Polyjuice: High-Performance Transactions via Learned Concurrency Control

Jiachen Wang[†◇*], Ding Ding[‡], Huan Wang[†◇*], Conrad Christensen[‡], Zhaoguo Wang[†◇*], Haibo Chen[†◇*], and Jinyang Li[‡]

[†]Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
[◇]Shanghai AI Laboratory
[*]Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China
[‡]Department of Computer Science, New York University

Abstract

Concurrency control algorithms are key determinants of the performance of in-memory databases. Existing algorithms are designed to work well for certain workloads. For example, optimistic concurrency control (OCC) is better than two-phase-locking (2PL) under low contention, while the converse is true under high contention.

To adapt to different workloads, prior works mix or switch between a few known algorithms using manual insights or simple heuristics. We propose a learning-based framework that instead explicitly optimizes concurrency control via offline training to maximize performance. Instead of choosing among a small number of known algorithms, our approach searches in a "policy space" of fine-grained actions, resulting in novel algorithms that can outperform existing algorithms by specializing to a given workload.

We build Polyjuice based on our learning framework and evaluate it against several existing algorithms. Under different configurations of TPC-C and TPC-E, Polyjuice can achieve throughput numbers higher than the best of existing algorithms by 15% to 56%.

1 Introduction

Concurrency control (CC) algorithms lie at the foundation of modern database systems [18]. A CC algorithm synchronizes a transaction's access to storage objects to maximize concurrent execution while guaranteeing correctness. As today's database systems are no longer disk-bound, the CC algorithm in use becomes crucial to a database's performance.

Traditional CC algorithms, such as two-phase-locking (2PL) [17] and optimistic concurrency control (OCC) [28], take fixed algorithmic steps regardless of the workload. Thus, it comes as no surprise that the relative performance of different algorithms varies depending on the transaction workload. Figure 1 shows the throughput of 2PL, OCC and IC3 [60] on a multi-core database under the TPC-C workload with a varying number of warehouses. OCC has the highest throughput under low contention (more warehouses) while the other two outperform OCC under high contention (fewer warehouses). Similar results have also been reported by others [67].

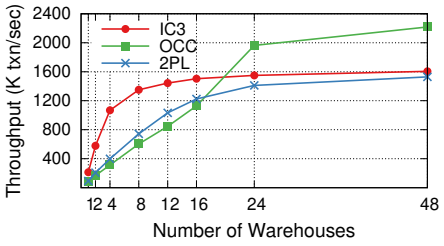


Figure 1: IC3, OCC, 2PL performance on TPC-C, 48 threads.

To adapt to different workloads, prior works propose a federated approach by simultaneously supporting a small number of existing CC algorithms, including 2PL and OCC. These systems require users to partition the workload either by data [55] or by transaction type [49, 53, 65]. The decision of which algorithm to use for each partition is either based on manual insights [49, 53, 65] or simple runtime metrics [55]. While this federated approach can improve performance, it has limitations. First, by limiting itself to using a small number of known algorithms, it lacks the flexibility to customize concurrency control to fully exploit the workload. Second, by relying on manual insights or simple heuristics, it lacks a systematic solution to optimize concurrency control for performance.

This paper aims to develop a learning-based framework to optimize concurrency control for a given workload. We assume the workload is known a prior such as past workloads, e.g. in the form of stored procedures. To enable learning, we design a "policy space" of fine-grained actions (a.k.a. algorithmic steps): each policy can be viewed as a CC algorithm that uses specific actions to synchronize different data accesses made by different transactions. All policies perform an explicit validation before transactions commit to ensure serializability. We use offline training to learn the highest

Jiachen Wang and Ding Ding are contributed equally to this paper.

performing policy for a given workload. This framework is expressive: it can learn new CC algorithms as well as existing ones. It also allows explicit optimization for performance via systematic searches of the policy space.

We have realized the design for learned concurrency control in a system called Polyjuice for multi-core in-memory databases. The core technical challenge of Polyjuice is to design the policy space. Inspired by reinforcement learning, we view each policy as a function that maps each state (i.e., the execution context of a data access) to actions that control the interleavings of accesses made by concurrent transactions. In Polyjuice, the state specifies what type of transaction is being used and which of its accesses are under execution. The actions support multiple ways of interleaving control, including deciding which data version to read, whether to expose an uncommitted write, how long to wait before access, and whether to perform early validation before commit.

Polyjuice represents each policy function using a table: the rows correspond to different states and the columns correspond to different kinds of actions. Polyjuice uses an evolutionary algorithm based training to search the policy space for the policy that has the highest commit throughput for a given workload.

We train and evaluate Polyjuice's performance on micro-benchmarks, TPC-C and TPC-E, and compare with existing algorithms, including Silo [57](OCC), 2PL, Tebaldi [53], CormCC [55] and IC3 [60]. Our experiments show that, for TPC-C and TPC-E with moderate to high contention, Polyjuice can find a CC policy whose throughput is better than the best of existing algorithms by 15% to 56%. Detailed analysis shows that Polyjuice can learn an interesting policy that is different than any of the existing algorithms to exploit the workload in subtle ways(§7.3). For workloads with almost no contention, Polyjuice learns the same policy as OCC and incurs 8% slowdown due to its implementation overhead.

As Polyjuice requires offline training, it is not suitable for dynamic workloads that can change rapidly and unpredictably. However, our analysis of an e-commerce website trace shows that real-world workloads are fairly predictable in terms of its peak hour workload characteristics including the likelihood of conflict. This suggests that it is practical to use Polyjuice to optimize a database's peak performance by training on traces of recently observed peak workloads.

In summary, our paper makes the follow contributions:

- We present the first framework to learn concurrency control using a policy space of fine-grained actions.
- We design Polyjuice's policy space according to the framework so that it can encode a variety of existing CC algorithms while allowing the exploration of new ones.
- We show that Polyjuice's policy, represented as a table, can be optimized simply using an evolutionary algorithm.
- Even for the heavily-studied TPC-C benchmarks, Polyjuice can find interesting and novel policies not seen in existing algorithms to improve transaction throughput under moderate to high contention.

2 Background and Motivation

Existing works have realized the inadequacy of using one fixed concurrency control algorithm for different workloads. For the solution, they propose a federated approach of mixing a few (typically 2 or 3) known CC algorithms [53, 55, 59, 65]. In this section, we discuss the limitations of this federated approach and motivate the need for a more expressive learning-based approach.

The federated approach of adapting CC to a workload is characterized by its *coarse-grained* way of mixing different algorithms. Specifically, this approach coarsely partitions the workload. The same CC algorithm is used within a workload partition, while a different algorithm may be used for a different partition. Two ways of partitioning can be found in existing work. CormCC [55] partitions by data: all accesses to data in the same partition use the same CC algorithm. Tebaldi [53] and Callas [65] group (a.k.a. partition) transactions by types: all transactions belonging to the same group (a.k.a. partition) use the same CC for all their data accesses.

The coarse-grained way of mixing CC algorithms is limited in its ability to fully exploit workload characteristics for performance. For example, with CormCC, if transactions T and T' both only access data within the same partition, they would synchronize all of their accesses using the same CC algorithm. Similarly for Tebaldi and Callas, if transactions T and T' are of the same type, they would always use the same CC algorithm. This is not optimal: if different data accesses of T and T' have different contention characteristics, they may be better served by different methods for controlling concurrency.

