Modeling Concurrency Control as a Learnable Function

HEXIANG PAN, National University of Singapore, Singapore
SHAOFENG CAI, National University of Singapore, Singapore
TIEN TUAN ANH DINH, Deakin University, Australia
YUNCHENG WU, Renmin University of China, China
YEOW MENG CHEE, Singapore University of Technology and Design, Singapore
GANG CHEN, Zhejiang University, China
BENG CHIN OOI, Zhejiang University, China

Concurrency control (CC) algorithms are important in modern transactional databases, as they enable high performance by executing transactions concurrently while ensuring correctness. However, state-of-the-art CC algorithms struggle to perform well across diverse workloads, and most do not consider workload drifts.

In this paper, we propose NeurCC, a novel learned concurrency control algorithm that achieves high performance across diverse workloads. The algorithm is quick to optimize, making it robust against dynamic workloads. It learns a function that captures a large number of design choices from existing CC algorithms. The function is implemented as an efficient in-database lookup table that maps database states to concurrency control actions. The learning process is based on a combination of Bayesian optimization and a novel graph reduction search algorithm, which converges quickly to a function that achieves high transaction throughput. We compare NeurCC against five state-of-the-art CC algorithms and show that it consistently outperforms the baselines both in transaction throughput and in optimization time.

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

Additional Key Words and Phrases: Transaction Processing, Concurrency Control, Machine Learning, AIxDB

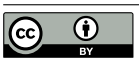
ACM Reference Format:
Hexiang Pan, Shaofeng Cai, Tien Tuan Anh Dinh, Yuncheng Wu, Yeow Meng Chee, Gang Chen, and Beng Chin Ooi. 2026. Modeling Concurrency Control as a Learnable Function. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 211 (June 2026), 28 pages. https://doi.org/10.1145/3802088

1 Introduction

Concurrency control (CC) algorithms allow database transactions to exploit hardware concurrency while ensuring correctness, i.e., transaction isolation. As modern database systems are no longer disk-bound, CC is essential for achieving high transaction processing performance. Classical CC algorithms, such as two-phase locking (2PL) and optimistic concurrency control (OCC), do not perform well across all workloads [46, 58]. In particular, 2PL typically outperforms OCC for high-contention workloads, but the reverse is true for low-contention workloads.

Recent works on CC combine multiple classical CC algorithms to support different workloads. They adopt the *classify-then-assign* approach, in which transaction operations are classified into different types, then the most appropriate CC algorithm is selected for each type. The classification

Authors' Contact Information: Hexiang Pan, National University of Singapore, Singapore, panh@u.nus.edu; Shaofeng Cai, National University of Singapore, Singapore, shaofeng@comp.nus.edu.sg; Tien Tuan Anh Dinh, Deakin University, Australia, anh.dinh@deakin.edu.au; Yuncheng Wu, Renmin University of China, China, wuyuncheng@ruc.edu.cn; Yeow Meng Chee, Singapore University of Technology and Design, Singapore, ymchee@sutd.edu.sg; Gang Chen, Zhejiang University, China, cg@zju.edu.cn; Beng Chin Ooi, Zhejiang University, China, ooibc@zju.edu.cn.



ACM 2836-6573/2026/6-ART211
https://doi.org/10.1145/3802088

can be based on the transaction [41, 42, 44, 55], or on the data [25, 39, 45, 46, 50]. The selection of CC algorithm is either heuristic [16, 25, 41, 42, 44, 45] or based on learning [46, 50]. We note that these works do not perform well for dynamic workloads that can drift over time. In particular, they require manual, time-consuming changes to the classification and algorithm selection when there are drifts. CormCC [45] and ACC [46] propose dynamic mechanisms that reduce the cost of manual changes, but they only support a small number of CC algorithms. Polyjuice [50] uses an evolutionary algorithm to learn the optimal combination of CC algorithms. It learns *CC actions* — e.g., to wait for dependent transactions to reach specific steps, or to expose uncommitted operations — for each data access. However, it does not support general transactions and does not work well for dynamic workloads due to the long optimization time.

Our goal is to design a learned CC algorithm that can achieve high performance across diverse workloads, and is efficient to optimize. More specifically, the algorithm can extract a high degree of concurrency from different workloads, while incurring small overhead. The short optimization time allows it to perform well under dynamic workloads. There are three challenges in designing this algorithm. First, a large number of CC algorithms proposed in the literature are optimized for specific workloads. Thus, for our algorithm to perform well across different workloads, it must incorporate design choices from as many existing CC algorithms as possible. Second, to achieve high performance, the algorithm's overhead must be smaller than the transaction execution time. This is non-trivial because, in modern in-memory databases, transactions can be executed in microseconds (or even in nanoseconds). Third, learning the optimal action for each transaction operation is challenging because most of the learned parameters (such as the wait parameters in Polyjuice) are non-continuous. This lack of locality renders machine learning techniques such as Policy Gradient [43] ineffective, thus algorithms like Polyjuice resort to costly optimization strategies such as evolutionary algorithms and random search.

We present NeurCC, a novel learned CC algorithm that achieves the goal above. It addresses the first challenge by proposing a unified model of existing concurrency control algorithms. In particular, NeurCC models a concurrency control algorithm as a learnable function $\mathcal{F} : \{\mathbf{s}\} \rightarrow \{\mathbf{a}\}$ that maps the database state $\mathbf{s}$ to a set of concurrency control actions $\mathbf{a}$. It captures the existing *classify-then-assign* approaches: $\mathbf{s}$ represents the features used for classification, $\mathbf{a}$ indicates the actions of the assigned CC algorithm. In NeurCC, a state $\mathbf{s}$ consists of features used by existing adaptive concurrency controls, such as data hotness [25] and the number of dependent transactions in the dependency graph [48]. An action $\mathbf{a}$ consists of a combination of conflict detection and resolution mechanisms. Instead of selecting from existing algorithms, NeurCC decomposes concurrency control into a sequence of conflict detection and resolution actions, which allows for a wide range of CC designs by enabling arbitrary combinations of these actions. The model is more general than existing adaptive concurrency control algorithms, such as Polyjuice [50] and bLDSF [48]. The former only maps data access IDs to pipeline-wait actions, and the latter only maps dependency set sizes to priority-related wait actions. Figure 1 illustrates the CC designs covered by Polyjuice and bLDSF. Specifically, Polyjuice treats all transactions accessing the same data as equals, which could result in suboptimal actions with high abort rates. bLDSF gives higher priority to transactions with more dependent transactions, but it does not consider transaction pipelining, which is essential for efficient transaction execution.

NeurCC addresses the second challenge by making the collection of the state $\mathbf{s}$ and the execution of $\mathcal{F}(\mathbf{s})$ highly efficient. Specifically, it excludes states that are expensive to compute, e.g., hash values [42] and conflict graphs [52], and instead uses states that can be computed in a lock-free manner, such as the number of direct dependencies and the number of executed operations. NeurCC implements $\mathcal{F}$ as an in-database lookup table for the state-action pairs. This table only consumes a small amount of memory, as NeurCC learns an efficient feature selector that reduces the input

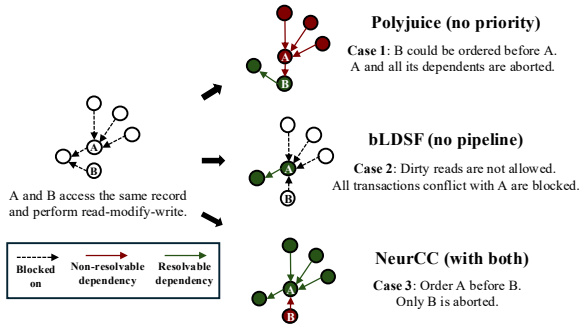


Fig. 1. Example of CC designs covered by different algorithms.

states to only hundreds of values. The whole table fits in the database cache, and the CC actions for an operation can be determined by a single table lookup, which only costs hundreds of CPU cycles.

NeurCC addresses the third challenge in three steps. First, it reduces the cost of evaluating the function's performance. Previous works evaluate the function's performance by loading the lookup table into the database and monitoring the system performance. This process takes seconds to minutes [50]. NeurCC adopts a Bayesian optimization approach that trains a surrogate model for predicting the system performance given an action lookup table. By sampling from this surrogate model, it can quickly explore the search space and guide the optimization process to select the next function to evaluate. Second, NeurCC re-formulates the original learning task of pipeline-wait actions as an optimal conflict graph learning task that exhibits locality and therefore is amenable to common learning techniques. This reformulation is important since learning the pipeline-wait actions directly is challenging due to the lack of locality around the optimal combination of them. In particular, a small modification to an action could lead to dependency cycles and significant performance reduction. NeurCC uses an efficient algorithm similar to genetic search for this conflict graph learning task. Third, NeurCC speeds up the learning process by prioritizing the actions that yield higher improvements. Specifically, it prioritizes fine-tuning wait-related actions that can bring higher performance improvements over actions like timeouts and back-off durations.

We compare NeurCC against state-of-the-art CC algorithms, including the recent learned CC algorithms [45, 50]. The results demonstrate that NeurCC consistently outperforms the baselines across diverse workloads. The algorithm achieves up to 3.32× higher throughput and 11× faster optimization time than the state-of-the-art learned baseline, Polyjuice [50], making it more robust against workload drifts. NeurCC has been integrated into our AI-powered data system NeurDB [2, 34, 61] as the concurrency control mechanism.

In summary, we make the following contributions:

- We present NeurCC, a novel learned concurrency control algorithm that achieves high performance across diverse workloads. The algorithm is quick to optimize, allowing it to adapt quickly to workload drifts.
- We propose a novel model of concurrency control, which is a function mapping database states into conflict-handling actions. This model captures a large number of existing CC algorithms.
- We design an efficient learning algorithm that quickly finds a high-performance concurrency control model.
- We conduct extensive evaluation of NeurCC, comparing it against five state-of-the-art CC algorithms, namely 2PL [38], Silo [49], CormCC [45], Polyjuice [50], and IC3 [52]. The results

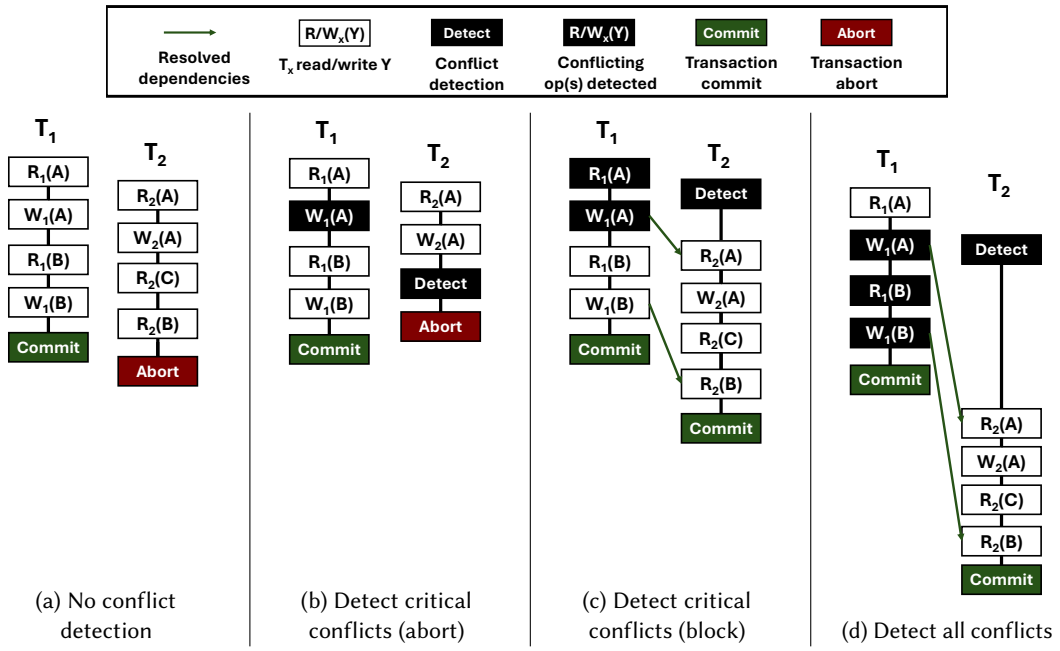


Fig. 2. Examples of different conflict detection approaches. T_2 has RW conflicts with T_1 on A and B.

show that NeurCC achieves high throughput and low optimization time. Its throughput is up to 3.32×, 4.38×, and 4.27× higher than that of Polyjuice, 2PL, and Silo, respectively.

