Data* 4, 4 (SIGMOD), Article 283 (September 2026), 26 pages. <https://doi.org/10.1145/3837121>

1 Introduction

Deterministic databases [40] offer a compelling path to building highly scalable, high-performance Online Transaction Processing (OLTP) systems by fundamentally eliminating the expensive runtime coordination overhead [17] found in traditional distributed systems, such as Two-Phase Commit (2PC) [28] for distributed transactions [8], or protocols like Paxos [23] and Raft [30] for state machine replication. This is achieved by reaching a consensus on the global transaction order before

*Corresponding author

Authors' Contact Information: Yu Yan, Harbin Institute of Technology, China, yanyu@stu.hit.edu.cn; Zhiyu Dai, Harbin Institute of Technology, China, daizhiyu@stu.hit.edu.cn; Zekai Lv, Harbin Institute of Technology, China, lvzekai@stu.hit.edu.cn; Sijia Cheng, Harbin Institute of Technology, China, sijiaorange@outlook.com; Yingze Li, Harbin Institute of Technology, China, 23b903046@stu.hit.edu.cn; Hongzhi Wang, Harbin Institute of Technology, China, wangzh@hit.edu.cn.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 2836-6573/2026/9-ART283

<https://doi.org/10.1145/3837121>

execution [1, 41]. By guaranteeing that all replicas process the same transactions in the same deterministic order, deterministic databases can eliminate coordination during the execution/commit path, allowing each replica to reproduce the same database state independently.

To achieve deterministic concurrency control, mainstream deterministic concurrency control protocols enter a deterministic stage to resolve conflicts and determine which transactions will commit. Specifically, after receiving a transaction batch with each transaction assigned a unique TID from the global sequencing layer, mainstream deterministic concurrency control protocols enter a deterministic stage to resolve conflicts and determine which transactions will commit, as shown in Figure 1a. Existing work can be broadly categorized into Ordering-Constrained and Execution-Order-Independent protocols. Ordering-Constrained protocols, which strictly commit transactions in a pre-determined TID order, mainly include two types of methods: those relying on pre-ordering and those designing strict ordered validation. Pre-ordering methods, such as Calvin [41], DecentSched [5], and Bohm [14], rely on actual read/write sets to plan conflict-free execution paths that conform to the given TID order. In contrast, other approaches enforce execution order through strict ordered validation, often at the cost of reduced concurrency. For instance, DOCC [12] employs serial validation to ensure that transactions commit in TID order. However, the pre-ordering methods that require actual read/write sets have poor applicability to dynamic transactions and can incur prohibitive pre-execution overhead [41]. Meanwhile, the strict serial validation severely limits multi-core scalability [12, 34].

The Execution-Order-Independent protocols [22, 32, 33, 46] allow more flexible transaction commitment to further improve concurrency. In the representative work Aria [26], all transactions are executed concurrently, and conflicts are subsequently resolved during validation using deterministic rules [21, 47]. Aborted transactions then re-run in the next batch. Building on Aria, some works further optimize concurrent execution in several ways: FGA and Dmucca use pre-executed read/write sets to guide scheduling decisions [11, 24], while others introduce dangerous relationship validation as in Harmony, or employ GPU-accelerated transaction splitting to boost concurrency as in LTPG. Execution-Order-Independent protocols offer better generality and concurrency [25, 43], but their approach leads to numerous cascading aborts and retries in high-contention workloads, wasting computational resources and causing performance collapse [45]. This issue is exacerbated by the re-injection of aborted transactions into subsequent batches.

There is a fundamental trade-off in deterministic scheduling. Ordering-Constrained methods avoid transaction re-running but sacrifice concurrency, while Execution-Order-Independent methods improve concurrency but suffer from cascading aborts and retries under high contention. Moderate contention is particularly challenging, as the benefit of proactive ordering may approach the cost of prediction, graph construction, and partitioning. In this paper, we aim to improve this trade-off by achieving higher concurrency with fewer aborts under high-contention workloads. The specific challenges are as follows:

a) High cost of obtaining conflict information. Existing deterministic protocols rely on actual read/write sets for conflict-aware scheduling, but obtaining them through transaction pre-execution incurs significant overhead and has limited applicability [35]. A primary challenge is to obtain conflict information at a low cost, without actual transaction execution.

b) Existing predictive approaches lack such theoretical guarantees, making them unsuitable for deterministic settings. Recent research has applied deep learning to predict transaction conflicts in non-deterministic concurrency control methods [31, 36]. These methods avoid transaction pre-execution but are inherently probabilistic. For a deterministic database, prediction must satisfy two strict constraints: the prediction process must be deterministic to ensure replicas reach identical scheduling decisions, and it must guarantee zero false negatives, as any missed conflict would compromise system state consistency.

c) Cascading aborts triggered by deterministic rules. In mainstream Execution-Order-Independent protocols, all transaction conflicts are evaluated simultaneously during a single validation phase using a fixed set of deterministic rules [48]. This approach fails to differentiate conflict priorities and thus cannot identify critical conflicts. Consequently, this indiscriminate application of rules often triggers unnecessary cascading aborts, severely limiting the number of transactions that can successfully commit within a batch.

To meet the above challenges, we propose PRDCC (PRedictive Deterministic Concurrency Control), with two core innovations: a deterministic conflict prediction model and an efficient deterministic partitioning algorithm. The main contributions are as follows:

a) We propose PRDCC, a prediction-driven deterministic concurrency control protocol described in Sec. 3, aiming to mitigate abort-induced collapse under high contention.

b) We design a deterministic conflict prediction model shown in Sec. 4 for extractable point/range predicates that guarantees zero false negatives within its scope and provides a tunable bound on false positives under standard hash-independence assumptions.

c) We propose an efficient deterministic partitioning algorithm in Sec. 5 that decomposes the predicted dependency graph into ordered WAW-free, acyclic execution blocks, enabling high intra-block concurrency.

d) Our evaluation in Sec. 6 shows that PRDCC improves throughput by up to 4.1x over existing works under high contention and improves performance stability; PRDCC-Stat demonstrates that full prediction and partitioning can eliminate aborts in certain settings.

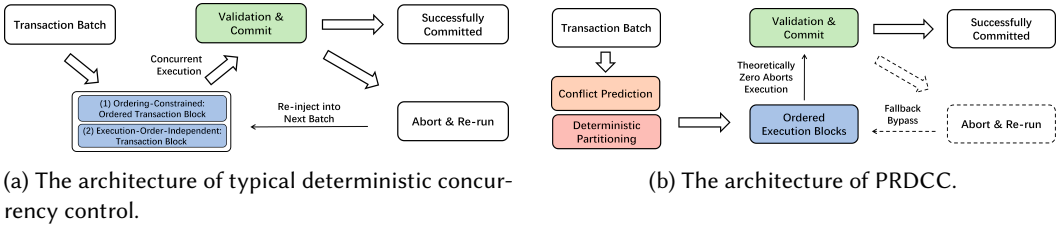


Fig. 1. Comparison of concurrency controls.

2 Background

Let the database state before the k -th scheduling round be S_k . A global sequencing layer continuously produces an ordered sequence of transaction blocks $\{b_1, b_2, \dots\}$, where each block $b_k = (T_{k,1}, \dots, T_{k,n_k})$ is an ordered tuple of transactions. All transactions are assigned globally unique and monotonically increasing transaction identifiers (TIDs). The TID order provides a deterministic reference order shared by all replicas.

PRDCC distinguishes three forms of transaction sets. A transaction block b is the primitive ordered unit produced by the sequencing layer. A transaction batch $\mathcal{B}$ is the unit on which PRDCC performs prediction and scheduling; it is formed by aggregating one or more consecutive transaction blocks to provide a larger optimization scope. An execution block $\mathcal{X}$ is the unit of concurrent execution and commitment. After scheduling, a batch $\mathcal{B}$ is deterministically decomposed into an ordered sequence of execution blocks $\Pi = (\mathcal{X}_1, \dots, \mathcal{X}_M)$.

A deterministic concurrency control protocol can be formalized as a deterministic scheduling function F . In the k -th scheduling round, this function takes the current database state S_k and an input set of transactions $\mathcal{I}_k$ as input, and outputs two sets: a set of successfully committed transactions C_k and a set of aborted transactions $\mathcal{A}_k$. This scheduling function must satisfy the

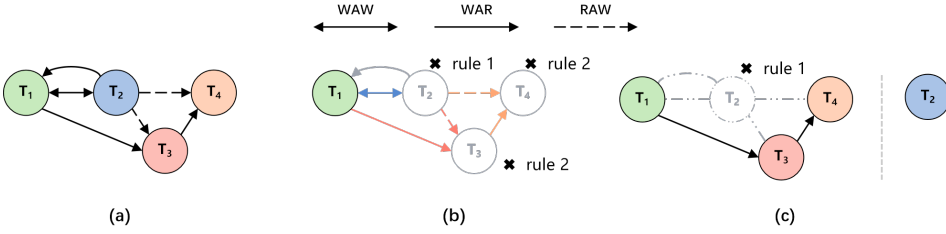


Fig. 2. Comparison of scheduling strategies.

following core constraints: a) **Determinism:** Given the exact same input (S_k, I_k) , the function F must produce the exact same output $(C_k, \mathcal{A}_k)$ on any replica, without coordination in the execution/commit path. b) **Correctness:** The new database state S_{k+1} is produced by serially applying the write operations of all transactions in C_k to S_k in TID order. The resulting state must be equivalent to a deterministic serial order induced by the ordered execution blocks, not necessarily the TID order.

3 Overview

Fundamental Idea Transaction aborts not only mean that all prior computation is wasted but can also trigger a chain reaction, leading to further aborts. This execution overhead is the primary source of system performance instability [34, 35]. Therefore, our first goal is to replace the high and uncontrollable cost of aborts with a low-cost, predictable analysis overhead. By introducing prediction and scheduling stages before transaction execution, PRDCC proactively avoids predictable data conflicts, transforming uncertain performance into a stable and efficient execution process.

Furthermore, we observe that rather than passively aborting conflicting transactions after validation, it is better to preemptively defer them during the scheduling phase. We call this the active deferral strategy. Its core is to substantially reduce the large and workload-dependent cost of aborts by trading it for a small, deterministic scheduling delay. A priori prediction enables the optimization of the deterministic validation process. In traditional protocols, concurrently evaluating all conflict rules can trigger unnecessary “cascading deferrals” [48]. To address this, we introduce staged decision making, which prioritizes absolute conflicts. This prevents erroneous decisions based on transactions that are already destined for deferral, thereby maximizing the concurrency within each execution block.

We illustrate this idea in Figure 2. Consider four transactions $\{T_1, T_2, T_3, T_4\}$ with the dependencies shown in part (a). In an Execution-Order-Independent protocol such as Aria (part (b)), all four transactions execute concurrently. During validation, the conflict between T_1 and T_2 causes T_2 to abort because T_1 has the smaller TID. Since the validation decisions are made over the original conflict relationships, dependencies involving T_2 further cause T_3 and T_4 to abort, leaving only T_1 to commit. In contrast, PRDCC identifies T_2 as the critical conflict node and defers it before execution (part (c)). Once T_2 is removed from the current block, the remaining transactions $\{T_1, T_3, T_4\}$ no longer contain these dependencies and can execute and commit concurrently. PRDCC applies this strategy to aggregated transaction batches and partitions them into multiple execution blocks.