A second limitation of existing federated CC work is their reliance on manual insights to partition the workload or to determine which CC algorithm to use for each partition. Callas and Tebaldi manually assign transactions to groups and choose a specific CC algorithm for each group. CormCC partitions the TPC-C workload by warehouse based on manual insights and uses simple runtime statistics (e.g. read/write ratio) to decide which CC algorithm to use for each partition.

Our approach. We aim to optimize CC for a given workload in a *fine-grained* way using a learning-based approach. Instead of partitioning the workload and using a single CC algorithm for all data accesses within the partition, we propose to allow each data access to use one of many different fine-grained "actions" to mediate potentially conflicting accesses. When deciding what action(s) to take to maximize performance, we are not concerned with correctness; instead, we rely on a separate validation mechanism to abort non-serializable transactions. As fine-grained actions lead to exponentially many choices for a given workload, it is impossible to rely on manual insights to choose the best action(s). A more prac-

tical solution is using a learning-based approach to select actions that explicitly optimize the performance for the given workload.

The main challenge of our approach is to design the learning framework with fine-grained actions for concurrency control. Ideally, the framework should be expressive enough to encode **most** existing CC algorithms and to allow the synthesis of new ones. In the next section, we discuss how to design such a learning framework.

3 Learning Concurrency Control

In this section, we examine how to frame concurrency control as a fine-grained learning task.

System settings. Our target setting is an in-memory database running on a single multi-core machine. We assume the kinds of transaction to be run on the database are known a priori, e.g. in the form of stored procedures. A number of existing work also exploit a known-workload in designing CC algorithms [41, 60, 65]. Our work focuses on learning concurrency control for read-write transactions, and reuses existing mechanisms to support logging and snapshot-based read-only transactions [57]. Although our learning framework is general enough to represent multi-version concurrency control (MVCC), our later system design does not support it because existing snapshot-based read-only transactions can already capture much of MVCC's performance benefits.

3.1 The learning framework

Our framework for learning concurrency control is inspired by reinforcement learning (RL). As one of the major branches of machine learning, RL involves learning how to interact with an environment to maximize a numerical reward. The key ingredients in RL are: a *policy* that maps perceived states of the environment to actions to be taken when those states are reached, a *reward* signal that defines the optimization goal, and the *environment* under which the learning system operates. In our context, the policy corresponds to the CC algorithm; the reward corresponds to some performance metric to be maximized; the environment captures the transaction workload and system setup under which the CC operates.

It is straightforward to decide on the optimization objective (a.k.a. reward). In this work, we use transaction throughput. Compared to latency or abort rate, transaction throughput is widely used as the key end-to-end performance metric for in-memory databases.

It is non-trivial to design a "policy space" to represent various CC algorithms. At a high level, a CC algorithm executes a transaction by controlling how its data access can interleave with potentially conflicting accesses from other concurrent transactions. As mentioned previously, we do not attempt to learn how to guarantee correctness. Instead, a learned CC algorithm always invokes a manually-designed validation procedure as part of transaction commit to ensure serializability. What we do learn is a policy that determines what actions to take in order to maximize performance for a given workload. A good CC policy balances how long transactions execute vs. how likely transactions are aborted, resulting in a high reward, as measured by how many transactions successfully commit per second. Aside from the CC policy, how long a database backs off before retrying an aborted transaction can also affect the performance. We separate the backoff policy from the CC policy, and this section focuses on the latter.

The policy space of concurrency control. Taking a page from reinforcement learning, we represent the policy as a mapping from some state of execution to a specific action to take upon encountering that state. Taking different actions in different states allows us to specialize a CC algorithm to optimize for a given workload. Thus, the state space should include information that is necessary to distinguish circumstances that require different actions, e.g. the type of transaction that is making the access, the type of access etc. In a later section (§4.2), we provide a concrete design of the state space. In the rest of this section, we focus on designing the action space.

Ideally, the action space should encompass a set of fine-grained actions that can be mixed and matched to represent many different CC algorithms. These actions can be classified into two categories: 1) actions that control how the data access of concurrent transactions can interleave during transaction execution, and 2) actions that control when and how to perform validation in order to detect whether an executed transaction has violated serializability. Next, we discuss the spectrum of actions available to use in each of the two categories.

Available actions for interleaving control. These actions mediate potentially conflicting data accesses, thereby affecting the set of dependencies that arise among concurrent transactions. There are 3 types of dependencies: write-write $\xrightarrow{ww}$ (a.k.a. write dependency), write-read $\xrightarrow{wr}$ (a.k.a. read dependency), or read-write $\xrightarrow{rw}$ (a.k.a. anti-dependency) [1]. What are the knobs of control that can affect these dependencies?

To discover these knobs in their full generality, let us assume a hypothetical yet still practical database design that keeps track of each read and write access of transactions in a per-object access list, similar to the approach taken in [42, 60]. As a transaction T performs data accesses, it may insert its reads/writes to the corresponding per-object access lists while also updating T_{dep}, the set of transactions that T becomes dependent on. Using this flexible way of tracking dependencies enables a wide range of design choices for interleaving control, as we will see next.

When executing transaction T, a CC algorithm has the following action choices:

	Interleaving control				Validation	
	Read wait	Read version	Write wait	Write visibility	Early validation	Validation method
2PL*	Until T_{dep} commits	latest committed	Until T_{dep} commits	Yes	Yes	n/a
OCC [28]	No	latest committed	No	No	No	physical cts
TicToc [68]					No	logical cts
Sundial [69]	No	latest committed	Until T_{wdep} commits	No	No	logical cts
Callas RP [65] IC3 [60], DRP [41]	Until T_{dep} finish certain access	latest un-committed	Until T_{dep} finish certain access	piece-end	piece-end	n/a or physical cts
MVTSO [2] (MVCC)	Until $T' \in T_{wdep}$ commits if $ts(T') < ts(T)$	largest committed $< ts(T)$	No	Yes	Yes	physical ts

Table 1: The choices made in existing CC algorithms according to the action space described in §3. T refers to the current transaction. T_{dep} refers to the set of transactions that T is dependent on (due to its conflicting access so far). T_{wdep} is the subset of T_{dep} whose writes have conflicted with T. $ts(T)$ refers to the timestamp assigned to T by MVTSO [2].

- *Read control*. There are two dimensions to these actions:
  1. *Wait*. This can let some dependent transaction $T' \in T_{dep}$ perform its conflicting write earlier than T's read, resulting in $T' \xrightarrow{wr} T$. Otherwise, a dependency cycle may arise with $T \xrightarrow{rw} T'$, resulting in aborts.
  2. *Which version of data to read, including either committed or uncommitted version*. This amounts to choosing which location in the access list to insert the read, thereby affecting dependencies. Specifically, since a read returns the latest write w before itself in the access list, there is a write-read dependency, $T' \xrightarrow{wr} T$, for every T' whose write appears before this read in the list. Additionally, a read also results in a set of read-write dependencies $T \xrightarrow{rw} T'$, for every T' whose write appears after this read in the list.
- *Write control.* There are also two dimensions to these actions:
  1. *Wait*. The rationale for this action is similar to that for reads.
  2. *Whether or not to make this write visible to the future reads of other transactions.* The write is buffered if it is not exposed. Otherwise, this write as well as all of T's previously buffered writes are made visible by appending them to the corresponding per-object access lists. The cumulative way of exposing writes makes sense because otherwise, any transaction that has read this but not a previous write of T would violate serializability and get aborted. Unlike a read, there's no flexibility to insert a write in any location but the end of the list; this is because we cannot allow a write to affect past reads. Exposing a write does not imply that uncommitted data will be read because transactions can choose to read committed versions only. In terms of the resulting dependencies, exposing a write causes $T' \xrightarrow{ww} T$ or $T' \xrightarrow{rw} T$ for any T' whose write or read appears before this write in the list.