The remainder of the paper is structured as follows. Section 2 discusses the design space of concurrency control algorithms. Section 3 provides an overview of NeurCC. Section 4 details how NeurCC performs fine-grained concurrency control via an optimized action lookup table. Section 5 describes the learning process. Section 6 presents the experimental evaluation. Section 7 discusses related works, and Section 8 concludes.

2 Concurrency Control Design Space

In this section, we first describe the design space of concurrency control (CC) algorithms. We then discuss how existing algorithms fit into the space, and identify the gaps representing new opportunities.

A transaction is modeled as a sequence of data access operations, each being either a read or a write. CC algorithms allow transactions to be executed concurrently and correctly [3, 4, 6, 7, 11–13, 15, 17, 21, 23, 25, 27, 29–31, 38–42, 45–47, 49–52, 55, 61]. To ensure correctness, a CC algorithm employs *CC actions* — algorithmic steps that ensure safe interleaving of data access operations among concurrent transactions. In the following, we describe the design space of CC actions consisting of two design dimensions: conflict detection and conflict resolution. A CC algorithm can use one or multiple actions from this space.

2.1 Conflict Detection

A conflict occurs when two operations from two concurrent transactions access the same data object, and one of the accesses is a write. Existing CC algorithms use different approaches to detect conflicts, based on their assumptions about the distribution and probability of conflicts

occurring during transaction execution. We identify three design choices for conflict detection: no detection, detecting critical conflicts, and detecting all conflicts. The first option optimistically assumes that there are no conflicts and therefore implements no conflict detection. The third option pessimistically assumes many conflicts and therefore detects them as soon as possible to avoid costly aborts. The second option is in between: it detects some critical conflicts while ignoring others.

We consider two transactions T_1 and T_2 executing concurrently. They both perform read/write on data object A and B. Figure 2 illustrates examples where T_2 has Read-Write (RW) conflicts with T_1, and T_2 uses different conflict detection approaches.

No detection. The CC algorithm assumes conflicts are rare and performs no conflict detection during execution. It relies on validation at commit time for correctness. The optimistic concurrency control (OCC) and its variants follow this approach. In particular, each transaction keeps its writes locally (as opposed to exposing them to other transactions), and assumes no other concurrent transactions will update its read data or read its uncommitted writes. The algorithm allows transaction execution to proceed without the overhead of conflict detection but incurs overhead (in aborts) when there are conflicts. Figure 2a shows an example where T_2 aborts during the final validation because T_1 updated its previously read A and B.

Detect critical. The CC algorithm avoids the cost of aborts by identifying conflicts that are likely to cause aborts based on heuristic rules. They are called *critical* conflicts, which may lead to validation failure or to dependency cycles. Upon detection, the algorithm immediately aborts the transaction (early validation), e.g., ASOCC [25] and SSI [12], or reorders operations to allow conflicting transactions to commit together (pipeline-wait), e.g., Bamboo [15] and IC3 [52]. In Figure 2b, T_2 detects that T_1 updated the previously read A during early validation and immediately aborts without executing to the end. In Figure 2c, T_2 detects that the immediate read of A would form a dependency cycle between T_1 and T_2. It then reorders its read of A to occur after T_1's write of A, thus breaking the cycle.

Detect all. The CC algorithm detects all conflicts that may lead to transaction aborts. The classical 2PL and its variants [4, 7, 38] adopt this approach. Figure 2d shows an example in which this approach causes T_2 to block for a long time, which increases transaction latency. Furthermore, the detection algorithm introduces some overhead because it has to find all conflicts.

2.2 Conflict Resolution

Once a conflict is detected, the concurrency control algorithm resolves it by coordinating accesses from the conflicting transactions. We identify two design choices for conflict resolution: timeout and priority.

Timeout. When the CC algorithm detects a conflict, it blocks the current transaction for a specific duration (a timeout), waiting for the conflicting operations to end in this duration, and aborts it when the timeout is exceeded. A timeout of 0 means that the transaction aborts immediately [7], which assumes pessimistically that the transaction will eventually abort. A large timeout, or timeout of ∞, means the transaction is blocked. In particular, the transaction is either put on a wait list associated with the data object, or is entering a spinning loop until the data object is available. This approach is implemented in lock-based and pipeline-wait-based CC algorithms, e.g., 2PL and IC3. The blocking overhead can be significant, especially when the transaction eventually aborts.

Priority. When multiple transactions are blocked on the same data object, the algorithm may order them based on their priorities. When the object becomes available, the highest-priority transaction

is woken up first. Most existing CC algorithms either do not consider priority [17, 49, 50], or use simple FIFO or timestamp priority [4, 15, 38, 60]. Recent works make heuristic priority assignment to conflicting transactions [18, 19, 48, 57]. For example, bLDSF [48] gives higher priorities to transactions with higher numbers of dependent transactions. Other algorithms [18, 19] consider transaction age and the number of held locks when assigning priorities.

Table 1. Comparison of state-of-the-art concurrency control algorithms and NeurCC in the design space. Timeout is omitted because it is supported by all algorithms. Tuned priority refers to either heuristic or learned priority assignments.

		Resolve w/o tuned priority		Resolve with tuned priority	
	No detection	Detect critical	Detect all	Detect critical	Detect all
OCC [3], Silo [49]	✓				
2PL [7], TSO [5, 22], MVCC [54]			✓		
Bamboo [15], Rebirth-Retire [60], EWV [11], IC3 [52]		✓			
bLDSF [48], VATS [18]					✓
Polaris [57]	✓				✓
MCC [55], Tebaldi [41]		✓	✓		
ACC [46], TxnSails [62], C3 [42], CormCC [45], HDCC [16]	✓		✓		
ASOCC [25], Polyjuice [50]	✓	✓	✓		
NeurCC	✓	✓	✓	✓	✓

2.3 State-of-the-Art Concurrency Control Algorithms

Table 1 compares state-of-the-art concurrency control algorithms within the design space. We omit timeout from the table because it is supported by all algorithms. It can be seen that some algorithms, such as OCC and 2PL, support only one type of CC actions. Other algorithms, such as Polyjuice [50] and ASOCC [25], support multiple CC actions, with which the algorithm can fine-tune conflict handling actions based on the workload to extract more concurrency.

The table reveals several gaps in the design space, i.e., CC actions that are not covered by existing algorithms. For example, no existing algorithms consider priority-based conflict resolution in combination with detecting critical conflicts. Most algorithms, including the ones that support multiple CC actions, do not consider tuned transaction priorities.

3 NeurCC Overview

In this section, we present an overview of NeurCC, a novel learning-based concurrency control algorithm that achieves high performance across diverse workloads. It supports the complete set of CC actions shown in Table 1, by capturing existing concurrency control algorithm designs via a learnable function. In the following, we describe the learning function, and the workflow that enables its efficient execution and optimization. We summarize the notations in Table 2.

3.1 Concurrency Control As a Learnable Function

We model concurrency control algorithm as a function $\mathcal{F}$ that maps the current database state $\mathbf{s}$ to a combination of conflict detection and conflict resolution actions $\mathbf{a}$.

$$\mathcal{F}(\mathbf{s}) = \mathbf{a}. \tag{1}$$

The state $\mathbf{s}$ comprises features used in various CC algorithms, such as transaction access ID, transaction type, and data hotness. The actions $\mathbf{a}$ include conflict detection actions, timeout values,

Table 2. Summary of notations

T	current transaction
op	current operation of the current transaction
$\mathbf{x}$	vector with all features
$\mathbf{s}$	database state
$\mathbf{a}$	CC actions for the current operation
$\mathcal{F}$	function
$\mathcal{E}$	feature selector
Score($\mathcal{F}$)	system performance using $\mathcal{F}$

and pipeline-wait actions. Before executing an operation, the algorithm collects the current database state $\mathbf{s}$, computes $\mathcal{F}(\mathbf{s})$, and then executes the resulting actions $\mathbf{a}$.

$\mathcal{F}$ captures many existing CC algorithms, including Polyjuice, ASOCC, 2PL, bLDSF, OCC, and IC3. We demonstrate this with two examples of 2PL (wait-die) and ASOCC. Let t_{req} and t_{owner} be the timestamps of the current transaction and of the oldest lock owner. Let *hotness* be the hotness level of the data object being accessed. 2PL (wait-die) and ASOCC can be modeled as follows:

$$\mathcal{F}_{\text{2PL}}(t_{req}, t_{owner}) = \begin{cases} (\textit{detect all}, 0, \varnothing) & t_{req} > t_{owner} \\ (\textit{detect all}, \infty, \textit{timestamp pri.}) & \text{otherwise} \end{cases}$$

$$\mathcal{F}_{\text{ASOCC}}(\textit{hotness}) = \begin{cases} (\textit{no detection}, 0, \varnothing) & \textit{hotness} = \text{cold} \\ (\textit{detect all}, \infty, \varnothing) & \textit{hotness} = \text{hot} \\ (\textit{detect critical}, 0, \varnothing) & \text{otherwise} \end{cases}$$

Each action consists of a conflict detection action, a timeout value, and a transaction priority ($\varnothing$ represents no priority). In these examples, t_{req}, t_{owner}, and *hotness* are the state $\mathbf{s}$. The output of $\mathcal{F}_{\text{2PL}}$ and $\mathcal{F}_{\text{ASOCC}}$ are then executed by the algorithm. In particular, 2PL (wait-die) aborts the transaction when t_{req} is higher than t_{owner}; otherwise, it blocks the transaction and wakes it up later in the timestamp order. ASOCC ignores conflicts for cold data. It detects all conflicts and blocks the transaction for hot data. In other cases, it aborts the transaction when there are critical conflicts.

We define the problem of finding $\mathcal{F}$ that extracts the most concurrency across diverse workloads as an optimization problem. Let $\mathbf{S}$ be the set of possible input states $\mathbf{s}$, Score($\mathcal{F}$) be the system performance[1] using $\mathcal{F}$, and $\mathbf{A}$ be the set of possible output actions $\mathbf{a}$. The optimization objective is defined as:

$$\text{maximize} \quad \text{Score}(\mathcal{F}) \tag{2}$$

$$\text{subject to} \quad \forall \mathbf{s} \in \mathbf{S},\ \mathbf{a} = \mathcal{F}(\mathbf{s}) \in \mathbf{A}. \tag{3}$$

NeurCC starts the optimization process when it detects a workload drift. Specifically, there is a background worker, named *optimizer*, that continuously monitors the system's *current* throughput, i.e. the current number of committed transactions per second (TPS). The *optimizer* tracks the throughput regardless of whether the system is saturated. Optimization is triggered when the relative change in throughput exceeds a predefined threshold. In the current design, the threshold value is determined empirically, e.g., 10%. The value is chosen to achieve reliable detection while avoiding unnecessary optimization triggers. We discuss this threshold in Section 6.2, and provide more details in the extended version [35]. We note that more complex workload drift detection techniques (e.g., two-sample-test [53]) could also be used. Since our main focus is on the learning

[1]We use throughput as the performance metric, similar to previous works [46, 50].

of concurrency control, we leave the design and integration of advanced drift detectors to future work.