New Architecture of PRDCC. Based on the above ideas, we propose a predictive deterministic concurrency control framework, whose workflow is depicted in Figure 1b: a) **Transaction Batch:** PRDCC receives multiple transaction blocks from the sequencing layer and aggregates them into a larger scheduling batch. b) **Conflict Prediction:** The scheduler invokes the deterministic conflict prediction model (shown in Sec. 4) to predict dependencies among transactions within the batch. c)

Deterministic Partitioning: Leveraging these predicted dependencies, the scheduler uses the deterministic partitioning algorithm (shown in Sec. 5) to decompose the batch into an ordered sequence of execution blocks $\{X_1, \dots, X_M\}$. d) **Block-wise Execution with Validation:** Execution proceeds sequentially across blocks to ensure serializability, while transactions within each block execute concurrently. Transactions in the same execution block do not observe one another's intermediate writes during execution. By deferring many predictable conflicts during scheduling (within the predictor's scope), PRDCC increases the commit success rate within each block. Any remaining conflicts, e.g., conflicts involving non-extractable queries, are detected by deterministic validation over the concrete read/write sets observed at runtime and handled via fallback.

Guarantees and Scope: PRDCC preserves replica determinism and correctness via deterministic post-execution validation and a fallback mechanism. The conflict predictor is a performance optimization: it is deterministic and provides a zero-false-negative guarantee only for extractable point/range predicates. Transactions outside this scope are routed to bypass execution block and validated deterministically, ensuring correctness regardless of prediction coverage.

4 Deterministic Conflict Prediction

In this section, we present our deterministic conflict prediction used to make a pre-execution decision, including the basic definitions (Sec. 4.1) and the model architecture (Sec. 4.2). We then detail two key components: an AST-based predicate range extractor (Sec. 4.3) and a two-level data summary (Sec. 4.4). Finally, Sec. 4.5 and Sec. 4.6 discuss model scope and practical considerations.

4.1 Basic Definition

We focus on extractable point/range predicates because they capture common index-based OLTP accesses, such as primary-key lookups, bounded range scans, and hotspot updates. Prior work shows that OLTP transactions often have small query footprints and frequently reduce to single-tuple index operations [18, 37], making our design a practical target.

Notation. We denote an SQL predicate (i.e., the WHERE-clause condition) by $\mathcal{P}$. In this section, we define the conflict predictor $\text{Pred}(\cdot, \cdot)$ only for extractable predicates that can be reduced to a set of point/range constraints. Predicates outside this class (e.g., involving UDFs, joins, or other data-dependent operators) are not handled by Pred . We fall back to bypass execution and deterministic post-execution validation for such transactions to preserve correctness and determinism.

To ensure correctness in a deterministic database, the conflict predictor must be a deterministic function $\text{Pred}(\cdot, \cdot)$ that maps any two transactions T_i and T_j with extractable predicates to a predicted dependency type. Let $\mathcal{D}(T_i, T_j)$ denote the true dependency (RAW, WAR, WAW, or independent IND), and let $\text{Pred}(T_i, T_j)$ denote the prediction result. The predictor must satisfy two constraints within its scope:

- **Zero False Negatives (within scope):**
 $\Pr[\text{Pred}(T_i, T_j) = \text{IND} \wedge \mathcal{D}(T_i, T_j) \neq \text{IND} \mid (T_i, T_j) \text{ is extractable}] = 0$. That is, the predictor never labels a truly conflicting extractable pair as independent.
- **Low False Positives (within scope):**
 $\Pr[\text{Pred}(T_i, T_j) \neq \text{IND} \wedge \mathcal{D}(T_i, T_j) = \text{IND} \mid (T_i, T_j) \text{ is extractable}] \leq \epsilon$. False positives, which cause unnecessary deferrals, should be minimized.

In theory, a transaction conflict exists between two transactions T_1 and T_2 if and only if their read/write sets—namely the read set $R(T)$ and the write set $W(T)$ —satisfy at least one non-empty intersection condition (RAW, WAR, WAW) [3]. The fundamental limitation of this criterion is its reactive nature; the precise read/write sets can only be known during or after transaction execution. PRDCC therefore does not attempt to determine the complete actual read/write set before execution.

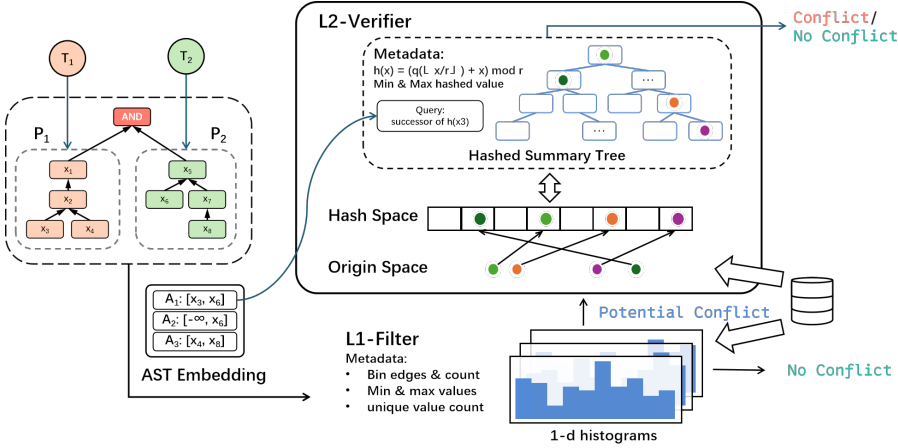


Fig. 3. Conflict Prediction Model Architecture: Predicate ranges are extracted via AST and equivalently transformed into a composite predicate, which is then efficiently evaluated by the two-level data summary.

Instead, its prediction stage identifies whether two transactions may have a dependency using statically extractable access predicates, while the concrete read/write sets observed at runtime are still checked by deterministic validation. For example, in TPC-C New-Order, the concrete order ID depends on the runtime value of `D_NEXT_O_ID`. PRDCC does not need to know this order ID in advance; it only needs to detect that two transactions targeting the same `WAREHOUSE/DISTRICT` field may access the same `D_NEXT_O_ID` value. This is sufficient to predict a potential dependency on the district hotspot object.

For any transaction T , it may contain multiple SQL statements, each with its own predicate and potentially over different relations. We therefore use $\mathcal{P}_T$ to denote the aggregate predicates induced by the extractable access predicates from transaction T . The operated data items $Op(T)$ (i.e., $R(T) \cup W(T)$) are the union of data items selected by these extractable predicates. We formalize this as follows:

Definition 1 (Data Set and Transaction Predicate). Let D be the set of all data items in the database (under the snapshot used for prediction). For any transaction T , its query condition can be represented as a logical predicate $\mathcal{P}_T$. The predicate $\mathcal{P}_T$ is a function such that, for any data item $t \in D$, $\mathcal{P}_T(t)$ returns a Boolean value in $\{\text{true}, \text{false}\}$.

Definition 2 (Transaction Operation Set). The operation set $Op(T)$ of a transaction T is a subset of D , consisting of all data items that satisfy its predicate $\mathcal{P}_T$:

$$Op(T) := \{t \in D \mid \mathcal{P}_T(t) = \text{true}\}. \quad (1)$$

Definition 3 (Conflict Condition). Given two concurrent transactions T_1 and T_2 , with corresponding predicates $\mathcal{P}_1$ and $\mathcal{P}_2$, a transaction conflict exists between T_1 and T_2 if and only if the intersection of their operation sets is non-empty:

$$\text{Conflict}(T_1, T_2) \iff Op(T_1) \cap Op(T_2) \neq \emptyset. \quad (2)$$

Directly computing $Op(T_1)$ and $Op(T_2)$ requires actually executing the queries, which is precisely what we want to avoid. Based on the above definitions, we can equivalently transform the set intersection problem into a decision problem regarding predicate logic.

Theorem 1. A conflict exists between two transactions T_1 and T_2 if and only if there exists at least one data item $t \in D$ that simultaneously satisfies both predicates $\mathcal{P}_1$ and $\mathcal{P}_2$. Equivalently, the

solution set of the composite predicate $\mathcal{P}_1 \wedge \mathcal{P}_2$ is non-empty over the data set D :

$$\text{Conflict}(T_1, T_2) \iff \{t \in D \mid (\mathcal{P}_1 \wedge \mathcal{P}_2)(t)\} \neq \emptyset. \quad (3)$$

Proof. If $Op(T_1) \cap Op(T_2) \neq \emptyset$, then there exists $t' \in Op(T_1) \cap Op(T_2)$. By Definition 2, this means $\mathcal{P}_1(t') = \text{true}$ and $\mathcal{P}_2(t') = \text{true}$; therefore, $(\mathcal{P}_1 \wedge \mathcal{P}_2)(t')$ holds, and the solution set satisfying $\mathcal{P}_1 \wedge \mathcal{P}_2$ is non-empty. The converse also holds. $\square$

This equivalence provides a useful abstraction for our predictor: it converts a physical set intersection problem, which depends on runtime data instantiation, into a logical problem of satisfiability of the composite predicate $\mathcal{P}_1 \wedge \mathcal{P}_2$, which can be analyzed before transaction execution.

4.2 Model Architecture

According to Theorem 1, our conflict prediction task is to efficiently determine if the composite predicate $\mathcal{P}_1 \wedge \mathcal{P}_2$ has a non-empty solution set, without scanning base records at runtime. Our model solves this using two core components, as illustrated in Figure 3.

First, the model receives the predicates of two transactions $(\mathcal{P}_1, \mathcal{P}_2)$ and merges them with a logical AND operation to construct a composite predicate AST. Subsequently, an AST-based predicate range extractor, detailed in Sec. 4.3, transforms the composite AST into one or more disjoint numerical range queries.

These range queries are then evaluated by a two-level data summary structure in Sec. 4.4. The first level of this structure is a lightweight L1 Range Filter (4.4.1). As depicted in the figure, this filter leverages coarse-grained statistics, such as 1-d histograms pre-built from the database, to perform rapid, low-overhead checks. For most queries, it can directly determine “No Conflict” and terminate the process. If a conflict cannot be ruled out, the query is marked as a “Potential Conflict” and forwarded to the next level.

The L2 Verifier (4.4.2) performs high-precision verification on these potential conflicts. To uphold the system’s zero-false-negative guarantee, it employs a locality-preserving double hashing function. This function maps values from the original space to a hash space, and all resulting hash values are stored in a self-balancing hashed summary tree. By executing efficient queries on this tree, the L2 Verifier performs an existence check for the query range and outputs a definitive “Conflict” or “No Conflict” result. This architecture enables our prediction model to achieve high execution efficiency while maintaining theoretical correctness within the supported predicate class.