Available actions for validation. Actions in this category can control two aspects of validation:

- *When to validate.* A transaction may validate its accesses at any time during execution, instead of only at commit time. Early validation can abort a transaction quicker to reduce wasted work.
- *How to validate.* The most precise form of validation is to explicitly check whether committing transaction T would form dependency cycles with other committed transactions [42]. However, such graph-based validation is expensive to implement for in-memory databases. A practical alternative is OCC-style validation [28,57] which uses each transaction's physical commit-timestamp (cts) as its serialization order. Although such validation is conservative and has false aborts, it is fast. Prior work has also proposed validation based on logical commit-timestamps [68].

3.2 Decomposing existing CC algorithms

We take a deep dive to study existing algorithms through the the lens of our framework. At a high level, existing algorithms differ from each other by the distinct combinations of action choices they have, even though their choices remain the same regardless of state.

As summarized in Table 1, traditional 2PL [17] and OCC [28] algorithms both read the latest committed data. OCC does not wait to perform any accesses nor does it expose the writes. By contrast, 2PL exposes writes in order to block future conflicting accesses. We can approximate 2PL's blocking behavior by the action choice that makes transaction T wait for all its dependent transactions T_{dep} to commit before its data access. This approximation is slightly less aggressive than that of 2PL, which makes T wait for T' to commit if

the current access *will* make T dependent on T'. We use the term 2PL* to refer to 2PL with this approximated blocking. Sundial [69] handles write-write conflicts with 2PL and read-write conflicts with OCC; thus, it blocks write access until all its write dependencies T_{wdep} commit and has no blocking for reads. As for validation, traditional algorithms do it only at commit time, except for 2PL whose deadlock detection or prevention mechanism can be viewed as a form of early validation done at every access.

Apart from traditional CC, our framework also applies to a class of recently proposed algorithms including Callas RP [65], IC3 [60] and DRP [41]. These algorithms structure each transaction as a series of pieces [50], and try to pipeline the execution of these pieces to enhance performance under contention. As shown in Table 1, unlike traditional CC, they make a transaction's writes visible and allow reads of uncommitted data. Furthermore, they make transaction T wait before an access until T's dependent transactions finish execution up to a certain point, determined by applying a static analysis of the transaction workload.

Although our design for learnable CC (§4) does not support MVCC, we can nevertheless examine MVCC algorithms using our framework. Table 1 shows the actions made by MVTSO [2]. Other MVCC algorithms [30, 46, 66] have similar actions but use different validation methods. Under MVTSO, a transaction reads the largest committed version smaller than its timestamp. Writes are exposed so that future reads by transactions with larger timestamps will wait for this transaction to commit. MVTSO also performs a form of early-validation and aborts T if there exists $T' \in T_{dep}$ such that $T' \xrightarrow{rw} T$ and T' has been assigned a larger timestamp.

Not all CC algorithms can be expressed by our framework. In particular, our framework tracks dependencies and controls the interleaving of data access at runtime, and therefore cannot encode those CC algorithms that pre-define dependencies according to some globally-agreed ordering prior to execution, e.g. Calvin [56], Granola [7], Eris [31] and RoCoCo [42]. Moreover, our framework assumes that each access of the transaction is executed one after another by a single thread, and hence cannot encode algorithms like Bohm [13] that uses multiple threads to execute a single transaction.

4 Polyjuice Design

We design Polyjuice according to the framework of §3. The design consists of two parts: 1) a suitable policy space. 2) a training procedure to optimize the policy for a given workload. This section describes the policy space. The next section (§5) discusses training.

4.1 Overview

System architecture. Polyjuice is a multi-core in-memory database. There is no multi-version support. For each data object, Polyjuice stores the latest committed data as well as a per-object access list. The access list contains all un-committed writes that have been made visible, as well as read accesses. A transaction uses the access list to track the dependencies for each data access. Polyjuice uses a pool of workers that run concurrently: each worker executes a transaction and commits it according to the learned CC policy, which has been trained offline. Fig. 2 shows Polyjuice's system architecture.

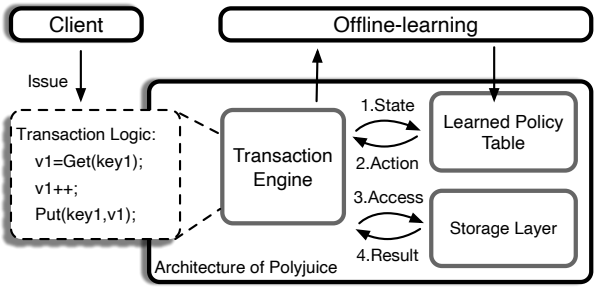


Figure 2: System architecture. Before executing a specific data access in the transaction, Polyjuice consults the learned actions in the policy table (step 1, 2 in the figure). Then, Polyjuice performs the access in the storage layer according to the actions.

Policy Representation. As discussed in § 3, we consider each learnable CC algorithm as a policy function p that maps from the *state space* (S) to the *action space* (A), $p : S \rightarrow A$. Both the state and action space consists of a number of dimensions; the size of the state/action space is exponential w.r.t. the number of dimensions.

We represent each policy function as a table: there are as many rows in the policy table as there are different states; there are as many columns as there are action dimensions. Such tabular representation is practical only if the state space is not too huge, which is the case in the workloads that we have studied. § 9 discusses the limitation of large state space and potential solutions.

For a given CC policy table, a cell $c_{i,j}$ at row i and column j indicates that for the access with execution context (state) i, the system should take the action given by cell $c_{i,j}$ for action type (aka dimension) j. In Polyjuice, each cell contains either a binary number for a binary action (e.g. whether to make writes visible or not), or an integer for a multi-valued action (e.g. how to wait for dependent transactions). Fig. 3 shows the CC policy table; details on its rows and columns are explained in §4.2 and 4.3. Polyjuice learns the backoff time for retrying aborted transactions separately (§4.5).

Policy-based Execution. In Polyjuice, the database is given the learned policy table with which to perform concurrency control. To execute a transaction according to the policy, Polyjuice looks up in the policy table at each data access to determine the corresponding set of actions. Some of these ac-

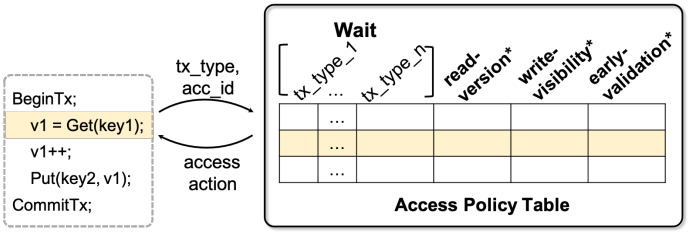


Figure 3: Policy table (* indicates a binary field).

tions are to be performed prior to the data access, e.g. whether and how long to wait, while others are to be done after the access, e.g. making a write potentially visible by appending it to the access list. After finishing execution, Polyjuice commits a transaction after performing the final validation to ensure serializability (§ 4.4).