3.2 NeurCC Workflow

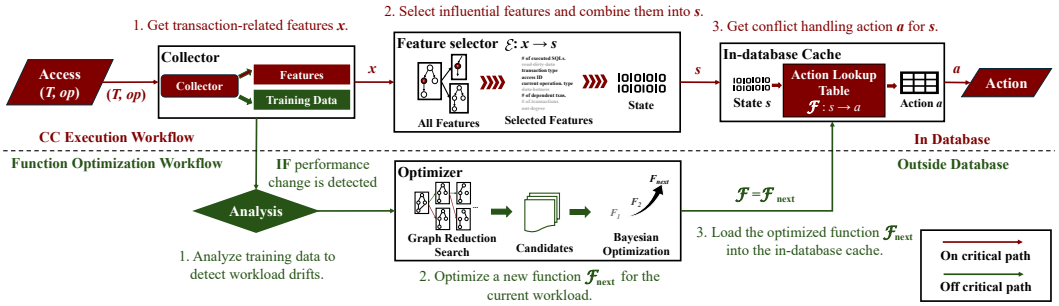


Fig. 3. NeurCC's concurrency control (CC) execution workflow and function optimization workflow.

Figure 3 shows the main components of NeurCC: a *collector* that tracks real-time database states, a *feature selector* $\mathcal{E}$ that maps the collected states into a state vector $\mathbf{s}$, an *in-database cache* that maintains the function $\mathcal{F}$ in the form of an action lookup table, and an *optimizer* that detects workload drifts and optimizes the function upon detection.

NeurCC operates across two workflows: the *CC execution workflow* and the *function optimization workflow*. The former is on the critical path of transaction execution. It is invoked before every data access operation to select the optimal actions. This workflow takes the current operation op and transaction T as input, and outputs a vector of actions $\mathbf{a}$ for the transaction manager to execute. The transaction manager first sends (T, op) to the collector, which collects database states based on T and op, then combines them into a feature vector $\mathbf{x}$. Next, the feature selector $\mathcal{E}$ selects important features from $\mathbf{x}$ and generates a state $\mathbf{s}$. Finally, the concurrency control actions $\mathbf{a}$ are returned from the lookup table, i.e., $\mathbf{a} = \mathcal{F}(\mathbf{s})$. This workflow is efficient such that it can be executed at per-operation granularity. We describe it in more detail in Section 4.

The second workflow is off the critical path of transaction execution. It maintains high performance under workload drifts by monitoring the system performance and optimizing the action lookup table when necessary. In this workflow, the function $\mathcal{F}$ and the corresponding system performance Score($\mathcal{F}$) are collected periodically. The data is analyzed to detect workload drifts based on performance changes. When a drift is detected, NeurCC starts optimizing the function for the current workload. The optimization process runs with a pre-defined time budget t_{max}, at the end of which the new function $\mathcal{F}_{next}$ is loaded into the database cache as a new action lookup table. We describe this workflow in more detail in Section 5.

3.3 Discussion

NeurCC evaluates the agent function at per-operation granularity. This is because important CC mechanisms, such as dirty-read visibility and pipeline waits, work at the level of individual operations (as opposed to at the level of individual transaction). We note that coarse-grained decisions can be beneficial, for example, in enabling scheduling operations before execution (as in deterministic CC). Integrating such coarse-grained decisions with our fine-grained agent function is left as future work.

Table 3. Features collected by NeurCC. T denotes the current transaction.

Feature	Definition
# of executed SQLs	Number of SQLs already executed by T.
Read dirty data	If T has read uncommitted dirty data from others.
Transaction type	Identifier of the transaction template to which T belongs (e.g., `NewOrder`, `Payment`).
Access ID	Identifier of the currently executing SQL.
Operation type	The current operation type (read/write).
Data hotness	Contention level of the accessed record (*cold*, *warm*, *hot*), derived from data access frequency and data correlation [25].
# of dependent txns.	Number of transactions T depends on.
# of transactions	Number of running transactions.
Out-degree	Number of transactions that depend on T.

The current design of NeurCC targets CC paradigms based on per-operation conflict handling, which keeps the state and action space of the agent function small and efficient to optimize. We note that NeurCC captures some transaction-chopping behaviors through its pipeline-wait actions (i.e., w_i and *expose*), and it can reproduce the behavior of algorithms like IC3 [52] that uses transaction chopping. Other paradigms, such as deterministic CC, sagas, or timestamp-ordering, entail more complex state and action representations (e.g., actions that schedule transactions before execution). Supporting them would significantly expand the agent function's state and action space, thus requiring new learning algorithms, which we leave for future work.

4 NeurCC Execution

In this section, we describe NeurCC's execution workflow which lies on the critical path of transaction execution. We first discuss what database states NeurCC collects, and how it selects important states as input to the function $\mathcal{F}$. We then present the detailed concurrency control algorithm that performs conflict handling based on the output of $\mathcal{F}$.

4.1 State Collection and Selection

Before executing a transaction operation, NeurCC first computes a vector $\mathbf{s}$ that captures the current, relevant database states (Figure 4). The vector $\mathbf{s}$ is important as it determines the quality of CC actions generated by $\mathcal{F}$. There are two challenges in computing $\mathbf{s}$. First, it must be efficient since it is on the critical path of transaction execution. Modern in-memory databases can commit a transaction in microseconds, therefore $\mathbf{s}$ should be computed in order of nanoseconds. Second, $\mathbf{s}$ must contain relevant database states so that CC actions returned by $\mathcal{F}$ can extract a high degree of concurrency, while remaining small to minimize storage overhead and to speed up the optimization of $\mathcal{F}$.

NeurCC employs two techniques to address the challenges above. The first is to use a fixed set of lightweight states (or features) that can be efficiently captured across transactions, in a lock-free manner. NeurCC collects nine features, which are listed in Table 3 and can be grouped into transaction features and graph features. These features have been shown to be effective at capturing transaction and dependency graph characteristics [18, 25, 41, 48, 50, 52]. More importantly, they are common features readily available in transactional databases, that is, they are not specific to any concurrency control algorithm.

The second technique is to use a feature selector $\mathcal{E}$ that selects influential features to achieve a balance between the quality of the optimized function and the optimization time. NeurCC provides a default $\mathcal{E}$ that achieves good performance, which the user can improve at some cost, by running the optimization algorithm described in Section 5.4.

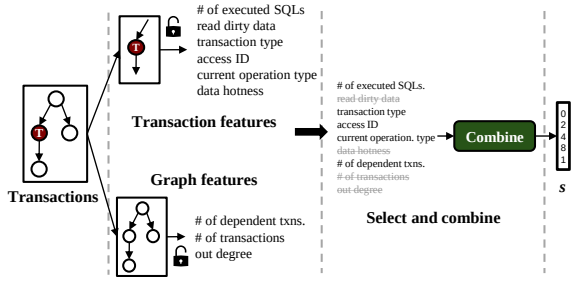
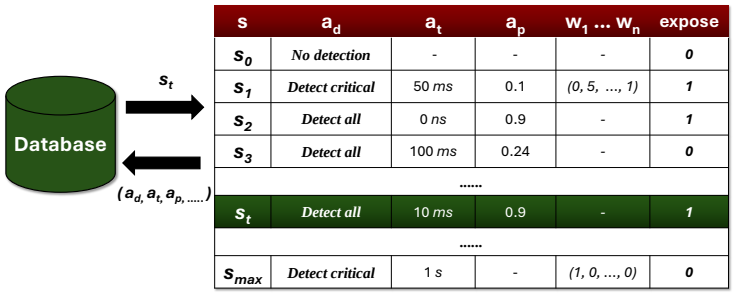


Fig. 4. NeurCC's state collection and selection process.

Before a transaction accesses a data record, NeurCC collects the current database state as a vector **s**. Consider ASOCC as a running example. This vector includes data hotness feature, thus access to records with different contention levels is mapped to different states, which enables the selection of appropriate conflict-handling actions as in ASOCC.

4.2 Action Function

After computing **s**, NeurCC executes $\mathcal{F}(\mathbf{s})$ to get the concurrency control actions **a** for the current transaction operation. The main challenge in designing $\mathcal{F}$ is to ensure both efficiency and effectiveness. Specifically, the function must be quick to evaluate, since it is on the transaction's critical path. Furthermore, the output actions must be able to extract high degrees of concurrency across different workloads. For efficiency, NeurCC implements $\mathcal{F}$ as a learnable action lookup table, reducing the function evaluation to a single table lookup. Although implementing $\mathcal{F}$ as a machine learning model can approximate the state-action mapping more accurately, the inference cost is too high for concurrency control. In contrast, the lookup table can return the actions in hundreds of CPU cycles.



s	a_d	a_t	a_p	$w_1 \dots w_n$	expose
s_0	*No detection*	-	-	-	**0**
s_1	*Detect critical*	50 *ms*	0.1	*(0, 5, ..., 1)*	**1**
s_2	*Detect all*	0 *ns*	0.9	-	**1**
s_3	*Detect all*	100 *ms*	0.24	-	**0**
.....					
s_t	*Detect all*	10 *ms*	0.9	-	**1**
.....					
s_{max}	*Detect critical*	1 *s*	-	*(1, 0, ..., 0)*	**0**

Fig. 5. NeurCC function implemented as a lookup table.

To achieve high performance across diverse workloads, $\mathcal{F}$ learns combinations of design choices from existing CC algorithms optimized for different workloads. An output of $\mathcal{F}$ contains a conflict

detection mechanism, which can be either *no detection*, *detect critical*, or *detect all*. It also contains a timeout value and a transaction priority for conflict resolution. Figure 5 shows an action lookup table, where the first column is the input state and the remaining columns make up the CC actions. Let n denote the number of transaction types. The table has the following schema:

$$\langle \mathbf{a}_d, \mathbf{a}_t, \mathbf{a}_p, (w_1, \ldots, w_n, expose) \rangle \tag{4}$$

$$\text{where} \quad \mathbf{a}_d \in \{no\ detection, detect\ critical, detect\ all\} \tag{5}$$

$$\mathbf{a}_t \in [0, +\infty), \quad \mathbf{a}_p \in [0, 1] \tag{6}$$

$$w_i \in \mathbb{N}, \quad expose \in \{0, 1\} \tag{7}$$

$\mathbf{a}_d$, $\mathbf{a}_t$, and $\mathbf{a}_p$ are the conflict detection method, timeout, and transaction priority, respectively. $(w_1, \ldots, w_n)$ represents the rule for identifying critical conflicts (as described in Section 2.1). Specifically, w_i means that T considers the first w_i accesses of a dependent transaction with type i as critical conflicts, thereby preventing the formation of dependency cycles among concurrent transactions that might read dirty data from each other. $expose$ determines if T makes its accesses visible to other transactions after executing op. NeurCC also implements several optimizations for the function. Due to space constraints, we discuss them in the extended version [35].

In the ASOCC example, NeurCC selects *no detection* for cold data, which corresponds to optimistic execution. It selects *detect all* for hot data, forcing conflict detection for operations accessing highly contended records. It selects *detect critical* for warm data, enabling early validation to proactively abort transactions that are likely to fail.

4.3 Transaction Execution

After determining the CC actions, NeurCC proceeds to execute the transaction operations. Similar to OCC, NeurCC does not guarantee all conflicts are detected and resolved during execution. Instead, it performs a validation before committing the transaction, once all operations are executed. We elaborate on these steps below.