4.3 AST-based Predicate Range Extraction

To map composite predicate from a write-involving predicate pair to data summary query interface, we transform logical constraints into geometric ranges via an AST-based extractor.

Initially, the SQL predicates of the transaction pair are parsed into individual ASTs. To capture the conflict condition defined in Theorem 1, we synthesize a composite AST by grafting these two trees as children under a new logical AND root node.

This composite tree serves as the input to Algorithm 1, which recursively computes a range set together with an extraction status. Internal nodes representing logical operators (Lines 2–9) aggregate the range sets returned by their subtrees using set intersection (AND) or union (OR); st_L and st_R denote the corresponding subtree statuses. If either status is UNKNOWN, the predicate is treated as non-extractable and routed to the fallback path. Leaf nodes serve as base cases: predicates on the target column are resolved into concrete intervals (Lines 10–11), while predicates independent of the target column are represented by the full range. By executing this function, we transform a complex existential predicate query into a set of range queries.

Algorithm 1 Extracting Ranges from a Predicate AST

Require: AST node ‘node’, target column ‘column’

Ensure: S is range set

```

1: switch (node.type) do
2:   case AND:
3:      $\langle S_L, st_L \rangle \leftarrow \text{EXTRACTRANGES}(\text{node.left}, \text{column})$ 
4:      $\langle S_R, st_R \rangle \leftarrow \text{EXTRACTRANGES}(\text{node.right}, \text{column})$ 
5:     return  $\text{INTERSECT}(S_L, S_R)$ 
6:   case OR:
7:      $\langle S_L, st_L \rangle \leftarrow \text{EXTRACTRANGES}(\text{node.left}, \text{column})$ 
8:      $\langle S_R, st_R \rangle \leftarrow \text{EXTRACTRANGES}(\text{node.right}, \text{column})$ 
9:     return  $\text{UNION}(S_L, S_R)$ 
10:  case Simple Predicate on ‘column’:
11:    return the single interval defined by the predicate.

```

4.4 Two-Level Data Summary

To efficiently answer the range queries produced by the range extractor, we design a two-level data summary architecture. It resolves the classic trade-off between precision and efficiency by cascading a lightweight L1 Range Filter with a high-precision L2 Verifier. The core idea is to use an extremely low-cost L1 filter, which may have false positives, to quickly filter out the vast majority of definitively conflict-free queries. Only when L1 cannot rule out a conflict do we activate the relatively more expensive but highly accurate L2 verifier for a final decision.

4.4.1 L1 Range Filter. The goal of the L1-Filter is to quickly determine if data might exist within a query range, in $O(1)$ time [13, 27].

Structure and Construction. We partition the entire key space D into contiguous buckets of a fixed size B . For each bucket, we maintain an exact counter. If the i -th bucket contains c data items from the data set D , its counter value is c . This parameter B serves as the fundamental atomic unit for range analysis. The construction requires only a single pass over D , incrementing the counter for the bucket corresponding to each key.

Query Mechanism. For a query range $[start, end]$, the L1 process is as follows: a) Calculate the starting block index idx_{start} and ending block index idx_{end} covered by the range. b) Check the counters of all blocks from idx_{start} to idx_{end} . c) If all counters in this interval are 0, we can certain that no data items exist within the query range. The L1-Filter returns “no conflict,” and the prediction process terminates. d) If at least one counter in this interval is greater than 0, it means that data may exist within the query range. The L1-Filter returns “possible conflict” and passes the query request to the L2-Verifier. To accelerate the range check, we can precompute a prefix-sum array, which reduces the time complexity of a range query from $O(idx_{end} - idx_{start})$ to $O(1)$.

4.4.2 L2 Verifier. When the L1-Filter reports a “possible conflict,” the L2-Verifier is activated to perform a high-precision verification on the given range, as is done in the Cold Filter sketch [9, 13, 49]. The design goal of L2 is to reduce the false positive rate caused by hash collisions to a theoretical minimum while guaranteeing zero false negatives for ranges handled by our extractor.

Theoretical Foundation. Traditional hash functions severely disrupt data locality, causing adjacent keys to be mapped to distant locations in the hash space, which makes range queries inefficient. To solve this problem, we introduce a form of locality-preserving double hashing.

Definition 4 (Double Hash Function). Our hash function $h(x)$ is defined as follows [10]:

$$h(x) = (q(\lfloor x/r \rfloor) + x) \pmod{r} \quad (4)$$

where r denotes the size of the hash space (codomain $[0, r - 1]$), and $q(\cdot)$ is a hash function selected from a pairwise independent family (e.g., $q(y) = ((\alpha y + \beta) \pmod{p}) \pmod{r}$).

It is crucial to note that r serves a dual physical role in this design: **Hash Space Size:** It determines the global capacity and collision probability of the summary structure; **Macroscopic Block Width:** It defines the width of partition blocks in the input key domain via the term $\lfloor x/r \rfloor$. By conceptually equating the input block width with the output hash space size, we ensure that within any single macroscopic block, the mapping $h(x)$ becomes a strictly *bijective circular shift*. This local bijectivity is the mathematical foundation that eliminates internal hash collisions, enabling the correctness of range queries. Furthermore, this width r is precisely the macroscopic boundary used for the hierarchical alignment with L1 buckets.

Hierarchical Alignment and Query Mechanism. To reconcile the conflicting requirements of granular filtering (favoring small L1 buckets) and collision resistance (favoring a large L2 hash space), we employ a hierarchical alignment strategy. We set the L2 hash space size (macroscopic block size) r to be an integer multiple of the L1 bucket size B (i.e., $r = m \cdot B$). This ensures that any L1 bucket is strictly contained within a single L2 macroscopic block.

We store the hash values of all keys in a hashed summary tree T_h . The query process for a range $I = [a, b]$ follows a *decompose-then-query* approach to guarantee correctness:

1) **Decomposition:** The query range I is decomposed into a set of sub-intervals $\{I_1, I_2, \dots\}$ aligned with the L1 bucket boundaries. 2) **L1 Pre-check:** For each I_j , we check the corresponding L1 bucket counter. If it is zero, I_j is skipped. 3) **L2 Verification:** If the counter is non-zero, we query T_h for $I_j = [a_j, b_j]$. Crucially, due to the alignment $r = m \cdot B$, each I_j falls strictly within one macroscopic block where the hash offset $q(\cdot)$ is constant. This ensures $h(x)$ is locally order-preserving (circularly). We simply check if any $y \in T_h$ exists in the circular interval defined by $[h(a_j), h(b_j)]$.

Specifically, we check if there exists $y \in T_h$ such that:

$$\begin{cases} h(a_j) \leq y \leq h(b_j) & \text{if } h(a_j) \leq h(b_j), \\ y \leq h(b_j) \vee y \geq h(a_j) & \text{otherwise.} \end{cases} \quad (5)$$

A conflict is reported if any sub-interval returns true.

The first case, where $h(a) \leq h(b)$, corresponds to the hash mapping preserving the order of the interval endpoints. In this case, the image set $h(I)$ corresponds to a single continuous interval $[h(a), h(b)]$ in the hash space. We verify this by finding the smallest element in T_h that is not less than $h(a)$ (i.e., the successor element $h_k = \text{successor}(h(a) - 1)$). If such an h_k exists and $h_k \leq h(b)$, we determine that a conflict exists; otherwise, there is no conflict. This operation has a time complexity of $O(\log |D|)$ on a balanced tree.

The second case occurs when the hashed endpoints wrap around (i.e., $h(a) > h(b)$). Due to the linear operation and modulo, the target interval is split into two discontinuous sub-intervals in the hash space: $[0, h(b)]$ and $[h(a), r - 1]$. We only need to check if any element in T_h falls into either of these two intervals. This can be done efficiently by querying the minimum and maximum values in T_h : if $T_h.\text{min} \leq h(b)$ or $T_h.\text{max} \geq h(a)$, a conflict is determined. If the minimum/maximum values in the tree are maintained in advance, this operation has a time complexity of $O(1)$.

Theoretical Guarantees. To meet the rigorous requirements of deterministic concurrency control, our prediction model ensures three fundamental properties. Foremost is the guarantee of **Zero False Negatives**. By leveraging the locality-preserving nature of our circular hash intervals, the L2-Verifier ensures that any query range I intersecting with the dataset ($D \cap I \neq \emptyset$) triggers a definitive

“conflict” report, thereby avoiding missed conflicts within scope where conflicting transactions are erroneously marked as safe. While safety is absolute, performance is optimized through a **Controllable False Positive Rate**. Since false positives in the L2 verifier arise solely from hash collisions, their probability is mathematically bounded. By applying the union bound over the dataset of size n and a query range of length l , the error rate is limited by: $P_{FP} \leq \sum_{x \in D} \sum_{y \in I} P(h(x) = h(y)) \leq \frac{n \cdot l}{r} \leq \epsilon$. So, by configuring the hash space size r proportional to the data volume and query scope (i.e., $r \propto n \cdot l / \epsilon$), we can limit the error rate to a configurable threshold to preserve high concurrency. Finally, the architecture ensures **Strict Determinism**. As the entire pipeline—from AST extraction to L2 tree traversal—relies solely on the immutable database snapshot and fixed logic without external entropy, the model yields identical scheduling decisions across all replicas given the same input and parameters.

4.5 Dynamic Maintenance of the Data Summary

To remain effective under continuous updates, the data summary must track changes to the underlying data. Our prototype maintains it synchronously, but write-intensive workloads may make this cost a bottleneck because inserts, deletes, and key updates modify the L1 counters and L2 hashed summary tree. This issue is manageable in principle: a promising direction is to use asynchronous or batched maintenance, or reuse existing statistics and query-optimization components to amortize overhead. The following maintenance procedure describes the synchronous mechanism used in our current prototype.

Let a data change operation be $op(k)$, where op is the operation type (INSERT, DELETE) and k is the key of the affected record. The summary maintenance process is as follows:

- a) **INSERT(k)**: When a new record is inserted, the counter of the corresponding block in L1 is incremented, and $h(k)$ is inserted into the L2 hashed summary tree T_h .
- b) **DELETE(k)**: When a record is deleted, the counter of the corresponding block in L1 is decremented, and $h(k)$ is deleted from T_h when this counter becomes zero.
- c) **UPDATE(k_{old}, k_{new})**: When a record key is modified, the operation is decomposed into two steps: DELETE(k_{old}) and INSERT(k_{new}).

To avoid false negatives during summary maintenance, the summary is maintained as an over-approximation of committed keys: for INSERT, the summary is updated before the data commit; for DELETE, the data commits before the summary is updated. This prevents maintenance from hiding visible keys. However, dynamic key changes remain outside the predictor’s scope: if a transaction moves a record from outside a range predicate into that range, a pre-execution summary may not detect the conflict. PRDCC preserves correctness through deterministic post-execution validation over the observed read/write sets; such missed conflicts are deterministically aborted and retried.