4.2 CC policy: state space

The term *state* is from the RL literature. In our case, state can be viewed as the execution context of the current data access. Ideally, the state space should be able to distinguish execution contexts that are best served by different actions. It should also be limited in size so that the resulting policy table is not too huge and can be searched efficiently during training.

Polyjuice's state space contains the following information:

1. The type of the transaction being executed. For a given workload whose transactions are specified in stored procedures, the type can be identified by the stored procedure name.
2. Which access of the transaction is being executed. We use an integer access-id to identify each access. Access-id is determined by the static code location that invokes the access. Using static information for access-id provides a good trade-off: it can discriminate most accesses while avoiding blowing up the state space.

It is tempting to include other useful information, such as which type of access (read/write/commit) and which data table is being executed. Interestingly, for most workloads, both of these can be uniquely determined by the access-id and thus we omit them from the state space. We have also experimented with adding the contention level of the accessed data to the state space. However, we found that doing so only benefited a few contrived micro-benchmarks. In practical workloads including TPC-C and TPC-E, distinguishing transaction type and access-id is sufficient to capture the main contention characteristics. Even for artificial workloads, it is difficult to find a scenario where including contention level results in noticeable performance improvements. Including contention level makes it possible to differentiate accesses with the same access id. However, Polyjuice's wait action (§ 4.3) cannot take advantage of such differentiation.

Size of state space (a.k.a. number of different states). The state space size determines the number of rows in the CC policy table. Let n be the number of different transaction types in a workload, and d_1, d_2, ..., d_n be the number of static data accesses for transaction of type $1, 2, ..., n$. Then the state space size (i.e. number of different states) is: $d_1 + d_2 + ... + d_n$.

4.3 CC policy: action space

Polyjuice's action space contains knobs in two categories: interleaving control and validation.

Supported actions for interleaving control. There are three classes:

- *Wait*. This action is invoked *before* a read or write. How to specify how long the wait should be? A naive design is to use absolute time intervals, but this makes the wait action sensitive to execution time variations, resulting in fragile policies.

  Since the goal of waiting is to let another potentially conflicting transaction to go ahead with its data access, we quantify how long transaction T should wait by how much progress the transactions that T depends on have made so far. This design is inspired by existing protocols like Callas RP [65] and others [41, 60]. More concretely, we group transactions by type, and measure the execution progress of a transaction type by access-id. The special value NO_WAIT indicates no waiting. Suppose the wait action for transaction type X has access-id a, then transaction T must wait for all T's dependent transactions of type X to finish execution up to and including a. For a workload with n different types of transactions, the wait action consists of n access-ids, one for each transaction type.
- *Read-version*. This action has a binary choice: CLEAN_READ for reading the latest committed version, DIRTY_READ for reading the latest uncommitted (but visible) version. Although there may be more than one uncommitted copy of data, there is no point in reading an earlier version because doing so would result in more dependencies and higher abort likelihood.
- *Write-visibility*. This action is invoked *after* a write access and is also binary: PRIVATE keeps the write in the private buffer, PUBLIC makes all private writes buffered so far visible by appending them to the access list.

Supported actions for validation. Validation always happens before commit (§ 4.4). Polyjuice also supports the action of early-validation, which can occur *after* any read/write. If it's set, this binary-valued action checks if the reads and writes done since the last validation have violated serializability. Earlier accesses, which have passed previous early-validation, are likely to have already been serialized and thus not checked. Early-validation does not guarantee correctness but avoids wasting work by detecting non-serializable access early.

Polyjuice supports the wait action before early-validation. The encoding of the wait action is the same as that for

reads/writes. To reduce the action space, we consolidate the two kinds of wait actions into one. In particular, Polyjuice uses the wait action corresponding to the next access-id if early-validation is enabled for the current access-id.

Upon failing early-validation, Polyjuice retries the transaction from the point of its last successful validation. In order to reduce the cost of the failed validation, we defer appending reads and visible-writes to their corresponding access lists until a successful early-validation. Otherwise, failing early-validation means having to remove previously appended reads/writes from access lists, and to abort transactions that have read those discarded writes. Conceptually, we can separate the decision of early validation from that of appending reads/writes to access lists. However, in our experience, doing so complicates the implementation without improving the final learned CC performance.

Size of action space (a.k.a. number of different action choice combinations per state). Let n be the number of different transaction types in a workload, and d_1, d_2, ..., d_n be the number of static data accesses for transaction of type $1, 2, ..., n$. Then the number of different action choice combinations can be calculated as: $d_1 * d_2 * ... * d_n$(wait choices)$*2$(read-version)$*2$(write-visibility)$*2$(early-validation).

4.4 Validation for correctness

Polyjuice uses an OCC-style physical timestamp-based validation in the final commit phase to ensure correctness. To commit a transaction T with validation, a worker takes 4 steps: 1) it waits for all T's dependent transactions to commit (or abort). 2) it locks each record in T's writeset 3) it validates each record in the readset by checking two conditions; whether the version-id of the current committed version in the database is different from that kept in the readset, and whether the record is being locked by another transaction. If either condition is true, T is aborted. 4) if validation succeeds, it applies T's writes to the database along with their version-ids, and releases the locks.

Our validation algorithm is identical to that of Silo [57] except for two additional mechanisms which are crucial for correctness. First, we use a unique version-id for committed as well as uncommitted versions, because the latter may be read from the access list. Second, we add the additional first step of waiting for T's dependent transactions to finish committing.

We provide a brief correctness argument here. A more detailed proof is in the Appendix. We argue the correctness of Polyjuice by reduction to Silo: if Polyjuice commits a transaction, then Silo would also commit it. According to step-1, Polyjuice ensures that if a transaction T is committed successfully, then before T starts the validation, all of its dependent transactions (e.g. T_{dep}) have been committed. This allows us to prove that executing T is equivalent to executing another hypothetical transaction T' which starts execution after all transactions in T_{dep} commit, reads/writes the same data as T, and starts validation at the same time as when T starts its validation. Therefore, if T passes the validation in Polyjuice, T' can pass the validation of Silo and successfully commit itself.

4.5 Learning backoff time

Separate from the CC algorithm, it is also important for performance to use an appropriate backoff time for retrying an aborted transaction. Existing systems, e.g. Silo, use simple binary exponential backoff which doubles the backoff time with each failed attempt. This simple strategy is inadequate as it often results in backoff times that are too short in the first couple of retries but too large after several successive retries. Furthermore, this strategy does not distinguish between different transaction types when adjusting backoff times. This is suboptimal: intuitively, one can increase the backoff time more aggressively for a transaction type more prone to contention.