4.3.1 Execution. A transaction in NeurCC has the following fields:

- $T.rw$ and $T.ws$: the local read and write sets.
- $T.dep$: the set of transactions that T depends on.
- $T.type$: the transaction type of T.
- $T.validate$: operations of T that have been executed but not yet validated.

For each operation op, NeurCC maintains the following information:

- $op.tx$: the transaction that contains op.
- $op.key$: the unique key of the data object accessed by op.
- $op.type$: the operation type (read/write).
- $op.clean$: whether op reads the latest committed (clean) data.
- $op.priority$: the priority of op when resolving its conflicts with other operations.

Algorithm 1 describes how NeurCC executes an operation. It performs early validation to proactively abort transactions that will fail later (line 14). It also performs early validation before exposing a write (line 21), where the failure of a transaction T could lead to the cascading abort of all transactions depending on it. The algorithm does not perform early validation for "detect all" because doing so incurs additional overhead and can restrict the search space. For example, it would prevent NeurCC from replicating 2PL, which does not incur the overhead of early validation checks.

NeurCC does not outperform 2PL when all the actions are detect-all, nor does it introduce a new lock-based protocol. However, we note that despite the extra validation cost compared to classical

Algorithm 1: Execute an operation with learned actions

```
1  Function Execute(T, op):
2      a ← GetCC(T, op) // compute s and F(s)
3      op.priority ← a_p , op.tx ← T
4      if a_d = no detection then
5          op.clean = True, S ← ∅
6      else if a_d = detect all then
7          S ← all operations from conflict transactions
8      else if a_d = detect critical then
9          if IsStoredProcedure() then
10             S ← ∅                    // Conflicts arising from pipeline waits.
11             for T' ∈ T.dep do
12                 bar ← w_{T'.type}
13                 S ← S ∪ {the first bar-th accesses in T'}
14         else if EarlyValidation(T) = Abort then
15             return Abort, ∅
16     S ← S − {op' | op' ∈ S and op'.priority < op.priority}
17     if Not all op' ∈ S finish within timeout a_t then
18         return Abort, ∅
19     result ← SafeExecute(T, op)
20     if IsStoredProcedure() and expose = 1 then
21         if EarlyValidation(T) = Abort then
22             return Abort, ∅
23         else
24             for op' ∈ T.ws do Lock tuple(op'.key)
25             for op' ∈ dirty reads on tuple(op.key) do
26                 T.dep ← T.dep ∪ {op'.tx}
27             for op' ∈ T.ws do
28                 Append op' to the version chain as dirty data
29                 Unlock tuple(op'.key)
30     return Succeed, result
31 Function SafeExecute(T, op):
32     if op.type = read then
33         if op.clean then
34             op' ← latest committed data on tuple(op.key)
35         else
36             op' ← latest dirty data on tuple(op.key)
37             T.dep ← T.dep ∪ {op'.tx}
38         T.rs ← T.rs ∪ {op'}
39         T.validate ← T.validate ∪ {op'}
40         return op'
41     else
42         T.ws ← T.ws ∪ {op}
43         return ∅
```

algorithms such as 2PL, the performance gained from the learned concurrency control actions outweighs this cost and leads to higher throughput, as will be illustrated in Section 6. Furthermore, when all actions in the agent function are *detect all*, NeurCC can skip validation mechanisms, including the tracking of $T.rs, T.ws, T.validate$ and the transaction validation at commit time, to further reduce the overhead over 2PL.

The algorithm distinguishes between stored procedure and interactive transaction modes. In the former, all the data accesses are known prior to transaction execution. In the latter, they are not known before transaction execution. For interactive transactions, the algorithm disallows actions that could violate correctness. Such actions include dirty reads and partial retry, which are disallowed because the behavior of the conflicting transaction is unknown, and because the user has already seen the outputs of some operations.

Given an operator op, the algorithm first invokes $GetCC(T, op)$ to get the action $\mathbf{a}$ (line 2). Next, it executes conflict detection and stores the conflicts in S. This step is skipped if $\mathbf{a}_d$ is *no detection*, and the algorithm enforces op to read clean data (lines 4–5). If $\mathbf{a}_d$ is *detect critical*, it performs early validation to check if previously read data have changed (line 14) and aborts T if the validation fails. The conflicts with lower priorities than that of op are removed from S. The remaining are resolved with the given timeout $\mathbf{a}_t$ (lines 16–17). If not all the conflicts in S are resolved within $\mathbf{a}_t$, T is aborted (line 18). Otherwise, the algorithm proceeds to read the latest committed data (line 34), or to write to its write set $T.ws$ (line 42).

For stored procedure transactions, the algorithm allows dirty reads. In particular, it uses $w_1, \ldots, w_n$ to detect critical conflicts before dirty reads, uses $expose$ to decide whether to make uncommitted write visible to other transactions, and records dependencies caused by dirty reads in $T.dep$ (lines 26 and 37). Each data object has a version chain containing dirty data. The execution when $\mathbf{a}_d$ is *no detection* or *detect all* is the same as that of interactive transactions. When $\mathbf{a}_d$ is *detect critical*, it detects critical conflicts between T and its dependent transactions in $T.dep$ (lines 11–13). In particular, for every $T' \in T.dep$, it adds the first $w_{T'.type}$ accesses of T' to S. Once all the conflicts in S are resolved, it lets op read the data at the tail of the corresponding version chain, and updates $T.dep$ to track the read-write dependency (lines 36–37). After executing op (line 19), if $expose = 1$, it performs early validation and aborts T if the validation fails. If the validation succeeds, it exposes the uncommitted data in $T.ws$ by appending them to the corresponding version chains.

We use the ASOCC example to illustrate how a transaction is executed with NeurCC. We assume the transaction is in the interactive transaction mode. Before executing an operation, NeurCC first computes the input state $\mathbf{s}$ and retrieves the corresponding actions by evaluating $\mathcal{F}(\mathbf{s})$ (line 2). If the selected action is *no detection* (i.e., the accessed data is cold), NeurCC executes it optimistically without blocking (line 4). If the action is *detect all* (i.e., the accessed data is hot), NeurCC performs conflict detection before execution (line 6), and upon detecting a conflict, it blocks or abort the transaction according to the selected timeout (line 17). If the action is *detect critical* (i.e., the accessed data is warm), NeurCC performs early validation (line 8), and if validation fails, it aborts the transaction immediately.

4.3.2 Validation and commit. NeurCC performs a four-step validation at commit time, similar to OCC. First, it waits for all transactions in $T.dep$ to complete, and aborts if any dependent transaction aborts. Second, it locks the private write set to avoid concurrent reads of the data in the set. Third, it verifies that data in the read set matches the latest committed versions, and aborts if the verification fails. Finally, it removes dirty data from T on the corresponding version chains and unlocks the write set. The latest committed data are updated with those in the write set. We present detailed validation and commit algorithms in the extended version [35].

4.3.3 Correctness. Similar to OCC, NeurCC ensures correctness (i.e., serializability) by performing validation before committing the transaction. A detailed proof of correctness is provided in the extended version [35].

5 Learning Function and Feature Selector

In this section, we describe how NeurCC learns the function $\mathcal{F}$. We observe that different actions are amenable to different optimization techniques. More specifically, model-based optimizations are suitable for some actions, whereas search-based techniques are suitable for others. Furthermore, different actions have different impacts on the overall throughput, and thus the order in which they are optimized can affect convergence speed. Based on these observations, we decompose $\mathcal{F}$ into three sub-functions: $\mathcal{F}_T$, $\mathcal{F}_D$, and $\mathcal{F}_P$, and optimize them separately. Their outputs are combined to obtain $\mathbf{a} = \langle \mathbf{a}_d, \mathbf{a}_t, \mathbf{a}_p, (w_1, \ldots, w_n, expose) \rangle$. $\mathcal{F}_T$ controls the timeout values, corresponding to $\mathbf{a}_t$. $\mathcal{F}_D$ controls the conflict detection and priority, corresponding to $(\mathbf{a}_d, \mathbf{a}_p)$. $\mathcal{F}_P$ controls pipeline-wait actions, corresponding to $(w_1, \ldots, w_n, expose)$.

Learning the optimal function requires evaluating the performance of $\mathcal{F}$, i.e., Score($\mathcal{F}$). Existing works [45, 46, 50] load the function into the running database and use the system throughput as Score($\mathcal{F}$). This approach is time-consuming, because loading and running the system takes seconds[2] to complete, and they need to be repeated at every optimization step, adding significant overheads to the learning process [50]. In addition, Score($\mathcal{F}$) is non-smooth due to its discrete input $\mathcal{F}_P$, which makes it difficult to optimize $\mathcal{F}_P$ with model-based optimization methods. NeurCC addresses the aforementioned limitations of the existing approaches for evaluating Score($\mathcal{F}$) in two ways. First, it employs an online-trained surrogate model to minimize the number of performance evaluations (Section 5.1). Second, it uses a graph reduction search algorithm to effectively optimize $\mathcal{F}_P$ (Section 5.2).

5.1 Learning Function Via Surrogate Model

As Score($\mathcal{F}$) is costly to evaluate, NeurCC treats it as a black-box [8, 14, 37] and adopts a Bayesian optimization approach. Bayesian optimization can efficiently identify high-performing functions with a small number of evaluations, thereby reducing the cost. In particular, NeurCC employs a probabilistic surrogate model, i.e., the Gaussian process model, which is trained online to approximate Score($\mathcal{F}$). This model is differentiable for efficient optimization and is well-suited as a surrogate model because it predicts both the mean and variance of the performance, which balances exploration of new areas in the function space with exploitation of known high-performing areas.

The surrogate model guides the optimization process by identifying the most promising function, i.e., the one expected to yield the highest performance improvement, for subsequent deployment and evaluation. Specifically, during an optimization step, the surrogate model is first fitted to the historical data collected since the last workload drift, which optimizes its mean vector and covariance matrix to maximize the data likelihood. Next, for an unevaluated function $\mathcal{F}$, the fitted model predicts the mean $\mu(\text{Score}(\mathcal{F}))$ and variance $\sigma(\text{Score}(\mathcal{F}))$ of its performance. After that, the Upper Confidence Bound (UCB) acquisition function computes a score for $\mathcal{F}$ as $\text{UCB}(\mathcal{F}) = \mu(\text{Score}(\mathcal{F})) + \lambda\sigma(\text{Score}(\mathcal{F}))$, where λ is a trade-off parameter. It is set to 2.576 in our experiments, corresponding to the 99th percentile of a standard normal distribution to achieve an effective balance between exploration and exploitation. Note that the UCB score is differentiable with respect to the parameters of $\mathcal{F}$, and therefore NeurCC can explicitly optimize these parameters to maximize the UCB score using the L-BFGS-B [26] algorithm, which produces $\mathcal{F}_{next}$, the function with the

[2]This process takes 3.68 seconds on average under our default experimental setting, as detailed in Section 6.1: TPCC workload, 16 threads, one warehouse.

highest expected performance. Once $\mathcal{F}_{next}$ is identified, it is evaluated in the database system, and its actual performance score is recorded. Finally, the new data point of $\mathcal{F}_{next}$ and its score are added to the historical data, and the surrogate model is updated. This process is repeated, progressively refining predictions of Score($\mathcal{F}$) and identifying better functions over time.