The maintenance operations are designed to be lightweight. For the L1 Range Filter, locating and updating the specific bucket counter via the key k involves simple arithmetic mapping, incurring strictly $O(1)$ time complexity. For the L2 Verifier, the cost is dominated by the structural adjustments of the self-balancing hashed summary tree T_h . Recall that $|D| = n$. Both insertion and deletion operations on T_h operate in $O(\log n)$ time. Consequently, the UPDATE operation, being a composition of one deletion and one insertion, also strictly maintains a logarithmic complexity of $O(\log n)$.

4.6 Scope and Considerations

While our two-level data summary is highly efficient at predicting conflicts for range/point predicates, it has inherent limitations on complex, data-dependent queries (e.g., predicates involving UDFs and joins, whose access sets cannot be conservatively bounded pre-execution). As a practical trade-off, PRDCC treats such transactions as non-predictable and routes them into a dedicated

bypass block appended after all predictable execution blocks. This design prevents an unpredictable minority from interleaving with predicted transactions and triggering cascading aborts.

Transactions in the bypass block execute under an execution-order-independent protocol, followed by a deterministic validation pass. A transaction T_i commits iff it has no WAR or WAW conflict with any already-committed transaction T_j ; otherwise, T_i is deterministically aborted and retried in the next batch. The validator performs these checks using the concrete read/write sets observed during speculative execution, thereby ensuring both serializability and replica determinism. Consequently, when the fraction of non-predictable transactions increases, PRDCC degrades smoothly toward the baseline protocol rather than losing correctness.

We adopt this scope–cost trade-off deliberately: extending the predictor to precisely cover such queries would require substantially richer analysis (e.g., deep semantic reasoning over UDFs and cross-table correlations), which can increase prediction latency and CPU overhead and potentially negate the throughput gains from proactive conflict management.

5 Deterministic Partitioning Algorithm

This section details our deterministic partitioning algorithm, which aims to produce conflict-free, highly concurrent execution blocks.

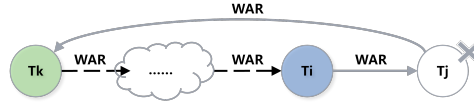


Fig. 4. Illustration of a potential-cycle risk structure.

5.1 Problem Formulation and Design Goals

We construct a directed dependency graph [2, 15], where transactions are vertices and directed edges represent predicted scheduling dependencies. The direction of a logical edge is defined with respect to the pre-assigned global TID order and does not describe the actual runtime interleaving order of read/write operations. Given a transaction batch $\mathcal{B}$ and its graph $G = (V, E)$ constructed by the prediction model (where $V = \mathcal{B}$), our goal is to compute an ordered partition $\Pi = (\mathcal{X}_1, \mathcal{X}_2, \dots, \mathcal{X}_M)$ of $\mathcal{B}$. This partition decomposes the original batch into a series of mutually exclusive and exhaustive subsets, which we term execution blocks.

To ensure concurrent safety, this partition must satisfy a core constraint: for every execution block $\mathcal{X}_k$ in the sequence, its induced subgraph in the predicted dependency graph, $G_k = G[\mathcal{X}_k]$, must simultaneously meet two conditions: a) its edge set must contain no WAW dependencies, and b) it must be a directed acyclic graph (DAG). These constraints determine which transactions may be placed in the same concurrent execution block; they are not intended to describe the physical read/write order during execution. Subject to these constraints, the optimization objective is to minimize the number of execution blocks, M .

However, finding the optimal partition (i.e., minimizing M) is an NP-hard problem, equivalent to computing the dependency graph dichromatic number [4, 19]. Therefore, our design goal is to construct a computationally lightweight, fully deterministic algorithm that guarantees acyclicity while generating high-quality approximate solutions. To achieve this, our algorithm is built upon two core design principles:

Principle One: Active Deferral. This principle shifts conflict resolution from reactive, post-execution aborts to proactive, pre-scheduling deferrals. If a candidate transaction is determined to violate the concurrent safety of the current execution block, it is not aborted but instead

deterministically placed into a deferred set for a subsequent block. This approach systematically replaces the high, uncontrollable cost of execution aborts with a predictable scheduling delay.

Principle Two: Staged Decision. Active deferral requires accurately identifying which transactions to defer. A naive approach that evaluates all conflict types simultaneously can lead to a cascading deferral effect, where a transaction is unnecessarily deferred due to a dependency on another transaction that will not be admitted to the current block (e.g., because it has already been deferred for a higher-priority reason such as a WAW conflict).

To address this, we identify two conflict patterns with distinct priorities and establish a strict order: a) **Absolute Conflicts:** WAW dependencies. Such conflicts directly violate serializability requirements, rendering any set of transactions containing them unsafe for concurrent execution.

b) **Potential Cycle Risks:** The introduction of transaction T_j forms a bridge structure $T_i \rightarrow T_j \rightarrow T_k$ within the current block as in Figure 4, where $T_i, T_k \in \mathcal{X}_{\text{cur}}$. Since transactions are added incrementally, any newly formed cycle must pass through the new transaction T_j . This implies T_j must have both an incoming edge from the current block and an outgoing edge to it. Detecting and preventing this structure is sufficient for safety to guarantee acyclicity

For any candidate transaction, the algorithm first checks for absolute WAW conflicts. A transaction with such a conflict is immediately deferred, terminating its evaluation. Only if no WAW conflicts are present does the algorithm proceed to evaluate potential cycle risks. This ensures decisions are based on a stable set of transactions cleared of absolute conflicts. It prevents erroneous deferrals by disregarding dependencies that involve transactions already set to be deferred. This strategy maximizes the concurrency of each execution block, as illustrated in Figure 2.

5.2 The Deterministic Partitioning Algorithm

We propose the deterministic partitioning algorithm detailed in Algorithm 2. It iteratively constructs an ordered partition Π of the batch. Each iteration partitions the remaining transactions ($\mathcal{B}_{\text{rem}}$) into a concurrently safe execution block ($\mathcal{X}_{\text{cur}}$), and transactions that cannot be safely included are placed into a deferred set $\mathcal{B}_{\text{def}}$ for the next round.

In each iteration, our algorithm performs a two-phase process to build an execution block. Phase 1 (Lines 7–11) filters transactions with absolute conflicts to create a WAW-free candidate set. Phase 2 (Lines 12–16) then builds the final acyclic block by deferring transactions that would form a potential-cycle risk structure to prevent cycles. This staged approach maximizes concurrency by preventing erroneous deferrals based on transactions already destined to be deferred.

Finally, algorithm determinism is guaranteed by three core elements: a) processing always follows the globally unique TID order; b) the underlying conflict prediction function is itself deterministic; and c) the partitioning logic is based on deterministic rules. The algorithm is also guaranteed to converge, as at least one transaction (the one with the minimum TID) is successfully assigned to an execution block in each iteration.

Correctness and Determinism Analysis

We formally establish that Algorithm 2 is deterministic and that every generated execution block is concurrently safe.

Determinism. All replicas start from the same input batch $\mathcal{B}$ and process transactions in the same globally ordered TID sequence. For each transaction, the decision of whether it is assigned to $\mathcal{B}_{\text{cand}}$, $\mathcal{X}_{\text{cur}}$, or $\mathcal{B}_{\text{def}}$ depends only on (i) the deterministic conflict predictor and (ii) fixed partitioning rules in Phase 1 and Phase 2. Therefore, the decision made for each transaction is identical across replicas, and the resulting ordered partition Π is deterministic.

Concurrent safety. The staged construction guarantees that each execution block is both WAW-free and acyclic. First, Phase 1 guarantees WAW-freedom. Suppose, for contradiction, that two transactions T_u and T_v with $TID(T_u) < TID(T_v)$ and a WAW dependency both appear in the

same execution block. Since transactions are processed in increasing TID order, T_u must already be in $\mathcal{B}_{\text{cand}}$ when T_v is examined. Then the Phase 1 WAW check would defer T_v to $\mathcal{B}_{\text{def}}$, contradicting the assumption that both are admitted to the same block.

Algorithm 2 Deterministic Partitioning Algorithm

Require: Transaction batch $\mathcal{B}$, Dependency Graph $G = (V, E)$

Ensure: Ordered execution-block sequence $\Pi = (\mathcal{X}_1, \dots, \mathcal{X}_M)$

```

1:  $\mathcal{B}_{\text{rem}} \leftarrow \mathcal{B}$ 
2:  $\Pi \leftarrow \emptyset$ 
3: while  $\mathcal{B}_{\text{rem}} \neq \emptyset$  do
4:    $\mathcal{X}_{\text{cur}} \leftarrow \emptyset$ 
5:    $\mathcal{B}_{\text{def}} \leftarrow \emptyset$ 
6:    $\mathcal{B}_{\text{cand}} \leftarrow \emptyset$ 
   // Phase 1: Filter a WAW-free candidate set
7:   for all  $T_j \in \mathcal{B}_{\text{rem}}$  (sorted by TID) do
8:     if  $T_j$  has no WAW conflict with any  $T_i \in \mathcal{B}_{\text{cand}}$  then
9:       Add  $T_j$  to  $\mathcal{B}_{\text{cand}}$ 
10:    else
11:      Add  $T_j$  to  $\mathcal{B}_{\text{def}}$ 
   // Phase 2: Build an acyclic execution block
12:   for all  $T_j \in \mathcal{B}_{\text{cand}}$  (sorted by TID) do
13:     if  $\nexists \{T_i, T_k\} \subseteq \mathcal{X}_{\text{cur}}$  s.t.  $(T_i \rightarrow T_j) \in E \wedge (T_j \rightarrow T_k) \in E$  then
14:       Add  $T_j$  to  $\mathcal{X}_{\text{cur}}$ 
15:     else
16:       Add  $T_j$  to  $\mathcal{B}_{\text{def}}$ 
   // Finalize round and prepare for the next
17:   Append  $\mathcal{X}_{\text{cur}}$  to  $\Pi$ 
18:    $\mathcal{B}_{\text{rem}} \leftarrow \mathcal{B}_{\text{def}}$ 
19: return  $\Pi$ 

```

Second, Phase 2 guarantees acyclicity. Suppose, for contradiction, that an execution block contains a directed cycle. Let T_{max} be the transaction with the largest TID on that cycle. By definition of a cycle, T_{max} must have both a predecessor T_{prev} and a successor T_{next} on the cycle. Since both have smaller TIDs than T_{max} , they must already have been admitted into $\mathcal{X}_{\text{cur}}$ when T_{max} is evaluated. Hence, adding T_{max} would create the bridge structure $T_{\text{prev}} \rightarrow T_{\text{max}} \rightarrow T_{\text{next}}$ explicitly checked in Phase 2, so T_{max} would be deferred rather than admitted. This contradiction shows that no cycle can remain in any execution block.

5.3 Practical Considerations and Optimizations

To ensure that the scheduling latency introduced by PRDCC does not impact the critical path of system throughput, we introduce a set of optimizations ranging from low-level implementation techniques to high-level adaptive policies.

Parallelizing Conflict Prediction.