For learning the backoff time, Polyjuice uses a separate backoff policy table. The rows (a.k.a. state space) of this table enumerate 3 dimensions: 1) the transaction type 2) the status of the current execution (commit or abort). 3) the number of aborted attempts prior to the current execution with a fixed cutoff: our current implementation uses 0, 1 or 2 to indicate whether there has been 0, 1 or 2+ aborts so far. The action space of the backoff policy table is inspired by recent work on learnable congestion control in networking [22]. Specifically, a worker adjusts the backoff time for each transaction type multiplicatively whenever it commits/aborts a transaction:

$$backoff = \begin{cases} backoff \times (1 + \alpha_{t,i,aborted}), & abort \\ backoff / (1 + \alpha_{t,i,committed}), & commit \end{cases}$$

In the above equations, $\alpha_{t,i,committed}$ or $\alpha_{t,i,aborted}$ is the learned parameter (aka action) in the policy table for transaction type t, number of prior aborted attempts i and execution status *committed* or *aborted*. To enable easier training, we use bounded discrete values for α. In particular, α can be zero, resulting in unchanged backoff time.

5 Training Policies

Overview The policy space discussed in §4 is exponentially large: there are a^s different policies, where s is the number of different states and a is the number of different actions per state. The goal of training is to efficiently search for a good policy for a given workload.

Polyjuice performs training offline. During regular execution, Polyjuice logs executed transactions together with their inputs. Using a separate training machine, Polyjuice emulates the target workload by reissuing transactions with their logged

inputs. We measure a policy's commit throughput under the emulated workload.

Polyjuice uses Evolutionary Algorithm (EA) for training. We have also explored the policy-gradient method from the RL literature (§5.2). Despite EA's simplicity, we have found it to be more efficient than the alternative (§7.5).

5.1 Training using Evolutionary Algorithm

EA is an optimization approach to search for a solution with good fitness by evolving a population of individuals via nature-inspired mechanisms such as crossover, mutation, and selection [10, 16, 20]. In Polyjuice, the fitness of an individual (aka a candidate policy) corresponds to the policy's commit throughput under the given workload.

EA starts by initializing the first generation of the population. The size of the population for each iteration is a configurable hyperparameter. To create a new children generation, EA performs mutation on the policies (including CC and backoff policies) of the current generation (parents). It then evaluates the "fitness" of each mutated child by measuring its throughput. Finally, EA selects N individuals according to their fitness to survive to the next generation.

Mutation. EA mutates each cell of a parent's CC and backoff policy table independently with probability p. If the cell corresponds to a binary choice such as read-version or write-visibility, the mutation flips the choice. If the cell corresponds to an integer choice (e.g. any of the wait actions), the mutation varies the integer value by some distance uniformly sampled from the interval $[-\lambda, \lambda]$. The mutated integer is clipped to always lie within the valid range. The initial values of mutation probability (p) and mutation interval (λ) are configurable hyperparameters. We decrease p and λ gradually as the training progresses to facilitate convergence. This is akin to the decrease in learning rate in gradient descent methods or the gradual reduction of temperature in simulated annealing.

Crossover, another popular EA mechanism, is not effective in our context. Crossover endows a child's policy with some rows from one parent and some rows from the other parent. Unfortunately, such a child is likely to perform worse than either of its parents. This is because, in most good policies, the wait actions of different rows are not independent but highly correlated. Thus, mixing the rows of different policies often results in worse performance.

At the end of each iteration, EA chooses N individuals with the best performance from the current population to survive to the next iteration. In our experiments, this simple selection mechanism trains faster than tournament selection [10, 16, 20].

Warm start. Instead of using all random policies, we seed the initial population with several known good policies, including OCC, 2PL*, and Callas RP/IC3. These policies are likely not optimal for the given workload, but they provide some good initial policies to give EA a "warm start" in training.

5.2 Alternative training method

Some recent works have used RL training methods to solve systems problems such as task scheduling [36], adaptive video streaming [35], multi-GPU dataflow systems [39, 40], congestion control [22], etc. We have experimented with the policy-gradient method for training a parameterized stochastic policy [61]. More concretely, we parameterize the policy table by representing each table cell using one or a set of parameters to denote the probability distribution of the action values. Suppose the cell at coordinate i, j corresponds to some action with M possible choices, we use M parameters, $p_{i,j}^0, p_{i,j}^1, \ldots p_{i,j}^{M-1}$, which are fed into a softmax function to denote the probability distribution of M choices.

For training, each iteration samples a batch of policies according to the probability distribution specified by the current table parameters. We measure the throughput of each sampled policy and use it as the "reward" in RL. Policy gradient maximizes the expected reward by performing gradient descent [61]. Our way of applying policy gradient is inspired by [3]. We compare RL- and EA-based training in §7.5.

5.3 Training for real-world deployments

Since Polyjuice relies on offline training to optimize its policy for a specific workload, this raises the question of how to use it in the real-world with changing workloads. We acknowledge that Polyjuice is not suitable for very dynamic and unpredictable workloads. However, we observe that many real-world workloads are fairly *predictable* on a day-to-day basis after analyzing the trace of an e-commerce website. This has motivated us to suggest the following deployment strategy for Polyjuice.

Optimize for the peak workload. Real-world systems are provisioned for the anticipated peak workload. Hence, our goal is to use Polyjuice to improve commit throughput during the peak time, in which the server receives the most requests in a day. There is no need to optimize for non-peak workloads because an under-utilized database is not a bottleneck for application performance. Therefore, we only need to train the policy tailored to the peak workload, and run the same policy during non-peak times as well.

Predict and retrain. Our analysis of the real-world trace shows that one can predict tomorrow's peak workload characteristics using the statistics gathered from today's peak workload (§7.6). Given this observation, one can collect the trace of the peak hour today, retrain the policy based on the trace, and run this policy for tomorrow. Doing so naively requires Polyjuice to retrain the policy every day. We can defer retraining if the predicted peak workload does not differ significantly from the one targeted by the current policy. Our analysis of the real-world trace shows that the peak workload can remain stable for many days after a significant change.

Hence, deferral can greatly decrease the number of retraining times. One is right to be concerned that deferred training and prediction errors can result in running a policy optimized for a different workload than the actual one happening. We also evaluate the effect of this discrepancy in §7.6.

6 Implementation

The pseudocode of how Polyjuice executes a transaction is included in the Appendix. We implemented Polyjuice in C++ using the codebase of Silo [57] by replacing Silo's concurrency control mechanism with Polyjuice's policy-based algorithm. We implemented Polyjuice's offline training separately in Python (and RL-based training in TensorFlow). The result of training is the policy table, which is written to disk as a file and later loaded into memory by the C++ database. Each worker thread in the C++ database maintains a pointer to the in-memory policy table, and the Appendix includes the pseudocode on how the database executes and commits transactions according to the policy. When switching the policy, we reset the policy pointer in each worker thread. Polyjuice doesn't need to atomically switch the policy pointers of all threads. This is because Polyjuice's validation procedure can ensure correctness regardless of the policies used during execution.

Like Silo, transaction logic is written in C++ using a few API calls (e.g. Get/Put/CommitTx). Each Get/Put/CommitTx API call's access-id is its corresponding sort order based on the API invocation's line number. For range queries, our current prototype reuses Silo's existing mechanism which always reads the committed value.

7 Evaluation

7.1 Experimental setup

Hardware. Our experiments are conducted on a 56-core Intel machine with 2 NUMA nodes. Each NUMA node has 28 cores (Xeon Gold 6238R 2.20GHz) and 188GB memory.