5.2 Pipeline-Wait Action Optimization via Learned Conflict Graph

The Bayesian optimization technique in Section 5.1 can effectively optimize $\mathcal{F}_T$ and $\mathcal{F}_D$, but optimizing $\mathcal{F}_P$ is more challenging, especially for complex workloads like TPC-C that involve many potential dependency cycles. We observe empirically that during the optimization of Score($\mathcal{F}_T$, $\mathcal{F}_D$, $\mathcal{F}_P$), the function exhibits poor locality with respect to $\mathcal{F}_P$ around the optimal solutions. Even minor changes to $\mathcal{F}_P$ can significantly reduce performance by introducing new dependency cycles that lead to cascading aborts or deadlocks [52]. As a result, the surrogate model trained via Bayesian optimization fails to approximate Score($\mathcal{F}_T$, $\mathcal{F}_D$, $\mathcal{F}_P$) accurately, because the optimization inherently assumes smoothness and locality in the approximated function.

To address the challenge of global optimization of $\mathcal{F}_P$, NeurCC learns an optimal conflict graph instead of learning these actions directly. The optimal conflict graph is the learned conflict graph from which the pipeline-wait actions that maximize the performance score are derived. This reformulation simplifies the learning process and improves optimization performance. A conflict graph is a graph generated from the workload, where each type of transaction access (e.g., a line of SQL statement) is represented as a node. If two operations have potential conflicts (e.g., read-write conflicts), an edge is added between the corresponding nodes. NeurCC uses SC-graph, a conflict graph widely adopted in existing CC algorithms [32, 50, 52]. These algorithms construct a static conflict graph and compute pipeline-wait actions by analyzing this graph before workload execution. However, they rely on static conflict graphs, which do not account for dynamic factors such as access skewness. We contend that the conflict graph should be dynamic, with conflict edges added based on certain probabilities. For example, in the TPC-C workload, when there are many warehouses and few concurrent users, the *payment* transaction's update WAREHOUSE table access is unlikely to conflict with the *new order* transaction's read WAREHOUSE table access, and therefore this edge can be deleted. However, when there are few warehouses and many concurrent users, these accesses are more likely to conflict, therefore adding the conflict edge is beneficial.

NeurCC employs a graph reduction search algorithm to efficiently learn the optimal conflict graph. Given an initial static conflict graph, the algorithm keeps track of the top-K best-performing conflict graphs found during the search. Each graph is evaluated by converting it into pipeline-wait actions and evaluating the corresponding function. Specifically, the algorithm first performs conflict graph analysis using an IC3-like algorithm [52] to find potential dependency cycles and to compute the pipeline-wait actions that resolve them. It next updates the current best agent function with the computed pipeline-wait actions. Finally, the algorithm evaluates the updated agent function and uses its performance as the score of the conflict graph. For the details of this algorithm, please refer to [35].

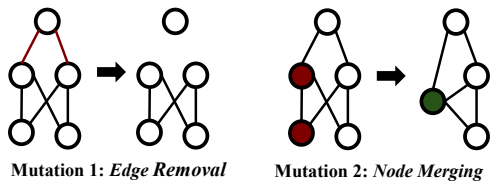


Fig. 6. Two types of graph reduction mutations.

During optimization, the algorithm iteratively improves the conflict graphs by mutating the current graphs to better capture actual conflicts. Each mutation simplifies the graph by edge removal or by merging consecutive nodes (as illustrated in Figure 6). The mutation aims to eliminate unnecessary conflicts that will be resolved during the pipeline-wait actions calculation. Since the mutation reduces the number of nodes or the number of edges, the size of the graph reduces quickly towards the optimal conflict graph. Our extended version [35] provides more details of the graph reduction search, including the exploration of conflict graphs, the calculation of pipeline-wait actions, the computation of the function with optimized pipeline-wait actions, and the optimizations applied during pipeline waits.

The graph reduction search algorithm complements the Bayesian optimization for $\mathcal{F}_P$. The former efficiently explores functions derived from conflict graphs, while the latter is effective at approximating high-performance but sparsely distributed functions. For instance, in the YCSB workload, we observe that a no-pipeline-wait policy can outperform learned pipeline-wait actions due to reduced computational costs associated with conflict detection and dependency tracking. Such a function cannot be identified through graph reduction search, but it can be found quickly by Bayesian optimization within minutes.

5.3 Function Optimization Pipeline

Given a limited time budget, it is important to prioritize the optimization of actions that are more likely to yield significant performance improvements. NeurCC uses an optimization pipeline that transfers knowledge from one optimization stage to the next. It allocates the time budget efficiently across these stages. Recall that the function $\mathcal{F}$ is decomposed into three sub-functions $\mathcal{F}_P$, $\mathcal{F}_D$, and $\mathcal{F}_T$ (see Section 5). The optimization pipeline consists of four stages, each targeting a subset of actions while keeping the remaining actions unchanged from the best-seen function. Each stage leverages the previously evaluated functions and their corresponding performance scores as initial training data for the stage's optimizer.

The first stage optimizes both $\mathcal{F}_P$ and $\mathcal{F}_D$ using the graph reduction search algorithm. These two sub-functions are optimized first because they lead to the largest performance gains, e.g., with improvements of nearly 80% in the TPC-C workload. $\mathcal{F}_P$ and $\mathcal{F}_D$ can be optimized simultaneously, as the *no detection* action and edge removal mutation both assume no conflicts around an operation. This stage starts with the same initial configuration as that of IC3, in which all conflict detection actions are set to *detect critical*. The graph reduction search iteratively refines the conflict graph. If a function input $\mathbf{s}$ maps to conflict graph nodes that are entirely disconnected from others, it indicates that database operations in state $\mathbf{s}$ rarely encounter conflicts. In such cases, we set the conflict detection action to *no detection* to reduce transaction execution costs. We set $K = 4$ to enable quick convergence to an efficient function. The stage ends when no further mutations yield performance improvements, typically within 10 minutes due to the monotonically reducing nature of the mutation process. The details on how we optimize $\mathcal{F}_P$ and $\mathcal{F}_D$ can be found in the extended version [35].

The second stage optimizes $\mathcal{F}_T$ using Bayesian optimization. At this stage, most parameters in $\mathcal{F}_T$ are continuous and exhibit good locality. This stage ends when no further performance improvements are observed after several consecutive evaluations (e.g., 20 evaluations). The third stage is the same as the first stage but with $K = 8$, allowing for deeper exploration of the conflict graph, in order to refine the conflict graph and functions. Finally, the fourth stage fine-tunes all the parameters of $\mathcal{F}$ using Bayesian optimization. This stage runs until the time budget is exhausted.

5.4 Feature Selector Optimization

We define the feature selector $\mathcal{E}$ as a function mapping a feature selection strategy to the system performance achieved with the selected features. Each feature selection strategy is represented as a vector, with each element indicating whether a corresponding feature is included and specifying its transformation type (linear, square root, or logarithmic). The output of $\mathcal{E}$ is a set of input features for $\mathcal{F}$, which is subsequently used during the function optimization. As in Section 5.1, NeurCC uses Bayesian optimization to iteratively propose new feature selection strategies, evaluate them, and refine the surrogate model.

To balance effectiveness and efficiency, the feature selection strategy is optimized to maximize the performance improvement achieved through function optimization within a user-defined time budget. Note that selecting too few features can overly constrain the search space of functions, leading to suboptimal solutions, whereas selecting too many features can expand the search space excessively, making the optimization computationally expensive. Therefore, we balance the two factors by incorporating the time budget into the performance. In our experiments, we run the feature selector optimization for one day, using a function optimization budget of one hour for each feature selection evaluation.

The optimization is costly and is performed only once when the system is deployed. Subsequent changes in system load, contention, or access patterns do not automatically trigger re-optimization. Instead, NeurCC relies on function optimization to handle such workload drifts. However, users can explicitly run the optimization to fine-tune the default selector for specific workloads. Our feature selectors and their configurations are available in the code repository [1].

6 Performance Evaluation

In this section, we evaluate the performance of NeurCC and compare it with state-of-the-art concurrency control (CC) algorithms under diverse workloads. We implement NeurCC in C++ and integrate it into Silo [49], a widely used database system for evaluating CC algorithms. Specifically, we add the following components to the database engine: a collector that captures transaction-related features, a feature selector that selects influential features, an in-database cache that stores the learned functions, and an optimizer that tracks system performance for function optimization and workload drift detection. We integrate Algorithm 1 into the concurrency control code of the database engine. We implement a script that processes optimization requests from the database. The code is available at [1]. Details on NeurCC deployment are included in the extended version [35].

6.1 Experimental Setup

All experiments are conducted on a 24-core Intel(R) Xeon(R) Gold 5317 server with 128GB of memory. The CPU runs at a base frequency of 800 MHz, and a maximum turbo frequency of 3.6 GHz.

Execution modes. NeurCC executes transactions in one of the two modes: stored procedure [15, 24, 33, 41, 42, 50, 52, 56], and interactive transaction [9, 36, 58, 59].

- **Stored procedure mode.** This mode represents how stored procedure transactions are executed by the database engine. In particular, all the transaction's statements are known before execution. As a result, concurrency control algorithms can use transaction type and access ID as features. The former can be assigned to each stored procedure, and the latter can be derived from the statement order, e.g., the line number of the API invocation within the procedure [50]. The transaction only returns results to the user after the transaction commits or aborts, which means it can read uncommitted data during execution and perform partial retries.

- **Interactive transaction mode.** This mode represents how interactive, user-driven transactions are executed. In particular, the transaction's statements (SQL queries) are issued and executed individually. The user waits for the response of each statement before issuing the next. Because the operations are not known in advance, concurrency control algorithms cannot rely on features such as transaction type and access ID. Furthermore, since the results are returned to the user immediately after each operation, the algorithms cannot perform partial retry and dirty read.

Workloads. We conduct experiments on the widely used YCSB and TPC-C workloads. We change three types of workload settings for evaluation: access pattern, contention level, and system load. The workloads and their settings are described below.

- **YCSB-extended.** We extend YCSB to group multiple read/write operations into a single transaction, and simulate varying access patterns by changing hotspot positions. Specifically, we fix the list of read and write operations for all transactions, representing stable operations found in real-world applications [50, 52]. We then vary the positions of hotspots, where hotspot accesses follow a skewed distribution (Zipfian factor $\theta = 1$), while other accesses follow a uniform distribution across all keys (Zipfian factor $\theta = 0$). For this workload, we use 16 threads.
- **TPC-C.** We use the standard TPC-C workload, which consists of five transaction types: two read-only transactions and three read-write transactions. This workload has complex inter-transaction dependencies, making it harder to optimize than YCSB-extended. We vary the contention level by changing the number of warehouses (fewer warehouses mean higher contention). We vary the system load by changing the number of threads (more threads mean a higher system load). For this workload, the default setting is 16 threads and one warehouse. For a fair comparison, we follow Polyjuice to include all five types of TPC-C transactions. We learn the concurrency control actions for three types, i.e., *payment*, *new order*, and *delivery*, while excluding the remaining two types, as they are read-only transactions.

We have also conducted experiments that evaluate NeurCC's function optimization time under interactive transaction mode, the impact of features, and the impact of transaction length. Due to space constraints, we present these experiments in the extended version [35].

Baselines. We compare NeurCC against five state-of-the-art concurrency control algorithms: Silo [49], Polyjuice [50], IC3 [52], 2PL [7], and CormCC [45]. For the first four baselines, we use the implementations in [50]. For CormCC, we follow the approach in Polyjuice and report the best performance between 2PL and Silo. We refer to this baseline as *CormCC**.

6.2 Stored Procedure Mode

NeurCC and all the baselines support the stored procedure mode. We run all the experiments five times and report the average results. Each algorithm runs for three hours to match the time Polyjuice requires to reach optimal performance. We use IC3 as the initial function for function optimization.