The construction of the dependency graph involves $O(N^2)$ pairwise conflict checks as N transactions, which could be computationally expensive. However, these checks are inherently stateless and independent, relying only on immutable transaction predicates and a read-only data summary

snapshot. We therefore decompose the prediction tasks and distribute them across a worker thread pool [20, 29], reducing the parallel scheduling latency from $O(N^2)$ to $O(N^2/P)$ on a system with P cores. Because L1 eliminates most obviously independent pairs and only unresolved pairs reach L2, the practical cost is substantially lower than the worst-case logical bound. This per-batch in-memory scheduling cost is intended to replace the less predictable physical cost of repeated execution, validation, rollback, retry, and memory management under high contention.

Bitset-based Graph Representation. To further minimize scheduling overhead, we implement the dependency graph using compact bitsets. For a batch of size N , the adjacency matrix is stored as N bit-vectors, allowing the partitioning algorithm to check dependency constraints via bit-parallel logical AND instructions. This representation leverages CPU parallelism to process up to 64 edges (on 64-bit architecture) in a single cycle, reducing the per-transaction verification cost to $O(\lceil N/W \rceil)$, where W is the machine word size. Consequently, the overall partitioning complexity is optimized to $O(N^2/W)$, which provides significant speedup over standard pointer-chasing graph traversals and improves cache locality for practical batch sizes (e.g., hundreds of transactions).

Adaptive Strategy Selection. To mitigate prediction overhead under low contention, PRDCC employs an adaptive mechanism. It deterministically selects the operating mode for the next batch by monitoring the Abort Ratio against a global threshold:

- a) High contention mode: When the abort ratio exceeds the threshold, the system enables conflict prediction and partitioning algorithm to proactively manage conflicts and maximize throughput;
- b) Low contention mode: When the abort ratio is below the threshold, prediction is disabled to avoid unnecessary scheduling overhead. The correctness and determinism remain ensured by validation and fallback, so PRDCC degrades gracefully to an execution-order-independent protocol.

This mechanism allows PRDCC to intelligently balance prediction costs against benefits, improving robustness while preserving replica determinism (the mode decision is deterministic given the same observed abort ratio and threshold). Furthermore, the current threshold-based adaptation can be extended into a learned policy. By integrating lightweight learning models [38] to forecast workload trends and dynamically tune the switching threshold, the system could achieve finer-grained adaptability across complex, fluctuating contention landscapes.

6 Evaluation

6.1 Experimental Setup

6.1.1 Hardware and Software Platform. All experiments are conducted on a two-node cluster. Each server has 8 virtual CPUs, 32GB of RAM, and runs on Ubuntu 22.04. To ensure a fair comparison, all protocols are implemented within our unified C++17 in-memory database prototype.

6.1.2 Baseline Protocols. We compare PRDCC against a series of representative baseline protocols to comprehensively evaluate its performance across different dimensions:

Aria [26]: This is our most important comparison target, representing the Execution-Order-Independent protocols that do not require actual read/write sets before execution.

Calvin [41]: As a representative of Ordering-Constrained protocols that rely on actual read/write sets. We assume precise read/write sets are available with zero extraction cost as an idealized reference. This setting is not intended to model a realistic deployment baseline, but to provide a reference for the performance of Ordering-Constrained methods when precise dependency information is available. Calvin- x denotes the variant with x lock manager threads.

DOCC [12]: An representative Ordering-Constrained protocol that does not require actual read/write sets before execution. It executes transactions optimistically but enforces the deterministic order through a strict serial validation and commit phase.

Bohm [14]: A representative pre-ordering method in the Ordering-Constrained category. It relies on prior knowledge of transaction operations and pre-allocates version placeholders for write operations, enabling deterministic execution according to the input transaction order.

6.1.3 Our Method Variants. To deeply analyze the design choices of PRDCC, we evaluate its two main versions:

PRDCC: Our full proposal, including the adaptive strategy selection mechanism.

PRDCC-Stat: A variant with the adaptive mechanism disabled, which always performs the full conflict prediction and deterministic partitioning process.

6.1.4 Benchmarks. We use two industry-standard benchmarks to simulate OLTP workloads with different characteristics:

YCSB [7]: We use YCSB to generate a key-value store workload with 10 million records. Following Aria, we make YCSB transactional by wrapping multiple YCSB operations into a single transaction. Since each operation is a point access to a primary-key record, the resulting transaction touches at most 10 distinct keys. All YCSB transactions in our setup are predictable because their accesses are extractable point predicates over the primary key. By adjusting the Zipf skew factor of the request key distribution, we control the degree of workload contention, from a nearly uniform low-contention scenario to a highly concentrated high-contention scenario with hotspots.

TPC-C [42]: TPC-C is a more complex OLTP benchmark that simulates a real-world order processing system. We implement three core transactions: New-Order, Payment, and Delivery. New-Order performs 13–33 reads and 13–33 row writes including inserts per transaction, depending on the number of order lines. Payment may first perform an auxiliary customer-index lookup when the customer is selected by last name, and then updates WAREHOUSE, DISTRICT, and CUSTOMER, resulting in roughly 4 reads and 3 writes. Delivery is more data-dependent: for each district in a warehouse, it selects the oldest undelivered order at runtime, removes the corresponding NEW_ORDER entry, and updates ORDERS, ORDER_LINE, and CUSTOMER; its read/write cardinality depends on the runtime-selected orders and order-line counts. The implemented TPC-C transactions are handled by the predictor through extractable point/range predicates over these tables, and the workload creates hotspot conflicts on WAREHOUSE, DISTRICT, and NEW_ORDER, making it suitable for evaluating hotspot-driven and data-dependent OLTP workloads.

6.1.5 Key Parameters and Metrics. Unless otherwise specified, all experiments use the following default settings.

Core Metrics: We primarily focus on three performance metrics: throughput (K tps), abort rate, and P95 transaction latency.

Experimental Procedure: Each run lasts for 25 seconds including 10-second warmup and 5-second cooldown. We repeat each run 5 times and report the average with standard deviation error bars.

System Configuration: Experiments are conducted on two nodes, with each node using 8 worker threads; The transaction block size is 100 transactions. PRDCC aggregates five transaction blocks into a single scheduling batch; The default YCSB workload is Write-Heavy, with a 5:95 read-to-write ratio; PRDCC switches to high contention mode when the batch abort rate exceeds 40%, balancing prediction benefits against scheduling overhead.

6.2 Performance under High Contention

6.2.1 YCSB: Throughput versus Data Skew. We evaluate protocols on YCSB with varying skew (θ) under **Write-Heavy** (5:95 R/W) and **Balanced** (50:50 R/W) workloads (Figure 5). The challenge of conflict detection in YCSB lies in conflicts arising from random overlaps among small access sets and being highly sensitive to data skew. Under low skew, the overlap probability between

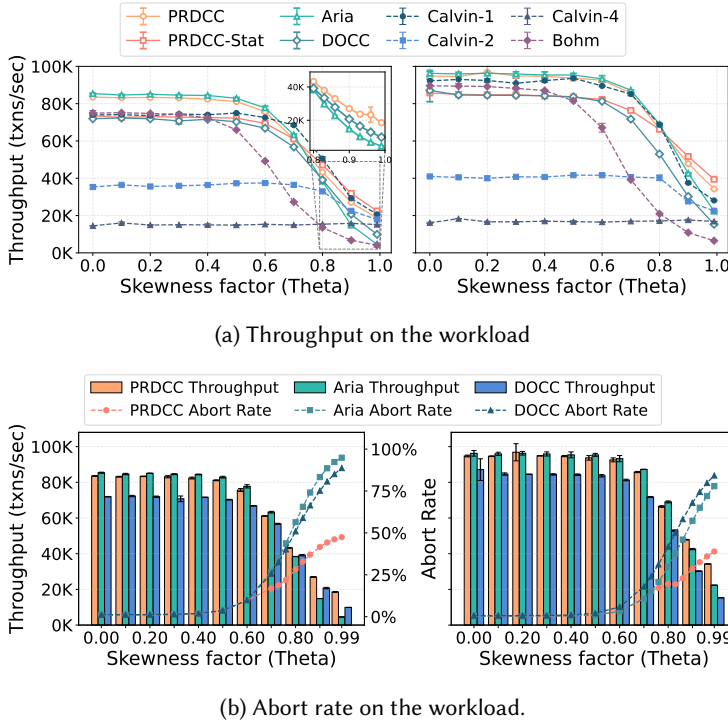


Fig. 5. Throughput and abort rate versus the Zipf skew factor under the YCSB benchmark. Write-Heavy workload (left) & Balanced workload (right). PRDCC-Stat and Calvin are omitted as they maintain 0% abort.

transactions is low, most are true negatives, and the benefits of prediction and reordering are limited. Under high skew, the dependency graph becomes dense, making proactive partitioning more beneficial.

a) Low Contention Performance. At $\theta < 0.6$, PRDCC closely matches Aria (85K tps), while outperforming DOCC, Bohm, and the Calvin variants. Notably, PRDCC's adaptive mechanism bypasses the prediction overhead that limits PRDCC-Stat to about 75K tps, achieving high performance in low-contention scenarios.

b) High Contention Performance. In the Write-Heavy workload at extreme skew ($\theta = 0.99$), Aria collapses to 4.5K tps due to a 94.9% abort rate. Conversely, PRDCC sustains 18.6K tps, a **4.1x improvement**. As skew approaches 0.7, PRDCC switches to prediction mode. At skew 0.9, prediction mode reduces the abort rate from 83.3% in Aria to 41.6%.

The residual abort rate reflects a trade-off governed by our 40% threshold. This cross-workload setting responds earlier to rising contention while avoiding unconditional prediction overhead in lower-contention phases. Notably, **PRDCC-Stat maintains a 0% abort rate**, confirming that full prediction can eliminate aborts; at $\theta = 0.99$, it reaches 22.5K tps and slightly exceeds Calvin-1 without relying on prior read/write set knowledge.

c) Performance on Balanced Workloads. In balanced workloads, Aria remains competitive because fewer writes reduce abort costs. This setting also exposes a weakness of full prediction: under moderate contention, the true conflict rate is still not high enough for abort avoidance to amortize the cost of graph analysis. For example, PRDCC-Stat performs below Aria around $\theta = 0-0.7$, indicating that Aria's optimistic execution path can be more favorable when abort costs

remain low. At $\theta = 0.99$, Aria reaches 22.3K tps, while PRDCC-Stat reaches 39.5K tps. Benefiting from the adaptive strategy, PRDCC remains comparable to Aria under low skew and performs better under high skew, reaching 34.3K tps at $\theta = 0.99$.

6.2.2 TPC-C: Generality on Complex OLTP Workloads. To evaluate the generality of PRDCC under complex transactional logic, we employ the TPC-C benchmark. We control the contention level by varying the number of warehouses from 1 (extreme contention) to 100 (low contention). The throughput results are visualized in Figure 6 and summarized in Table 1.

Table 1. Performance comparison on the TPC-C benchmark.