Workloads. We use three benchmarks, TPC-C [5], TPC-E [6], and a micro-benchmark with ten types of transactions. In our experiments, each worker retries an aborted transaction indefinitely until success, to ensure that committed transactions adhere to the workload's specified mix ratio of different transaction types. If we had not done this and let a worker give up an aborted transaction and start a new one with a different type, we would incorrectly learn a policy that intentionally aborts some transaction types to maximize aggregate throughput.

Baselines for comparison . We compare Polyjuice with five existing algorithms: OCC (Silo) [57], 2PL [17], IC3 [60], Tebaldi [53] and CormCC [55]. For Silo and IC3, we use the authors' source code. For Tebaldi and CormCC, we simulate them in our codebase to provide an apples-to-apples comparison. For 2PL, we implement it in Silo's codebase with an optimized WAIT-DIE mechanism. The optimization avoids aborts if locks are acquired following a global order, as is the case with our TPC-C and microbenchmark.

Methodology. For the training, we use 300 iterations by default. After each iteration, we pick 8 policies from the current population. For each of them, we generate another 4 children policies and add them to the selection pool. Therefore, there are a total of $8 * 5 = 40$ policies at each iteration. To evaluate the performance of the learned policy as well as other baseline algorithms, we run the workload five times, with each run taking 30 seconds. By default, the graphs show the median.

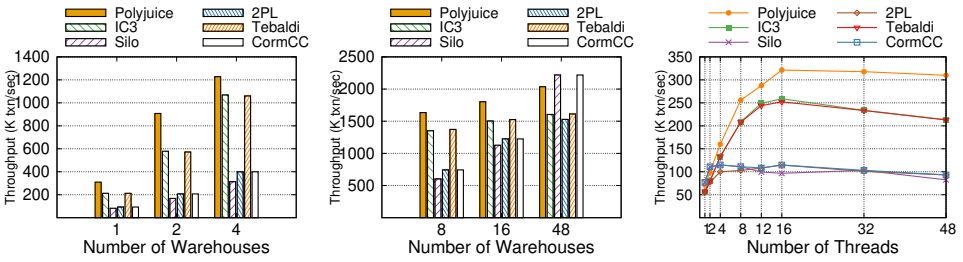
7.2 TPC-C

For the TPC-C benchmark, we evaluate the three read-write transactions only, as the remaining two read-only transactions can be processed with the snapshot mechanism derived from Silo. We vary the number of warehouses in the benchmark to change the level of contention.

By default, we use the 3-layer configuration for Tebaldi, which divides the read-write transactions into two groups (NewOrder, Payment vs. Delivery) isolated by 2PL [53]. Tebaldi's 2-layer configuration puts all read-write transactions into the same group, which is the same as IC3. We simulate CormCC according to its paper [55]. In particular, we partition the workload by warehouse so that all accesses to the same warehouse are protected by the same CC. Moreover, as all warehouses are inter-changeable in our benchmark, all partitions should also use the same CC protocol. Based on this observation, we measure the performance of 2PL and OCC, and pick the one with the better performance as the CC protocol for each partition.

Throughput. Fig. 4a and 4b show the throughput of various algorithms with 48 threads under different contention levels. Fig. 4a gives the throughput under high contention. Polyjuice achieves significant performance improvements. Specifically, with two warehouses, its throughput reaches 907K TPS, which is more than 1.5× of other algorithms. IC3 and Tebaldi have higher throughput than other existing algorithms because they can exploit a form of "pipelined" execution. Both have the same throughput, which differs from the original paper, as we disable their manual optimization for commutativity and uniqueness. Compared with IC3 and Tebaldi, Polyjuice achieves 56% improvement because of two factors: First, it can avoid unnecessary waiting because it uses the runtime information to infer the CC action, while IC3 only leverages the static information. Second, Polyjuice can either read dirty or clean versions of data. This flexibility enables it to achieve more efficient interleavings. We provide a detailed analysis with an example in § 7.3.

Fig. 4b shows the throughput under moderate and low contention. Polyjuice outperforms the others for 8 and 16 ware-



(a) High Contention (b) Moderate to Low Contention (c) Scalability (1 WH)

Figure 4: TPC-C Performance and Scalability

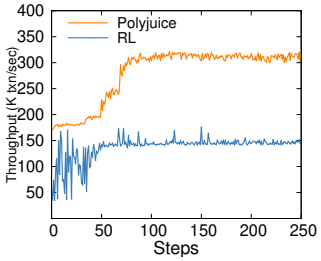
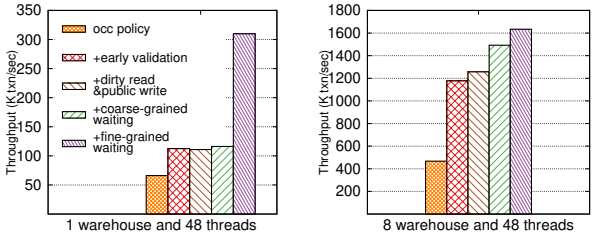


Figure 5: EA v.s. RL

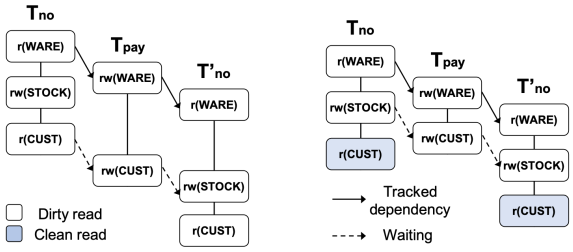
		Polyjuice	IC3	Tebaldi	Silo	2PL	CormCC
	Neworder	163/151/179/245	251/246/296/345	246/240/291/354	1084/20/62/263	450/31/49/178	450/31/49/178
Latency(μs)	Payment	163/151/181/252	247/242/291/340	242/236/285/348	6/4/9/24	658/19/97/1554	658/19/97/1554
AVG / P50 / P90 / P99	Delivery	172/167/194/269	156/152/177/223	155/151/175/208	108/101/120/248	183/145/279/621	183/145/279/621

Table 2: Latency for each transaction type in TPC-C with 1 warehouse and 48 threads



(a) High Contention (b) Moderate Contention

Figure 6: Factor Analysis On TPC-C Benchmark



(a) IC3 interleaving. (b) Polyjuice interleaving.

Figure 7: Polyjuice's learned policy results in a more efficient interleaving for TPC-C than IC3.

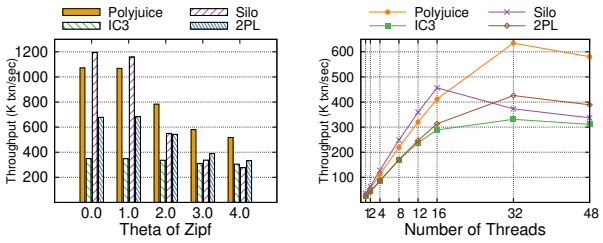
houses. For 48 warehouses, in which each worker corresponds to its local warehouse, Polyjuice is slightly slower (8%) than Silo, even though Polyjuice learns the same policy as Silo. This is because Polyjuice needs to maintain additional metadata in each tuple, which affects the cache locality.