Varying access pattern. We fix the contention level and system load while varying the access pattern using four different hotspot distributions.[3] Figure 7a shows the results for the YCSB-extended workload. The hotspot distributions, shown from left to right, shift from being skewed at the beginning of the transaction to being skewed at the end. We observe that NeurCC has the best throughput, up to 3.32×, 4.38×, and 4.27× higher than that of Polyjuice, 2PL, and Silo, respectively. The reason is that NeurCC learns a set of optimized pipeline-wait actions, timeouts, and conflict

[3]The values of 1 and 0 indicate hotspot and uniform access, respectively. The four access patterns, from the left to the right in Figure 7a, are $(0,0,0,1,0,0,0,0,0,0)$, $(1,0,0,1,1,0,0,0,0,0)$, $(0,0,0,1,0,1,0,1,1,0)$, and $(1,1,0,0,1,0,0,1,1,0)$.

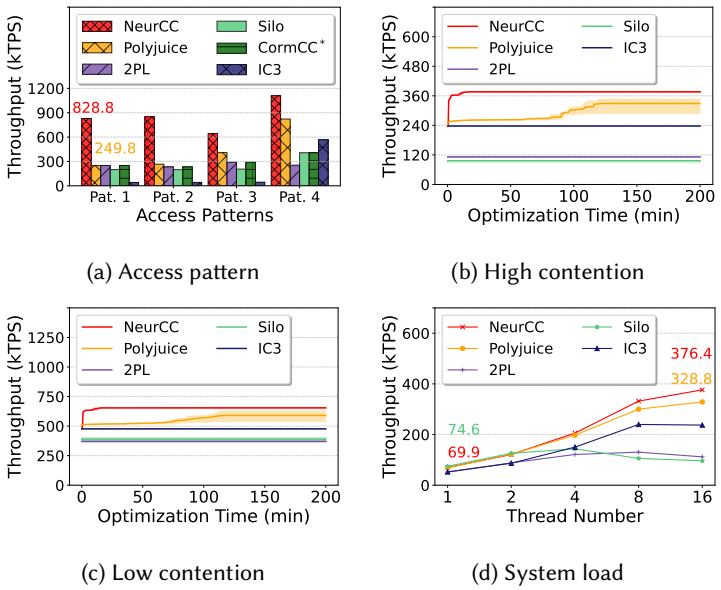


Fig. 7. Performance comparison between NeurCC and baselines under the YCSB-extended and TPC-C workloads in the stored procedure mode.

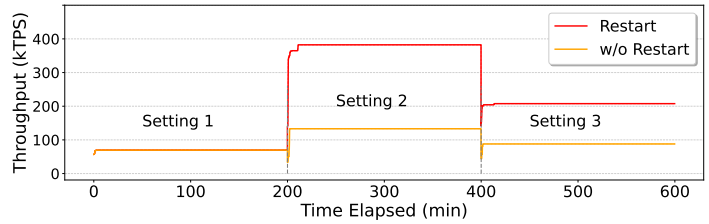
detection actions, enabling efficient transaction pipelining for hotspots while reducing the overhead for non-conflicting operations.

The performance gap between NeurCC and the second-best baseline decreases as the hotspot distribution shifts toward the end of the transaction. This is because the learned pipeline-wait actions, which contribute the most to the performance advantage of NeurCC, become less effective as the hotspots are closer to the end of the transactions. In particular, pipeline-wait actions allow dirty reads, but the advantage of dirty reads over clean reads depends on the trade-off between the gain from reducing block time and the overheads from cascading aborts and other issues [11, 15]. As the hotspots shifting towards the end, the gain from reducing block time becomes smaller.

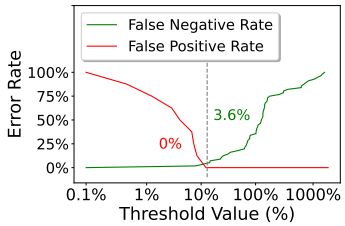
Varying contention level. We fix the thread number to 16 and measure the performance with one warehouse, representing high contention (Figure 7b), and with four warehouses, representing low contention (Figure 7c). The shaded areas represent the interquartile range, i.e., the 25th to 75th percentiles of throughput across multiple runs, illustrating the fluctuations of throughput in different CC algorithms. NeurCC achieves up to 1.14× throughput of the second-best baseline. NeurCC also exhibits a more stable learning process compared to Polyjuice. Polyjuice uses random genetic search, which leads to high fluctuations, whereas NeurCC employs a surrogate model to guide optimization, which achieves better stability throughout the optimization process.

Varying system load. We fix the number of warehouses to one and vary the system load by increasing the number of threadsfrom 1 to 16. Figure 7d shows the results. There are three main observations. First, NeurCC outperforms all the baselines under high load (more than four threads). This is because our algorithm finds a better combination of CC actions that allows for higher concurrency. Second, both NeurCC and Polyjuice perform slightly worse (less than 6.2% drop in throughput) than Silo under low load (one thread), even though they converge to the same CC actions. This gap is also reported in Polyjuice [50]. The reason is that the learning processes

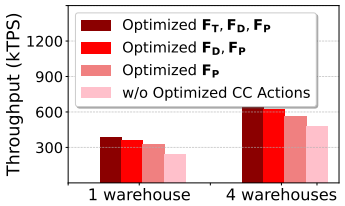
of NeurCC and Polyjuice incur additional overheads, which do not exist in the non-learned Silo system. Third, NeurCC achieves higher throughput than the baselines even when the workload does not saturate the system (e.g., with 4 threads). This is because NeurCC can still extract more concurrency from the workload than the baselines.



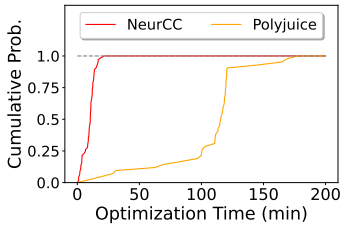
(a) Workload drift



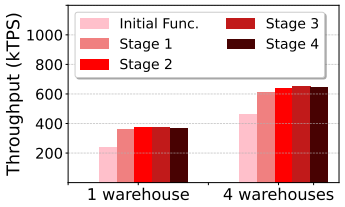
(b) Impact of threshold value



(c) Impact of learned actions



(d) Time to peak performance



(e) Impact of optimization stages

Fig. 8. Analysis of NeurCC in the stored procedure mode.

Workload drift. We simulate workload drifts under the TPC-C workload by changing the number of threads at different time intervals, i.e., 1 thread for Setting 1 (starting at minute 0), 16 threads for Setting 2 (starting at minute 200), and 4 threads for Setting 3 (starting at minute 400). Figure 8a compares the system throughput when optimizing from the initial function ("Restart"), versus when optimizing from the previously learned function ("w/o Restart"). It can be seen that NeurCC reacts quickly to workload drifts. Furthermore, the "w/o Restart" approach fails to achieve optimal performance within three hours in both Setting 2 and Setting 3. This is because our graph reduction search algorithm involves only edge removal and node merging mutations. We note that in order to fully explore the conflict graph search space, the optimization process must begin with the full conflict graph, containing all nodes and edges. Therefore, starting the optimization from a previously learned optimal function, where the conflict graph has already been reduced, restricts

the exploration of search space, and consequently leads to a suboptimal conflict graph for the new workload.

Impact of threshold value. We use eight different TPC-C workload settings[4] and evaluate how the threshold value affects the accuracy of drift detection when switching the workload among these settings. We use false positive rate and false negative rate as the accuracy metrics. For the false positive rate, we set the workload to one of the eight settings and report the percentage of the experiment runs where a workload drift is falsely detected despite no actual workload drift occurring. For the false negative rate, we evaluate all 7×8 possible workload drifts, starting from one of the eight settings and changing to another, and report the percentage of the experiment runs where these drifts are not detected. As shown in Figure 8b, the threshold value of 10% leads to robust detection (3.6% false negative rate) without over-triggering the optimization process (0% false positive rate).

Impact of learned actions. We evaluate the impact of the learned actions by reverting the learned CC actions in the final learned function to the actions in the initial function (IC3). Figure 8c shows that different actions have different impacts. Among them, the $\mathcal{F}_D$ and $\mathcal{F}_P$ sub-functions (see Section 5) — ones that are prioritized in our optimization pipeline — account for the majority of the performance improvement (85.8% and 80.6% in the one warehouse and 4 warehouse settings, respectively). The reason is that, although dirty reads allow for a higher degree of concurrency between transactions, they also introduce the risk of dependency cycles that can severely degrade performance. By optimizing $\mathcal{F}_D$ and $\mathcal{F}_P$, NeurCC mitigates this risk while ensuring high concurrency.

Impact of optimization stages. We evaluate how each optimization stage contributes to the agent function. Starting from the initial agent function, we measure the system throughput after each optimization stage. Figure 8e shows the results under TPC-C workloads with one and four warehouses. It can be seen that the highest improvements are achieved in the early stages. In particular, the largest performance gains are observed in Stage 1, where the optimization of $\mathcal{F}_D$ and $\mathcal{F}_P$ are prioritized. The later stages yield marginal improvements, indicating that the agent function has likely converged. The results demonstrate that the proposed optimization pipeline is effective: by prioritizing influential actions early, NeurCC achieves fast convergence, which is useful given the limited optimization time budgets.

Optimization time. Figure 8d compares the CDF of the optimization time for NeurCC and Polyjuice to reach peak performance. It can be seen that NeurCC has a significantly lower average optimization time (9.67 minutes) compared to Polyjuice (1.78 hours), or an 11× speedup. Further, the optimization time of NeurCC exhibits a lower variance than that of Polyjuice, because it uses graph reduction search guided by an online-trained surrogate model, whereas Polyjuice uses a random genetic search algorithm.

6.3 Interactive Transaction Mode

We compare NeurCC against three other baselines that support the interactive transaction mode, namely 2PL, Silo, and CormCC. In this mode, we use 2PL as the initial function for function optimization.

Varying contention level. Figure 9a and Figure 9b show the performance under one warehouse, i.e. high contention, and four warehouses, i.e. low contention. NeurCC outperforms the baselines under

[4]Denote (x, y) as x threads and y warehouses, the eight settings are $(1,1)$, $(2,1)$, $(4,1)$, $(8,1)$, $(16,1)$, $(16,2)$, $(16,4)$, and $(16,8)$.

both settings, achieving up to 1.96× higher throughput than that of the second-best baseline. This gap is due to NeurCC's learned conflict detection actions and wait priorities. For example, under the one warehouse setting, it learns to perform early validation after accessing hotspots (e.g., before a *new order* transaction reads the ITEM table), which leads to a timely abort when the transaction is likely to fail. It also assigns a low priority to early operations on the DISTRICT table, which reduces the likelihood of unresolvable conflicts. It ensures that a *new order*/*payment* transaction T_1, which has completed a *read-modify-write* operation on the DISTRICT table, is ordered before another *new order*/*payment* transaction T_2 that has not yet performed such operations. This way, T_2 does not miss updates from T_1, allowing them to commit together.