Protocol	Throughput	Std Dev	Abort Rate	Improvement
High Contention (1 Warehouse)				
PRDCC	14.9K	0.4	46.8%	3.06x
Aria	4.9K	0.03	93.5%	
Low Contention (100 Warehouses)				
PRDCC	43.0K	0.7	13.1%	0.94x
Aria	45.9K	0.9	13.5%	

a) **Performance under Extreme Contention.** With 1 warehouse, PRDCC outperforms Aria by **3.06x**. TPC-C’s complex logic increases the penalty for aborts and retries: Aria commits 4.9K tps with a 93.5% abort rate, while PRDCC commits 14.9K tps and reduces the abort rate to 46.8%. Despite the overhead of analyzing complex SQL statements, PRDCC’s conflict prediction and partitioning remain beneficial for computationally expensive transactions under extreme contention.

b) **Effect of Contention and Adaptation.** As the number of warehouses increases, TPC-C retains structured hotspot accesses, so full prediction remains useful across a wider contention range. Therefore, under high and moderate contention, PRDCC performs close to PRDCC-Stat and clearly outperforms Aria. However, starting from the moderate-contention regime, the weakness of PRDCC-Stat also begins to appear: when abort costs remain relatively low, Aria’s optimistic execution path becomes more favorable. At lower contention, adaptive switching begins to reduce unnecessary prediction overhead: at 100 warehouses, PRDCC reaches 43.0K tps, slightly above PRDCC-Stat’s 41.5K tps, and close to Aria’s 45.9K tps, with similar abort rates for PRDCC and Aria (13.1% vs. 13.5%). Thus, in TPC-C, the benefit of adaptation is relatively modest and mainly appears near the low-contention end, while PRDCC remains competitive with Aria.

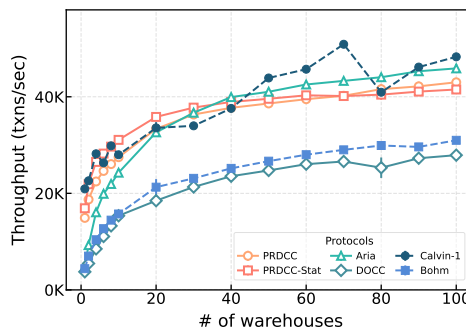


Fig. 6. Throughput of each protocol versus the number of warehouses under the TPC-C benchmark.

We observe different performance of Calvin in low contention for TPC-C and YCSB. Under TPC-C, Calvin's pre-execution lock acquisition avoids post-execution validation and almost eliminates conflict-induced aborts, achieving higher throughput. Under YCSB at low skew, Calvin's centralized lock manager becomes bottleneck, while PRDCC's adaptive strategy disables prediction and falls back to execution-order-independent mode, thereby achieving better scalability.

6.3 Time Cost Analysis

6.3.1 Tail Latency Analysis. To complement the throughput and abort-rate results, we also report tail latency in Figure 7. In YCSB, PRDCC reports a P95 latency of 1.66 ms, higher than Aria (1.17 ms) and DOCC (1.14 ms), but lower than Calvin-1 (3.12 ms) and Bohm (13.81 ms). In TPC-C, PRDCC reports a P95 latency of 2.89 ms, between Aria/DOCC (2.07/1.78 ms) and Calvin-1/Bohm (3.23/5.13 ms). These results are consistent with PRDCC's design: deterministic conflict prediction and partitioning introduce pre-execution scheduling overhead, while reducing expensive abort/retry behavior in high-contention workloads.

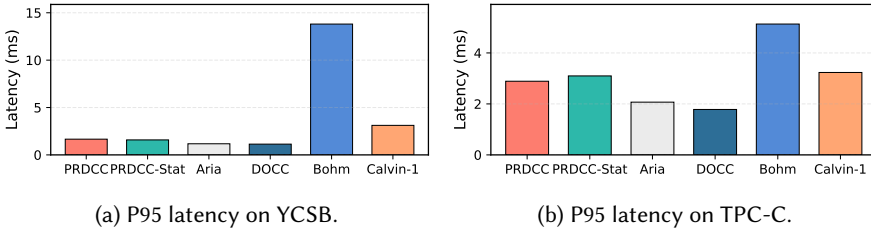


Fig. 7. P95 latency distribution under YCSB and TPC-C workloads.

6.3.2 Cost Breakdown Analysis. To quantify scheduling overhead, we analyze the execution-time breakdown of PRDCC-Stat under four representative settings: TPC-C with 10 and 100 warehouses, and YCSB under Write-Heavy (5:95 R/W) and Balanced (50:50 R/W) workloads.

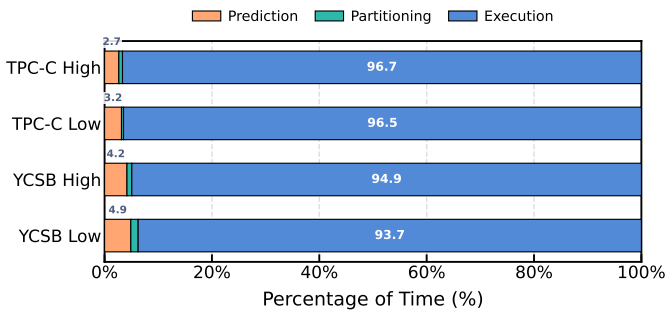


Fig. 8. Cost breakdown of PRDCC-Stat across TPC-C and YCSB workloads.

As illustrated in Figure 8, the scheduling overhead of PRDCC-Stat remains moderate across evaluated settings. Because PRDCC-Stat always enables deterministic conflict prediction and partitioning, this experiment measures the cost of the proactive scheduling stage. The combined prediction and partitioning cost accounts for only 3.34%–6.25% of total time. Prediction is the dominant component, while partitioning contributes only 0.37%–1.37%, indicating that graph partitioning itself is not the main bottleneck.

6.4 Robustness Analysis

This section evaluates the robustness of PRDCC under workload drift, parameter sensitivity, and unpredictable queries.

6.4.1 Effect of Batch Aggregation. We study how the number of transaction blocks aggregated before scheduling affects PRDCC on the YCSB Write-Heavy workload (as shown in Figure 9). Each transaction block contains 100 transactions, and the figure reports committed transactions per scheduling batch as multiple blocks are aggregated for prediction and partitioning.

Low Contention. At skew < 0.6 , most transaction pairs are independent, so PRDCC commits nearly all transactions and committed transactions per batch grow almost proportionally with aggregation.

High Contention. At skew 0.99, denser predicted dependencies cause PRDCC to partition the batch into numerous small (avg. size 3) but internally conflict-free blocks.

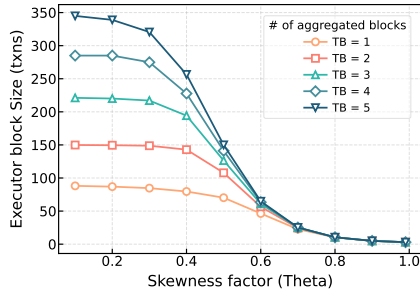


Fig. 9. Committed transactions per scheduling batch versus the number of transaction blocks aggregated.

6.4.2 Dynamic Adaptability under Workload Drift. Real-world OLTP systems rarely face static workloads. We evaluate PRDCC under a 60-second mixed YCSB workload with three phases: Balanced (0–20s), Write-Heavy (20–40s), and Balanced (40–60s).

We monitor the real-time throughput and abort rate for both Aria and PRDCC, as visualized in Figure 10. During the initial balanced phase, PRDCC commits about 33K transactions per second, compared with about 22K for Aria, while keeping the abort rate around 39% rather than Aria's 78%. When contention spikes at 20s, Aria's performance drops to about 4.6K tps with a 95% abort rate. In contrast, PRDCC's adaptive mechanism responds to the higher abort ratio and stabilizes throughput at about 18.6K tps (roughly **4.1x** over Aria), with the abort rate around 47%. After the workload returns to the balanced phase at 40s, PRDCC recovers to about 35K tps, while Aria returns to about 22K tps. This confirms that PRDCC's adaptive mechanism provides robustness against workload drift without permanently incurring scheduling overhead.

6.4.3 Sensitivity Analysis of Adaptive Threshold. We analyze the sensitivity of the adaptive threshold on TPC-C by varying it from 0% to 100% (left panel of Figure 11). A threshold of 0% enforces full prediction for every batch, eliminating aborts and reaching 20.5K tps in this high-contention setting. Conversely, a 100% threshold degenerates to Aria-like behavior, causing aborts to spike to 81.6%. Between these extremes, the abort rate remains stable across a broad threshold range (10%–70%) because PRDCC chooses the operating mode for the next batch according to the abort ratio observed in the previous batch. Near the switching boundary, the system may alternate between bypass mode and full-prediction mode.

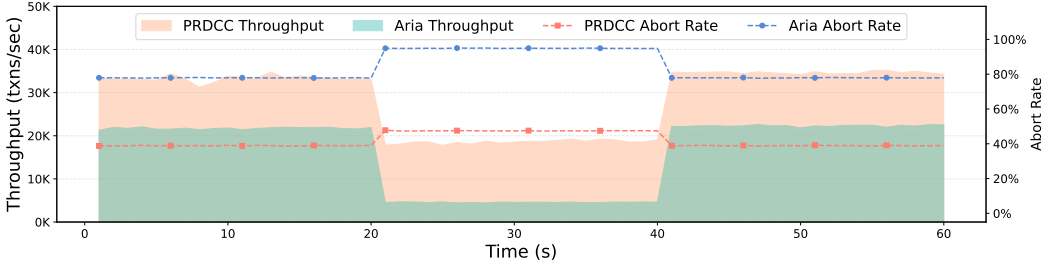


Fig. 10. Comparison of performance and abort rate between PRDCC and Aria under a changing workload.

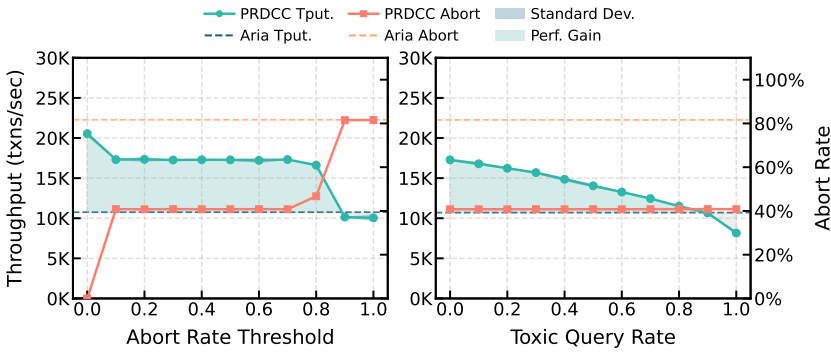


Fig. 11. Sensitivity to the adaptive abort-rate threshold and the ratio of complex (unpredictable) queries.

6.4.4 Robustness against Complex Queries. Real-world workloads often contain SQL queries with logic that resists static analysis, such as UDFs and join operations. We evaluate the robustness of PRDCC by varying the unpredictable query ratio from 0% to 100% on TPC-C. The term "toxic query" refers to queries that lack the predictor's supported predicate scope in our implementation. Our experiment simulates unpredictable dependencies by randomly erasing the prediction result for a transaction pair with probability `toxic_query_rate`, and the pair is then treated as non-predictable.