Scalability. Fig. 4c shows the scalability of Polyjuice under high contention (1 warehouse). Polyjuice has the same scalability as IC3 and Tebaldi, which can scale to 16 threads. Compared with them, Silo and 2PL do not scale beyond four threads because they cannot exploit parallelism under high contention. CormCC also has the scalability issue because it is limited by the protocols (2PL and OCC) it uses.

Performance of each transaction type. We also study the throughput and latency for each type of read-write transaction with 1-warehouse and 48 threads (Table 2). For Polyjuice, the throughput of each type is 132K (NewOrder), 126K (Payment) and 11K (Delivery) TPS, which follows TPC-C specified ratio (45:43:4) very closely. This is because each worker retries an aborted transaction infinitely until it succeeds before starting a new transaction. Therefore, the ratio of the per-type commit throughput is exactly the same as how each worker generates these types. For the latency of NewOrder, Polyjuice has higher P99 latency than 2PL, but lower latency than Silo, IC3 and Tebaldi. For Delivery, the outcome is flipped: Polyjuice has lower P99 latency than 2PL, but higher latency than Silo, IC3 and Tebaldi. For Payment, Polyjuice has lower P99 latency than IC3, Tebaldi and 2PL.

Factor analysis. To better understand the advantages of Polyjuice, we perform a factor analysis to examine the benefits of different actions. We start with a policy including only the actions of OCC (Table 1). Then, we gradually add other actions into the action space and measure the performance improvements. We classify the waiting actions into coarse-grained waiting and fine-grained waiting. The former means the actions of waiting for the dependent transaction to commit and learning the backoff. The latter refers to waiting for a certain access of the dependent transaction to finish.

Fig. 6a and 6b show the factor analysis result with 1 and 8 warehouses. For the 1-warehouse workload, adding "early validation" into the action space can improve the performance by 70%, because it can detect the conflicts earlier and reduce the retry cost. Polyjuice gets a performance boost after applying fine-grained waiting actions (116K to 309K TPS) due to full exploitation of the potential parallelism. However, each action



(a) Throughput (b) Scalability (θ is 3.0)

Figure 8: TPC-E Performance and Scalability

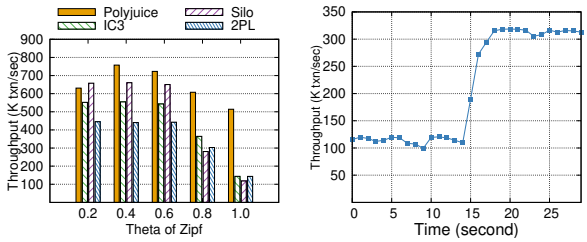


Figure 9: Micro-benchmark with 10 tx types.

Figure 10: Throughput during policy switch.

has a different effect factor with different workloads. For the 8-warehouse workload, adding "early validation" achieves larger improvement (467K to 1177K TPS) than others.

7.3 A case study of learned policy

We analyze an example learned policy to understand how it outperforms existing CC algorithms.

Fig. 7 shows an example of how IC3 and our learned policy mediate the data access of 3 concurrently executing transactions: T_{no} (NewOrder), T_{pay} (Payment) and T'_{no} (NewOrder). All three access the same warehouse. Fig. 7 shows a few crucial data accesses for each transaction: For NewOrder transactions (T_{no}, T'_{no}), these accesses are: read from WAREHOUSE table (r(WARE)), followed by an update to STOCK table (rw(STOCK)), and finally read from CUSTOMER table (r(CUST)). The crucial accesses of Payment (T_{pay}) are: update to WAREHOUSE (rw(WARE)) and update to CUSTOMER (rw(CUST)).

The three transactions conflict because they access the same record in WAREHOUSE. Fig. 7 shows a specific dependency pattern that can arise from their WAREHOUSE access, $T_{no,r(WARE)} \rightarrow T_{pay,rw(WARE)} \rightarrow T'_{no,r(WARE)}$ as all WAREHOUSE accesses use dirty reads. As shown in Fig. 7a, to avoid the dependency cycle, IC3 makes T_{pay}'s read of CUSTOMER wait for T_{no}'s CUSTOMER update to finish. This is because IC3 always uses dirty reads, so $T_{pay,rw(CUST)}$ must be ordered after $T_{no,r(CUST)}$ in accordance with their WAREHOUSE access' ordering. IC3 also makes T'_{no} STOCK update wait for T_{pay}'s CUSTOMER update, even though these two access different tables. This is because IC3 only tracks the immediate dependency: by waiting for T_{pay}'s CUSTOMER update, it ensures that T_{no} and T'_{no} will not form a dependency cycle even though T'_{no} is not aware of the transitively dependent T_{no}.

Fig. 7b shows the interleaving obtained by Polyjuice, which is more efficient. Unlike IC3, the learned policy makes T_{pay}'s CUSTOMER update wait for T_{no}'s STOCK access which is earlier than $T_{no,r(CUST)}$. This shorter wait works because the learned policy also makes T_{no}'s CUSTOMER read a committed version, which helps avoid the conflict between $T_{no,r(CUST)}$ and $T_{pay,rw(CUST)}$. This is in contrast to IC3, which makes $T_{no,r(CUST)}$ perform a dirty read. The learned policy still makes T_{no}'s STOCK update wait for T_{pay}'s CUSTOMER update like IC3 does, but the overall interleaving is more efficient.

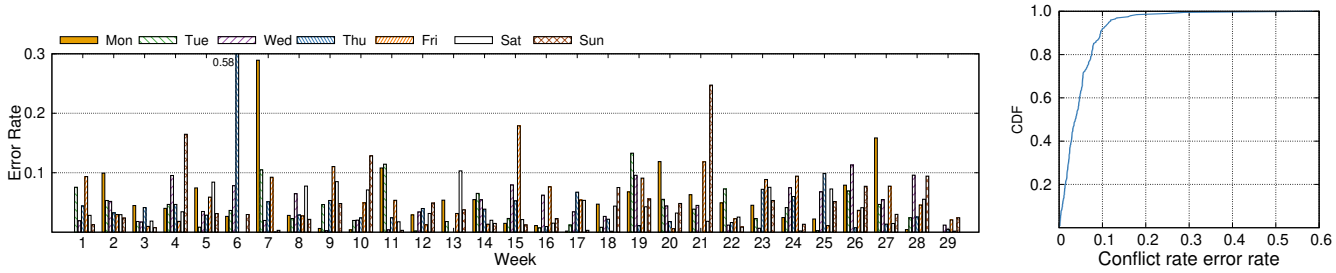
Apart from IC3, neither CormCC nor Tebaldi can exploit this interleaving. CormCC does not allow dirty reads. Tebaldi uses the same action (either dirty or clean read) for all accesses within a transaction. Fig. 7b's interleaving requires using dirty reads for NewOrder's WAREHOUSE access and clean reads for CUSTOMER access.

7.4 Bigger benchmarks

We use two bigger benchmarks to check if Polyjuice can learn a CC policy in a much larger search space. The first benchmark includes three read-write transactions from TPC-E, TRADE_ORDER, TRADE_UPDATE and MARKET_FEED. Compared with the state space of TPC-C (total 26 states), this benchmark is much more complex (total 65 states). The second benchmark is a micro-benchmark with ten types of transactions each with 8 accesses performing random updates (total 80 states). For each type of transaction, the last operation updates records in a unique table to distinguish it from other types. We build this benchmark because the action space grows exponentially with increasing transaction types.