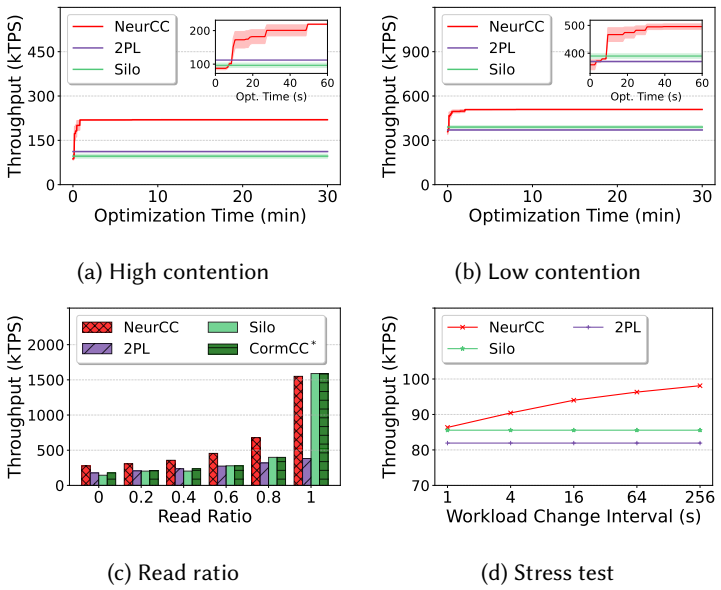
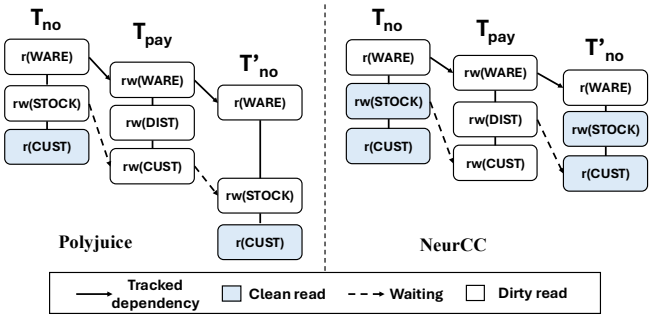


Fig. 9. Performance comparison between NeurCC and baselines in the interactive transaction mode.

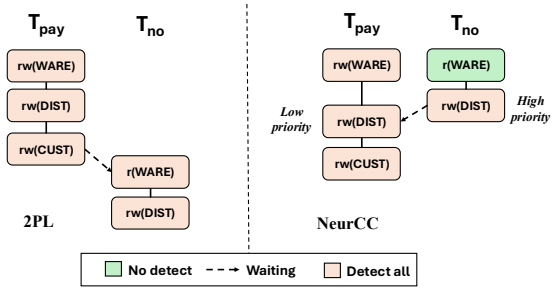
Varying read ratio. To understand the impacts of the read ratio, we perform one set of experiments using the YCSB-extended workload. Specifically, we fix the transaction length at 10 and vary the read ratio (i.e., the percentage of read operations) from 0% to 100%. The results are shown in Figure 9c. The throughput of all algorithms increases as the read ratio increases. This trend is expected because higher read ratios result in fewer conflicts among transactions. Further, NeurCC outperforms the second-best baselines by up to 1.7× when the read ratio is below 100%. For read-only workloads, NeurCC is slightly worse (by 2.5%) than Silo. The reason is that learning-based algorithms can learn complex actions that deliver high performance when there are many conflicts, but they are less effective when there are few conflicts, due to the inherent overhead in the learning process.

Rapid workload changes. Although real-world workloads typically evolve gradually [10, 28], rapid workload changes can still arise from planned events such as yearly promotional and sales events. To evaluate NeurCC's robustness under such rapid changes, we vary the number of client threads from 1 to 16 at different switching frequencies. Figure 9d shows the average throughput over 1 hour. It can be seen that NeurCC performs consistently better than the baselines, even under

highly unstable workloads. At the highest switching frequency, i.e. 1 second, NeurCC achieves 85.90 ktps, which is comparable to the best static baseline (Silo achieves 85.57 ktps). The results demonstrate that NeurCC is robust under rapid workload changes.



(a) Stored procedure mode



(b) Interactive transaction mode

Fig. 10. Examples of the learned actions.

6.4 Examples of Learned Actions

Figure 10a and Figure 10b show examples that compare the CC actions learned by NeurCC with those learned by the second-best baselines. While our implementation uses row-level locking, we show only table names in the examples for clarity.

In the stored procedure mode, Polyjuice is the second-best baseline. As shown in Figure 10a, both NeurCC and Polyjuice read dirty data during WAREHOUSE table accesses and track the transaction dependency $T_{no} \rightarrow T_{pay} \rightarrow T'_{no}$. Polyjuice learns to perform the STOCK table access of T'_{no} only after the STOCK table access of T_{no}, preventing a potential dependency cycle $T_{no} \rightarrow T_{pay} \rightarrow T'_{no} \rightarrow T_{no}$. In contrast, NeurCC learns a conflict graph where STOCK table accesses of T_{no} and T'_{no} are unlikely to conflict with each other. Therefore, it enables a more efficient transaction interleaving where the STOCK table access from T_{no}, the DISTRICT table access from T_{pay}, and the STOCK table access from T'_{no} can be executed in parallel.

In the interactive transaction mode (Figure 10b), 2PL is the second-best baseline, and it resolves the conflicts between T_{no} and T_{pay} by performing the WAREHOUSE table access of T_{no} only after T_{pay} commits. This results in a long execution time. On the other hand, NeurCC learns to order T_{no} before T_{pay}, allowing T_{no} to optimistically access the WAREHOUSE table without waiting for T_{pay} to commit. It also learns to assign a high priority to the DISTRICT table access of T_{no}, ordering this access before that of T_{pay}. This prevents a dependency cycle $T_{no} \rightarrow T_{pay} \rightarrow T_{no}$. The

safe interleaving of the two transactions allows them to commit faster in NeurCC. This example demonstrates that by enabling better interleaving of transactions, NeurCC can outperform 2PL despite incurring extra validation costs.

7 Related Work

Adaptive CC algorithms aim to achieve high performance across different workloads. They follow a classify-then-assign workflow: analyzing workload information, classifying transactions based on manually selected features, and assigning the most suitable CC algorithm for each type of transaction. The assignment step can be either heuristic [16, 20, 25, 41, 42, 44, 45] or based on learning [46, 50, 62].

Among the algorithms adopting heuristic assignment, C3 [42] uses locality-sensitive hashing (LSH) to cluster transactions with similar SQL texts, which tend to conflict on shared data objects. It applies 2PL for intra-group conflicts and OCC for inter-group conflicts. AEP [44] predicts whether a distributed transaction will encounter conflicts based on local data, then uses EWL when there are no conflicts, and uses PSL when there are. MCC [55] and Tebaldi [41] classify transactions based on prior workload knowledge, constructing an algorithm hierarchy that switches between RP, SSI, and 2PL, depending on conflict type. ASOCC [25] categorizes data into hot, warm, and cold based on the contention level, then uses OCC for cold data, 2PL for hot data, and early validation with retry for warm data. NeurCC covers more CC designs than these algorithms, and it learns a fine-grained and flexible combination of them for better performance.

NeurCC shares similarities with other algorithms that adopt learning-based assignment. Polyjuice [50] classifies transaction operations by their access ID (a hard-coded unique transaction access identifier) and uses an evolutionary algorithm to optimize pipeline-wait actions and transaction back-offs. ACC [46] clusters transactions similar to PartCC [21] and assigns the best algorithm among OCC, 2PL, and PartCC to each cluster. For fast reconfiguration, it also trains a model that predicts the optimal algorithm for each cluster, enabling automatic online reconfiguration with offline-trained models. NeurCC supports more CC design choices than these algorithms, and achieves better performance across more diverse workloads. Furthermore, it does not require prior knowledge of the workloads for access ID assignment or transaction clustering.

8 Conclusions

In this paper, we presented NeurCC, a novel learned concurrency control algorithm that achieves high performance across diverse workloads. NeurCC models concurrency control as a learnable function. It learns design choices from a large number of existing concurrency control algorithms. The algorithm employs an optimization algorithm that quickly converges to a combination of concurrency control actions that achieves high transaction throughput. Specifically, it uses Bayesian optimization and graph reduction search to reduce the overhead of performance evaluations, thereby speeding up the optimization process. Our evaluation shows that NeurCC delivers high throughput, outperforming the baselines. In particular, its throughput is up to 3.32×, 4.38×, and 4.27× higher than that of Polyjuice, 2PL, and Silo, respectively. Further, NeurCC's optimization time is up to 11× faster than that of other learned baselines, making it more robust and adaptive to workload drifts.

Acknowledgments

We would like to thank the anonymous reviewers and meta-reviewer for their constructive comments. Yuncheng Wu's work is supported by National Natural Science Foundation of China U24B20144. Gang Chen's work is supported by the National Regional Innovation and Development Joint Fund (No. U24A20254).

References

[1] 2026. Code Implementation for NeurCC. https://github.com/neurdb/neurcc.

[2] 2026. NeurDB. https://github.com/neurdb/neurdb.

[3] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. 1995. Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, Michael J. Carey and Donovan A. Schneider (Eds.). ACM Press, 23–34. https://doi.org/10.1145/223784.223787

[4] Claude Barthels, Ingo Müller, Konstantin Taranov, Gustavo Alonso, and Torsten Hoefler. 2019. Strong consistency is not hard to get: Two-Phase Locking and Two-Phase Commit on Thousands of Cores. *Proc. VLDB Endow.* 12, 13 (2019), 2325–2338. https://doi.org/10.14778/3358701.3358702

[5] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley. http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx

[6] Michael J. Carey, Sanjay Krishnamurthi, and Miron Livny. 1990. Load Control for Locking: The 'Half-and-Half' Approach. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, April 2-4, 1990, Nashville, Tennessee, USA*, Daniel J. Rosenkrantz and Yehoshua Sagiv (Eds.). ACM Press, 72–84. https://doi.org/10.1145/298514.298543

[7] Alain Chesnais, Erol Gelenbe, and Isi Mitrani. 1983. On the modeling of parallel access to shared data. *Commun. ACM* 26, 3 (1983), 196–202.

[8] Andrew R. Conn, Katya Scheinberg, and Luís Nunes Vicente. 2009. *Introduction to Derivative-Free Optimization.* MPS-SIAM series on optimization, Vol. 8. SIAM.

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

[10] Sudipto Das, Feng Li, Vivek R. Narasayya, and Arnd Christian König. 2016. Automated Demand-driven Resource Scaling in Relational Database-as-a-Service. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 1923–1934. https://doi.org/10.1145/2882903.2903733

[11] Jose M. Faleiro, Daniel Abadi, and Joseph M. Hellerstein. 2017. High Performance Transactions via Early Write Visibility. *Proc. VLDB Endow.* 10, 5 (2017), 613–624. https://doi.org/10.14778/3055540.3055553

[12] Alan D. Fekete. 2018. Serializable Snapshot Isolation. In *Encyclopedia of Database Systems, Second Edition*, Ling Liu and M. Tamer Özsu (Eds.). Springer. https://doi.org/10.1007/978-1-4614-8265-9_80774

[13] Peter A. Franaszek, John T. Robinson, and Alexander Thomasian. 1991. Wait Depth Limited Concurrency Control. In *Proceedings of the Seventh International Conference on Data Engineering, April 8-12, 1991, Kobe, Japan.* IEEE Computer Society, 92–101. https://doi.org/10.1109/ICDE.1991.131456

[14] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D. Sculley. 2017. Google Vizier: A Service for Black-Box Optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, August 13 - 17, 2017.* ACM, 1487–1495. https://doi.org/10.1145/3097983.3098043

[15] Zhihan Guo, Kan Wu, Cong Yan, and Xiangyao Yu. 2021. Releasing Locks As Early As You Can: Reducing Contention of Hotspots by Violating Two-Phase Locking. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 658–670. https://doi.org/10.1145/3448016.3457294

[16] Yinhao Hong, Hongyao Zhao, Wei Lu, Xiaoyong Du, Yuxing Chen, Anqun Pan, and Lixiong Zheng. 2025. A Hybrid Approach to Integrating Deterministic and Non-deterministic Concurrency Control in Database Systems. *Proc. VLDB Endow.* 18, 5 (2025), 1376–1389. https://www.vldb.org/pvldb/vol18/p1376-lu.pdf