As shown in the right panel of Figure 11, PRDCC's throughput decreases gradually as more transaction pairs are treated as non-predictable. It drops from 17.3K tps at 0% to 14.9K tps at 40%, and further to 10.7K tps at 90%, which remains close to Aria's 10.7K tps. When all pairs are treated as non-predictable, PRDCC drops to 8.2K tps, below Aria, while the measured abort rate remains around 40.8%. This trend is consistent with PRDCC's scope: partial prediction coverage can still reduce abort/retry costs, but the benefit narrows as more transactions are routed through the non-predictable bypass path.

7 Related Work

Existing research in deterministic concurrency control can be broadly classified into two categories based on whether they adhere to the TID order for commitment:

Ordering-Constrained protocols.

Ordering-Constrained protocols strictly commit transactions according to the pre-determined TID order. One category relies on actual read/write sets to plan a conflict-free execution path. For instance, Calvin [41] employs a single-threaded analysis phase before execution to traverse the

actual read/write sets and construct a global dependency graph. Worker threads then execute non-conflicting transactions concurrently based on this graph. DecentSched [5] similarly constructs the dependency graph in a decentralized, concurrent way. Bohm [14] adopts an operation-granularity reservation approach, pre-allocating version placeholders for write operations during analysis. The common limitation of these methods is reliance on a priori knowledge, which restricts applicability when actual read/write sets are not known beforehand.

To avoid the dependency on a priori knowledge, another class of Ordering-Constrained protocols, like DOCC [12], allow transactions to execute speculatively and concurrently. They then employ a strict serial validation phase, where transactions are checked for conflicts and committed in TID order. This serial validation, while ensuring order and determinism, becomes a major bottleneck that limits multi-core scalability. Block-STM [16] builds on this line of work by introducing optimizations such as write-set speculation, where aborted transactions retain write placeholders from previous attempts to reduce conflicts for subsequent read operations. Although these protocols enhance generality by removing the need for a priori knowledge, the concurrency they sacrifice to preserve strict ordering limits performance on modern multi-core hardware.

Execution-Order-Independent protocols.

Execution-Order-Independent protocols permit the commit order to deviate from the TID order, creating opportunities for performance optimization. In Aria [26], all transactions in a batch execute speculatively and concurrently based on a common database snapshot, with read/write operations buffered in workspaces. After execution, the system enters a validation phase where transaction conflicts are detected and resolved according to deterministic rules. Many systems [32, 33] have adopted similar concurrency control models, tailored to their domains. LTPG [46] employs GPU-accelerated transaction splitting to boost concurrency. Harmony [22] is a blockchain database, introducing a dangerous relationship validation mechanism to reduce conflicts.

Furthermore, recent studies have applied machine learning to non-deterministic concurrency control to optimize performance. These works mainly focus on two aspects: one line of research uses supervised models to predict transaction conflict probability [31, 36] and inform scheduling decisions. Another approach employs reinforcement learning or evolutionary algorithms to explore the policy space and discover scheduling strategies tailored to specific workloads [6, 39, 44].

8 Conclusion

In this paper, we propose PRDCC, a novel prediction-driven method for deterministic concurrency control, designed to address the performance collapse of traditional methods under high data contention. Experimental results show that PRDCC mitigates abort-induced collapse under high contention and improves throughput by up to $4.1\times$ on YCSB and $3.06\times$ on TPC-C compared with execution-order-independent protocols such as Aria.

In future work, we plan to extend PRDCC in two key directions. First, we aim to address the maintenance overhead of the prediction model in write-intensive scenarios by exploring asynchronous maintenance strategies (e.g., epoch-based batched updates) and probabilistic data structures. These techniques will trade a bounded increase in false positives for significantly lower update latency on the critical path. Second, we intend to integrate the PRDCC prototype into a mature DBMS. Then, we plan to evaluate PRDCC with more realistic trace-driven workloads and broader benchmark suites. This will allow us to validate its effectiveness under complex storage hierarchies and realistic durability constraints, paving the way for industrial adoption.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (No. 62502126, 62232005)

References

- [1] Daniel J. Abadi and Jose M. Faleiro. 2018. An overview of deterministic database systems. *Commun. ACM* 61, 9 (Aug. 2018), 78–88. doi:10.1145/3181853
- [2] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*. 1–15.
- [3] Philip A Bernstein and Nathan Goodman. 1981. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)* 13, 2 (1981), 185–221.
- [4] Drago Bokal, Gasper Fijavz, Martin Juvan, P Mark Kayll, and Bojan Mohar. 2004. The circular chromatic number of a digraph. *Journal of Graph Theory* 46, 3 (2004), 227–240.
- [5] Chen Chen, Xingbo Wu, Wenshao Zhong, and Jakob Eriksson. 2024. Fast Abort-Freedom for Deterministic Transactions. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 692–704.
- [6] Shuhan Chen, Congqi Shen, and Chunming Wu. 2025. Intelligent Transaction Scheduling to Enhance Concurrency in High-Contention Workloads. *Applied Sciences* 15, 11 (2025), 6341.
- [7] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.
- [8] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google’s globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.
- [9] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.
- [10] Marco Costa, Paolo Ferragina, and Giorgio Vinciguerra. 2024. Grafite: Taming adversarial queries with optimal range filters. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–23.
- [11] 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.
- [12] Zhi-Yuan Dong, Chu-Zhe Tang, Jia-Chen Wang, Zhao-Guo Wang, Hai-Bo Chen, and Bin-Yu Zang. 2020. Optimistic transaction processing in deterministic database. *Journal of Computer Science and Technology* 35, 2 (2020), 382–394.
- [13] Cristian Estan and George Varghese. 2002. New directions in traffic measurement and accounting. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*. 323–336.
- [14] Jose M Faleiro and Daniel J Abadi. 2014. Rethinking serializable multiversion concurrency control. *arXiv preprint arXiv:1412.2324* (2014).
- [15] 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 (Jan. 2017), 613–624. doi:10.14778/3055540.3055553
- [16] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. 2023. Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 232–244.
- [17] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. 2017. An evaluation of distributed concurrency control. *Proceedings of the VLDB Endowment* 10, 5 (2017), 553–564.
- [18] Michael Haubenschild and Viktor Leis. 2025. Oltp in the cloud: architectures, tradeoffs, and cost. *The VLDB Journal* 34, 4 (2025), 42.
- [19] Winfried Hochstättler, Felix Schröder, and Raphael Steiner. 2020. On the complexity of digraph colourings and vertex arboricity. *Discrete Mathematics & Theoretical Computer Science* 22, Graph Theory (2020).
- [20] Kaisong Huang, Tianzheng Wang, Qingqing Zhou, and Qingzhong Meng. 2023. The art of latency hiding in modern database engines. *Proceedings of the VLDB Endowment* 17, 3 (2023), 577–590.
- [21] 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.
- [22] Ziliang Lai, Chris Liu, and Eric Lo. 2023. When Private Blockchain Meets Deterministic Database. *Proceedings of the ACM on Management of Data* 1, 1 (May 2023), 1–28. doi:10.1145/3588952
- [23] Leslie Lamport. 2001. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)* 32, 4 (Whole Number 121, December 2001) (2001), 51–58.
- [24] Xingchen Li, Xinyuan Wang, and Hejiao Huang. 2024. Practical Deterministic Transaction Processing with Low-cost Re-execution. In *2024 IEEE 30th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 487–495.
- [25] 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.

- [26] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: a fast and practical deterministic OLTP database. Proceedings of the VLDB Endowment 13, 12 (Aug. 2020), 2047–2060. doi:10.14778/3407790.3407808
- [27] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2006. An integrated efficient solution for computing frequent and top-k elements in data streams. ACM Transactions on Database Systems (TODS) 31, 3 (2006), 1095–1133.
- [28] C Mohan, Bruce Lindsay, and Ron Obermarck. 1986. Transaction management in the R* distributed database management system. ACM Transactions on Database Systems (TODS) 11, 4 (1986), 378–396.
- [29] Steven M Nowick and Montek Singh. 2011. High-performance asynchronous pipelines: An overview. Ieee design & test of computers 28, 5 (2011), 8–22.
- [30] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In 2014 USENIX annual technical conference (USENIX ATC 14). 305–319.
- [31] Hexiang Pan, Shaofeng Cai, Yeow Meng Chee, Gang Chen, Tien Tuan Anh Dinh, Yuncheng Wu, and Beng Chin Ooi. 2025. CCaaS: Concurrency Control as a Learnable Function. arXiv preprint arXiv:2503.10036 (2025).
- [32] Zeshun Peng, Yanfeng Zhang, Qian Xu, Haixu Liu, Yuxiao Gao, Xiaohua Li, and Ge Yu. 2022. Neuchain: a fast permissioned blockchain system with deterministic ordering. Proceedings of the VLDB Endowment 15, 11 (2022), 2585–2598.
- [33] Evaggelia Pitoura and Panos K Chrysanthis. 1999. Scalable processing of read-only transactions in broadcast push. In Proceedings. 19th IEEE International Conference on Distributed Computing Systems (Cat. No. 99CB37003). IEEE, 432–439.
- [34] 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.
- [35] Kun Ren, Alexander Thomson, and Daniel J Abadi. 2014. An evaluation of the advantages and disadvantages of deterministic database systems. Proceedings of the VLDB Endowment 7, 10 (2014), 821–832.
- [36] Yangjun Sheng, Anthony Tomasic, Tieying Zhang, and Andrew Pavlo. 2019. Scheduling OLTP transactions via learned abort prediction. In Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management. 1–8.
- [37] Michael Stonebraker, Samuel Madden, Daniel J Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2018. The end of an architectural era: it's time for a complete rewrite. In Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker. 463–489.
- [38] Jian Tan, Tieying Zhang, Feifei Li, Jie Chen, Qixing Zheng, Ping Zhang, Honglin Qiao, Yue Shi, Wei Cao, and Rui Zhang. 2019. ibtune: Individualized buffer tuning for large-scale cloud databases. Proceedings of the VLDB Endowment 12, 10 (2019), 1221–1234.
- [39] Darren Teh and Jianzhi Wang. 2025. Investigations in Optimal Transaction Scheduling. (2025).
- [40] Alexander Thomson and Daniel J Abadi. 2010. The case for determinism in database systems. Proceedings of the VLDB Endowment 3, 1-2 (2010), 70–80.
- [41] 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. ACM, Scottsdale Arizona USA, 1–12. doi:10.1145/2213836.2213838
- [42] Transaction Processing Performance Council. 2010. TPC Benchmark C. <http://www.tpc.org/tpcc/>
- [43] 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.
- [44] Jiachen Wang, Ding Ding, Huan Wang, Conrad Christensen, Zhaoguo Wang, Haibo Chen, and Jinyang Li. 2021. Polyjuice: {High-Performance} transactions via learned concurrency control. In 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21). 198–216.
- [45] Xinyuan Wang, Yun Peng, and Hejiao Huang. 2024. Gria: an efficient deterministic concurrency control protocol. Frontiers of Computer Science 18, 4 (2024), 184204.
- [46] Jianpeng Wei, Yu Gu, Tianyi Li, Jianzhong Qi, Chuanwen Li, Yanfeng Zhang, Christian S. Jensen, and Ge Yu. 2024. LTPG: Large-Batch Transaction Processing on GPUs with Deterministic Concurrency Control. In 2024 IEEE 40th International Conference on Data Engineering (ICDE). IEEE, Utrecht, Netherlands, 3865–3877. doi:10.1109/ICDE60146.2024.00296
- [47] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. Tictoc: Time traveling optimistic concurrency control. In Proceedings of the 2016 International Conference on Management of Data. 1629–1642.
- [48] Huan Zhang, Xiaodong Qi, Haibo Tang, Zhao Zhang, Cheqing Jin, and Aoying Zhou. 2025. Loom: A Deterministic Execution Framework Towards Nested Contract Transactions. In 2025 IEEE 41st International Conference on Data Engineering (ICDE). IEEE, 2408–2421.
- [49] Yang Zhou, Tong Yang, Jie Jiang, Bin Cui, Minlan Yu, Xiaoming Li, and Steve Uhlig. 2018. Cold filter: A meta-framework for faster and more accurate stream processing. In Proceedings of the 2018 International Conference on Management of Data. 741–756.