[17] Meichun Hsu and Bin Zhang. 1992. Performance Evaluation of Cautious Waiting. *ACM Trans. Database Syst.* 17, 3 (1992), 477–512. https://doi.org/10.1145/132271.132275

[18] Jiamin Huang, Barzan Mozafari, Grant Schoenebeck, and Thomas F. Wenisch. 2017. A Top-Down Approach to Achieving Performance Predictability in Database Systems. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 745–758. https://doi.org/10.1145/3035918.3064016

[19] Jiamin Huang, Barzan Mozafari, and Thomas F. Wenisch. 2017. Statistical Analysis of Latency Through Semantic Profiling. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic (Eds.). ACM, 64–79. https://doi.org/10.1145/3064176.3064179

[20] Kaisong Huang, Jiatang Zhou, Zhuoyue Zhao, Dong Xie, and Tianzheng Wang. 2025. Low-Latency Transaction Scheduling via Userspace Interrupts: Why Wait or Yield When You Can Preempt? *Proc. ACM Manag. Data* 3, 3 (2025), 182:1–182:25. https://doi.org/10.1145/3725319

[21] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.* 1, 2 (2008), 1496–1499. https://doi.org/10.14778/1454159.1454211

[22] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2014. Exploiting hardware transactional memory in main-memory databases. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, Isabel F. Cruz, Elena Ferrari, Yufei Tao, Elisa Bertino, and Goce Trajcevski (Eds.). IEEE Computer Society, 580–591. https://doi.org/10.1109/ICDE.2014.6816683

[23] 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, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 21–35. https://doi.org/10.1145/3035918.3064015

[24] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. 2016. Towards a Non-2PC Transaction Management in Distributed Database Systems. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 1659–1674. https://doi.org/10.1145/2882903.2882923

[25] Qian Lin, Gang Chen, and Meihui Zhang. 2018. On the Design of Adaptive and Speculative Concurrency Control in Distributed Databases. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 1376–1379. https://doi.org/10.1109/ICDE.2018.00154

[26] Dong C. Liu and Jorge Nocedal. 1989. On the limited memory BFGS method for large scale optimization. *Math. Program.* 45, 1-3 (1989), 503–528. https://doi.org/10.1007/BF01589116

[27] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: A Fast and Practical Deterministic OLTP Database. *Proc. VLDB Endow.* 13, 11 (2020), 2047–2060. http://www.vldb.org/pvldb/vol13/p2047-lu.pdf

[28] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. 2018. Query-based Workload Forecasting for Self-Driving Database Management Systems. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 631–645. https://doi.org/10.1145/3183713.3196908

[29] Axel Mönkeberg and Gerhard Weikum. 1991. Conflict-driven Load Control for the Avoidance of Data-Contention Thrashing. In *Proceedings of the Seventh International Conference on Data Engineering, April 8-12, 1991, Kobe, Japan*. IEEE Computer Society, 632–639. https://doi.org/10.1109/ICDE.1991.131512

[30] Axel Mönkeberg and Gerhard Weikum. 1992. Performance Evaluation of an Adaptive and Robust Load Control Method for the Avoidance of Data-Contention Thrashing. In *18th International Conference on Very Large Data Bases, August 23-27, 1992, Vancouver, Canada, Proceedings*, Li-Yan Yuan (Ed.). Morgan Kaufmann, 432–443. http://www.vldb.org/conf/1992/P432.PDF

[31] Shuai Mu, Sebastian Angel, and Dennis E. Shasha. 2019. Deferred Runtime Pipelining for contentious multicore software transactions. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, George Candea, Robbert van Renesse, and Christof Fetzer (Eds.). ACM, 40:1–40:16. https://doi.org/10.1145/3302424.3303966

[32] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. 2014. Extracting More Concurrency from Distributed Transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, Jason Flinn and Hank Levy (Eds.). USENIX Association, 479–494. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/mu

[33] Cuong D. T. Nguyen, Kevin Chen, Christopher DeCarolis, and Daniel J. Abadi. 2025. Are Database System Researchers Making Correct Assumptions about Transaction Workloads? *Proc. ACM Manag. Data* 3, 3 (2025), 131:1–131:26. https://doi.org/10.1145/3725268

[34] Beng Chin Ooi, Shaofeng Cai, Gang Chen, Kian Lee Tan, Yuncheng Wu, Xiaokui Xiao, Naili Xing, Cong Yue, Lingze Zeng, Meihui Zhang, et al. 2024. NeurDB: An AI-powered Autonomous Data System. *arXiv preprint arXiv:2405.03924* (2024).

[35] Hexiang Pan, Shaofeng Cai, Tien Tuan Anh Dinh, Yuncheng Wu, Yeow Meng Chee, Gang Chen, and Beng Chin Ooi. 2026. NeurCC Extended Version. https://arxiv.org/abs/2503.10036.

[36] Andrew Pavlo, Carlo Curino, and Stanley B. Zdonik. 2012. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman (Eds.). ACM, 61–72. https://doi.org/10.1145/2213836.2213844

[37] Rommel G Regis and Christine A Shoemaker. 2013. Combining radial basis function surrogates and dynamic coordinate search in high-dimensional expensive black-box optimization. *Engineering Optimization* 45, 5 (2013), 529–555.

[38] Daniel J. Rosenkrantz, Richard Edwin Stearns, and Philip M. Lewis II. 1978. System Level Concurrency Control for Distributed Database Systems. *ACM Trans. Database Syst.* 3, 2 (1978), 178–198. https://doi.org/10.1145/320251.320260

[39] Zechao Shang, Feifei Li, Jeffrey Xu Yu, Zhiwei Zhang, and Hong Cheng. 2016. Graph Analytics Through Fine-Grained Parallelism. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 463–478. https://doi.org/10.1145/2882903.2915238

[40] 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, aiDM@SIGMOD 2019, Amsterdam, The Netherlands, July 5, 2019*, Rajesh Bordawekar and Oded Shmueli (Eds.). ACM, 1:1–1:8. https://doi.org/10.1145/3329859.3329871

[41] Chunzhi Su, Natacha Crooks, Cong Ding, Lorenzo Alvisi, and Chao Xie. 2017. Bringing Modular Concurrency Control to the Next Level. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 283–297. https://doi.org/10.1145/3035918.3064031

[42] Xuebin Su, Hongzhi Wang, and Yan Zhang. 2021. Concurrency Control Based on Transaction Clustering. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 2195–2200. https://doi.org/10.1109/ICDE51399.2021.00223

[43] Richard S. Sutton, David A. McAllester, Satinder Singh, and Yishay Mansour. 1999. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In *NIPS*, Sara A. Solla, Todd K. Leen, and Klaus-Robert Müller (Eds.). The MIT Press, 1057–1063.

[44] Ann T. Tai and John F. Meyer. 1996. Performability Management in Distributed Database Systems: An Adaptive Concurrency Control Protocol. In *MASCOTS '96, Proceedings of the Fourth International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems, February 1-3, 1996, San Jose, California, USA*. IEEE Computer Society, 212–216. https://doi.org/10.1109/MASCOT.1996.501020

[45] Dixin Tang and Aaron J. Elmore. 2018. Toward Coordination-free and Reconfigurable Mixed Concurrency Control. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, Haryadi S. Gunawi and Benjamin C. Reed (Eds.). USENIX Association, 809–822. https://www.usenix.org/conference/atc18/presentation/tang

[46] Dixin Tang, Hao Jiang, and Aaron J. Elmore. 2017. Adaptive Concurrency Control: Despite the Looking Glass, One Concurrency Control Does Not Fit All. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2017/papers/p63-tang-cidr17.pdf

[47] 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 ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman (Eds.). ACM, 1–12. https://doi.org/10.1145/2213836.2213838

[48] Boyu Tian, Jiamin Huang, Barzan Mozafari, and Grant Schoenebeck. 2018. Contention-Aware Lock Scheduling for Transactional Databases. *Proc. VLDB Endow.* 11, 5 (2018), 648–662. https://doi.org/10.1145/3187009.3177740

[49] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 18–32. https://doi.org/10.1145/2517349.2522713

[50] Jia-Chen 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 2021, July 14-16, 2021*, Angela Demke Brown and Jay R. Lorch (Eds.). USENIX Association, 198–216. https://www.usenix.org/conference/osdi21/presentation/wang-jiachen

[51] Tianzheng Wang and Hideaki Kimura. 2016. Mostly-Optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores. *Proc. VLDB Endow.* 10, 2 (2016), 49–60. https://doi.org/10.14778/3015274.3015276

[52] Zhaoguo Wang, Shuai Mu, Yang Cui, Han Yi, Haibo Chen, and Jinyang Li. 2016. Scaling Multicore Databases via Constrained Parallel Execution. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, San Francisco, CA, USA, 1643–1658. https://doi.org/10.1145/2882903.2882934

[53] Peizhi Wu and Zachary G. Ives. 2024. Modeling Shifting Workloads for Learned Database Systems. *Proc. ACM Manag. Data* 2, 1 (2024), 38:1–38:27. https://doi.org/10.1145/3639293

[54] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment* 10, 7 (2017), 781–792.

[55] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. 2015. High-performance ACID via modular concurrency control. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, Ethan L. Miller and Steven Hand (Eds.). ACM, Monterey, CA, USA, 279–294. https://doi.org/10.1145/2815400.2815430

[56] Chang Yao, Meihui Zhang, Qian Lin, Beng Chin Ooi, and Jiatao Xu. 2018. Scaling distributed transaction processing and recovery based on dependency logging. *VLDB J.* 27, 3 (2018), 347–368. https://doi.org/10.1007/S00778-018-0500-2

[57] Chenhao Ye, Wuh-Chwen Hwang, Keren Chen, and Xiangyao Yu. 2023. Polaris: Enabling Transaction Priority in Optimistic Concurrency Control. *Proc. ACM Manag. Data* 1, 1 (2023), 44:1–44:24. https://doi.org/10.1145/3588724

[58] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.* 8, 3 (2014), 209–220. https://doi.org/10.14778/2735508.2735511

[59] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2017. Building Consistent Transactions with Inconsistent Replication. *ACM Trans. Comput. Syst.* 35, 4 (2017), 12:1–12:37. https://doi.org/10.1145/3269981

[60] Qian Zhang, Yiwen Xiang, Jianhao Wei, Yang Yang, Yifan Li, Xueqing Gong, and Wanggen Liu. 2025. Rebirth-Retire: A Concurrency Control Protocol Adaptable to Different Levels of Contention. *Proc. VLDB Endow.* 18, 9 (2025), 3162–3174. https://www.vldb.org/pvldb/vol18/p3162-zhang.pdf

[61] Zhanhao Zhao, Shaofeng Cai, Haotian Gao, Hexiang Pan, Siqi Xiang, Naili Xing, Gang Chen, Beng Chin Ooi, Yanyan Shen, Yuncheng Wu, and Meihui Zhang. 2024. NeurDB: On the Design and Implementation of an AI-powered Autonomous Database. *CIDR* (2024).

[62] Qiyu Zhuang, Wei Lu, Shuang Liu, Yuxing Chen, Xinyue Shi, Zhanhao Zhao, Yipeng Sun, Anqun Pan, and Xiaoyong Du. 2025. TxnSails: Achieving Serializable Transaction Scheduling with Self-Adaptive Isolation Level Selection. *Proc. VLDB Endow.* 18, 11 (2025), 4227–4240. https://www.vldb.org/pvldb/vol18/p4227-lu.pdf

Received October 2025; revised January 2026; accepted February 2026