Appendix Navigation

In this appendix, we provide detailed theoretical proofs, extended analyses, and supplementary experiments to further substantiate the design choices and performance claims of PRDCC. To help readers navigate this rich supplementary material, we offer the following roadmap:

- (1) Sec. A presents some comparisons between typical deterministic concurrency control and our PRDCC.
- (2) Sec. B presents the formal proof of correctness of L2 Verifier, demonstrating that our L2-Verifier guarantees zero false negatives by ensuring that any data item within a query range is deterministically mapped to a conflicting hash bucket interval.
- (3) Sec. C provides the derivation for false positive analysis, quantifying the probability of false positives in the L2-Verifier. It establishes the relationship between hash space size and collision probability, proving that the error rate is controllable via parameter tuning.
- (4) Sec. D proves the determinism of our prediction, confirming that the entire conflict prediction pipeline, from AST parsing to L1/L2 summary querying, is a deterministic function of the database state, ensuring consistent behavior across all replicas.
- (5) Sec. E presents micro-benchmark results for the core components of PRDCC. We first analyze the effect of batch aggregation on the number of committed transactions per scheduling batch, and then evaluate the accuracy of the deterministic conflict prediction model using TP, FP, TN, and FN statistics under TPC-C and YCSB workloads.
- (6) Sec. F explains how PRDCC handles Unknown transactions whose access sets cannot be deterministically extracted. We describe the Unknown Bypass Strategy (Algorithm ??), including bypass-block execution/validation and next-batch fallback for aborted transactions.

A Typical Deterministic Concurrency Control VS PRDCC.

After receiving a transaction batch with each transaction assigned a unique TID from the global sequencing layer, mainstream deterministic concurrency control protocols enter a deterministic stage to resolve conflicts and determine which transactions will commit.

Ordering-Constrained protocols like Calvin obtain actual read/write sets to construct a dependency graph and formulate an execution plan of conflict-free, ordered transaction blocks, to prevent conflicts during execution. The Execution-Order-Independent methods first execute all transactions concurrently on a database snapshot. A validation phase then inspects the actual read/write sets to resolve conflicts deterministically. This validation can be parallel with TID-based rules as in the Execution-Order-Independent protocol Aria, or serial to enforce commit order as in the Ordering-Constrained protocol DOCC.

However, these traditional approaches struggle under high data contention. PRDCC, inserts a lightweight, deterministic scheduling stage prior to execution. This stage uses a conflict prediction model and a deterministic partitioning algorithm to decompose the transaction batch into internally conflict-free execution blocks without physically pre-executing transactions to obtain read/write sets. We aim to replace the high, uncontrollable cost of transaction aborts with a predictable scheduling overhead, thereby maintaining system stability and high throughput under high contention.

B Correctness of L2 Verifier

For any query range $I = [a, b]$, if $D \cap I \neq \emptyset$, the query result of the L2-Verifier will necessarily be “conflict.”

Proof. Assume there exists a data item $x^* \in D \cap I$. We show the model always detects it. First, the decomposition step splits I into sub-intervals $\{I_j\}$. Since $x^* \in I$, there must exist exactly one

sub-interval $I_k = [a_k, b_k]$ such that $x^* \in I_k$. Second, the L1 pre-check for I_k will pass because the bucket counter corresponding to x^* must be at least 1. Third, we consider the L2 verification on I_k . Because of the alignment $r = m \cdot B$ and the fact that $|I_k| \leq B$, the sub-interval I_k is strictly contained within a single L2 macroscopic block determined by $\lfloor a_k/r \rfloor$. Within this block, the hash offset $C = q(\lfloor a_k/r \rfloor)$ is constant. Thus, for any $x \in I_k$, $h(x) = (C + x) \pmod{r}$. This linearity implies that the image set $h(I_k)$ forms a contiguous circular interval from $h(a_k)$ to $h(b_k)$ in the hash space. Since $x^* \in D$, $h(x^*)$ is present in T_h . Since $x^* \in I_k$, $h(x^*)$ must fall within this circular interval. The tree query logic (checking the range defined by endpoints) explicitly covers both standard ($h(a_k) \leq h(b_k)$) and wrapped ($h(a_k) > h(b_k)$) cases. Therefore, the verifier will find $h(x^*)$ and report a conflict. $\square$

C False Positive Analysis

The only cause of false positives in the L2-Verifier is hash collisions, and its probability P_{FP} is controllable.

Proof. Assume the indexed key domain is discrete (integer keys), and let l denote the number of keys in the query interval $I = [a, b]$, i.e., $l = b - a + 1$. A false positive occurs if and only if $D \cap I = \emptyset$, but there exists a data point $x' \in D$ and a point $y \in I$ such that $h(x') = h(y)$.

We first analyze the collision probability. For any two distinct keys $x' \neq y$, if they are not in the same macroscopic block (i.e., $\lfloor x'/r \rfloor \neq \lfloor y/r \rfloor$), the condition $h(x') = h(y)$ is equivalent to $q(\lfloor x'/r \rfloor) - q(\lfloor y/r \rfloor) \equiv y - x' \pmod{r}$. Since q is chosen from a pairwise independent hash family over $\mathbb{Z}_r$, the probability of this equality holding is $1/r$.

If they are in the same macroscopic block, then $q(\lfloor x'/r \rfloor) = q(\lfloor y/r \rfloor)$ and $h(x') = h(y)$ implies $x' \equiv y \pmod{r}$, which is impossible for distinct keys within the domain $[0, r - 1]$. Therefore, the collision probability for any pair of distinct keys is bounded by $1/r$.

We use the union bound to calculate the overall false positive probability. Let $|D| = n$.

$$P_{FP} = P\left(\bigcup_{x' \in D, y \in I} \{h(x') = h(y)\}\right) \leq \sum_{x' \in D} \sum_{y \in I} P(h(x') = h(y)). \quad (6)$$

Substituting the upper bound for collision probability $1/r$, we get:

$$P_{FP} \leq \sum_{x' \in D} \sum_{y \in I} \frac{1}{r} = \frac{n \cdot l}{r}. \quad (7)$$

To control the false positive rate below a preset threshold ϵ , we set the hash space size $r = \lceil nL/\epsilon \rceil$, where L is the maximum query interval length allowed by the system ($l \leq L$). Then, the false positive probability satisfies:

$$P_{FP} \leq \frac{n \cdot l}{nL/\epsilon} = \frac{l}{L} \epsilon \leq \epsilon. \quad (8)$$

By choosing a sufficiently large parameter r , we can control the false positive probability P_{FP} to an arbitrarily low acceptable level. $\square$

D Determinism of Prediction

Our proposed two-level conflict prediction model is a deterministic function. Given the exact same database state $\mathcal{S}$ and input transaction pair (T_i, T_j) , the model will produce an identical prediction result on any replica.

Proof. The model determinism stems from the inherent determinism of each of its computational steps. First, parsing SQL predicates into ASTs is a standard, deterministic process. Algorithm 1, based on fixed logical and set-theoretic rules, deterministically converts an identical AST into an identical set of query intervals.

Second, querying the data summary is deterministic. All replicas build identical data summaries from the same database state $\mathcal{S}$. Consequently, the L1-Filter query on the same interval will yield a consistent result. If the query proceeds to the L2-Verifier, the hash function $h(x)$ and its parameters are globally fixed across all replicas. Thus, the hashed summary tree T_h built from the same database state $\mathcal{S}$ is also identical. Query operations on T_h are deterministic algorithms, making the L2-Verifier decision deterministic as well.

Since every stage in the pipeline—from predicate analysis to L1/L2 verification—is deterministic, the entire prediction model, as a composite function, is also deterministic. This ensures that all replicas will consistently reach the same prediction conclusion without coordination. $\square$

E Prediction Accuracy Analysis

We evaluate the accuracy of our deterministic conflict prediction model by running PRDCC on TPC-C and YCSB workloads. We record the counts of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN) to compute precision and recall.

Table 2. Accuracy evaluation of the conflict prediction model under different workloads.

Workload	TP	FP	TN	FN	Precision	Recall
TPC-C	81645	371	917984	0	99.55 %	100 %
YCSB-Write-Heavy	9996	11	989993	0	99.89 %	100 %
YCSB-Balanced	10008	0	989992	0	100 %	100 %

The results in Table 2 highlight two key properties of our model:

a) **Absolute Safety (Zero False Negatives)**. Across all evaluated transactions within the predictor’s supported predicate class, the model false negative (FN) count is consistently zero, resulting in a perfect 100% recall. This result suggests that for the evaluated standard OLTP workloads, our extractor does not miss conflicting pairs, guaranteeing system correctness and providing a safe foundation for its use in deterministic databases.

b) **Excellent Precision (Low False Positives)**. While ensuring safety, the model demonstrates extremely high precision, exceeding 99.5% in all workloads. This means the model rarely makes overly conservative predictions (i.e., false positives), reducing unnecessary deferrals so that the partitioning algorithm can preserve high concurrency.

In summary, the micro-benchmarks show that PRDCC achieves high prediction accuracy within scope and that partitioning can reduce aborts by proactively deferring predicted conflicts.

F Handling Unknown Transactions

Transactions whose access sets cannot be deterministically extracted by Algorithm 1 are treated as $\mathcal{T}_{unknown}$. PRDCC first partitions the predictable transactions $\mathcal{T}_{predict}$ using Algorithm 2, and then appends all $\mathcal{T}_{unknown}$ transactions, together with any fallback transactions from the previous batch, to a single bypass block ordered by TID.

The bypass block executes speculatively on the database snapshot produced by the preceding predictable blocks and is followed by deterministic validation. A transaction T_i commits if and only if it has no WAR/WAW conflict with a committing transaction T_j with $TID(T_j) < TID(T_i)$. Transactions that fail validation are deterministically aborted and re-injected into the bypass block of the next batch with the highest priority. Thus, unsupported predicates reduce prediction coverage but do not affect correctness or replica determinism.

Received 17 January 2026; revised 19 May 2026; accepted 12 June 2026