TPC-E. We vary the contention in TPC-E by controlling the updates on SECURITY table. Specifically, all updates follow the Zipf distribution and we vary the θ of Zipf from 0.0 to 4.0 to increase the contention. We didn't evaluate Tebaldi as it doesn't provide a manual grouping strategy for TPC-E. Similarly, we didn't evaluate CormCC as it is unclear how to partition the data for TPC-E.

As shown in Fig. 8a, the throughput of Polyjuice is 42%, 49% and 55% higher than other algorithms when contention is high ($\theta = 2, 3, 4$). Unlike TPC-C, in this experiment, the improvement of Polyjuice is mainly attributed to the learned backoff. Specifically, Polyjuice learns a different backoff mechanism from Silo's design. We find out that in Silo, the frequent aborts of TRADE_ORDER result in a large backoff under high contention and the system spends a lot of time waiting before retry. In Polyjuice, for TRADE_ORDER transaction, it wouldn't increase the backoff even though



(a) Error rates for each day

(b) CDF of error rates

Figure 11: Error rates of conflict rate

the transaction is aborted. Although the abort rate remains high compared with Silo, the overall throughput is higher. Fig. 8b shows the scalability of Polyjuice under TPC-E with $\theta = 3$. Polyjuice's performance can scale to 18.5× with 48 threads over that with a single thread, which is higher than IC3 (12.3×) and 2PL (16.6×). Silo (9.4×) does not scale due to the frequent transaction aborts.

Micro-Benchmark with 10 Types of Transactions. For this benchmark, we change the access distribution of the first operation to vary the contention level. Specifically, we change the θ of Zipf from 0.2 to 1.0 in the range of 4K. Other operations randomly update the records in the range of 10M, which results in little contention. Fig. 9 shows the result, Polyjuice's throughput is at least 66% higher than other concurrency control mechanisms under high contention scenarios. This is because the learned policy pipelines the operations on some of the high-contention records while optimizing the waits for low-contention records.

7.5 Training

We have also implemented policy-gradient based RL training for the same workload. We initialize RL with an IC3-like policy to improve its training under this high contention workload. The initialization sets the parameters corresponding to IC3 actions with a high probability (in our case, 80%). The comparison result is shown in Fig. 5. The RL agent converges after around 100 iterations, but the throughput of the learned policy is only 178K TPS. In contrast, Polyjuice can learn a 309K TPS policy in 100 iterations. Our training runs on a single machine for now; each iteration takes 80 seconds, most of which are spent on evaluating policy performance.

7.6 Coping with real-world workloads

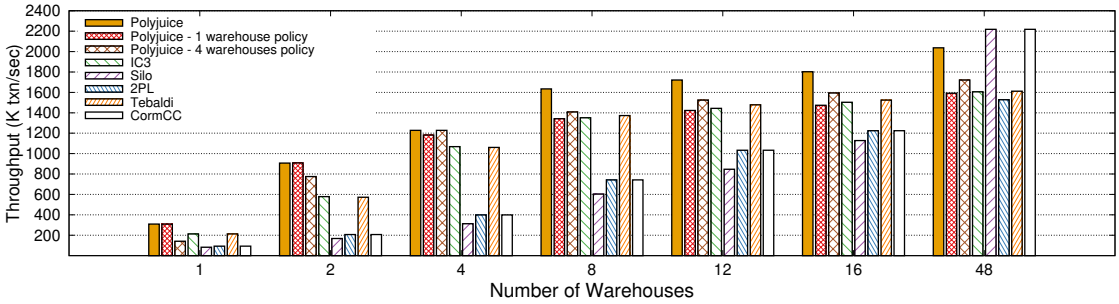
7.6.1 Trace analysis

The trace. Our analysis is based on the trace of a real-world e-commerce website, downloaded from Kaggle [24]. The trace includes a log of requests sent to the web server, including the request time and several parameters. There are three types of requests: `VIEW`, for when a user views a product; `CART`, for when a user adds a product to the shopping cart; and `PURCHASE`, for when a user purchases a product. As `VIEW` corresponds to a read-only request, we only include the two types of read-write requests `CART` and `PURCHASE` in our analysis.

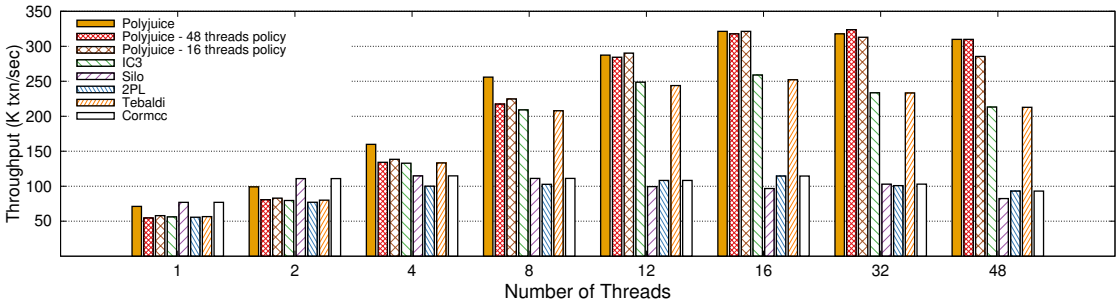
Workload predictability. For this analysis, we extract all the logged requests from Oct. 7th 2019 to Apr. 26th 2020 (29 weeks). After removing 7 invalid days, there are 196 days in total. We only consider the peak-hour workload for each day, since there is no need to optimize settings when the database is under-utilized and its commit throughput is limited by the incoming request rate instead of the CC performance.

As proposed in § 5.3, we predict tomorrow's peak workload characteristic to be the same as today's peak. How accurate is such a prediction? For our analysis, we characterize a workload by its contention level, which has the most effect on the learned policy. However, since the trace does not contain information on how long each request executes, we approximate the likelihood of contention by considering two requests to be in conflict with each other if they are sent by different users but operate on the same product id during some time window. We define $conflict\_rate = conflict\_requests/total\_requests$ within n minutes. In our analysis, we set $n = 5$ and split an hour into 12 intervals. We use the mean of the 12 conflict rates to represent the contention in this hour and pick the hour with the most requests as the peak workload in a day. We note that *conflict_rate* is heavily influenced by the request rate; the bigger the request rate, the higher the measured conflict rate.

Fig. 11 shows the error when predicting tomorrow's peak hour contention level using today's peak hour statistics. The error rate is calculated as $error\_rate = abs((tomorrow - today)/today)$. The smaller the *error_rate* is, the closer the next day's peak workload contention matches that of today. Fig. 11a shows the error rate of the conflict rates for all 195 days (except for the first day), and Fig. 11b shows the CDF of the error rates distribution. We can see that, there are only 3 days when the error rate of prediction is larger than 20%. After manually checking these 3 days, we find out that they are due to a significantly higher or lower request rate, which affects the conflict rate.



(a) TPC-C, 48 threads.



(b) TPC-C, 1 warehouse.

Figure 12: Throughput under different workloads

We also analyze how frequently one needs to retrain. As suggested in § 5.3, we assume retraining is deferred until the predicted conflict rate differs from the one used for training the current policy by 15%. For the trace analyzed, we only need to retrain 17 times to cover a period of 195 days.

7.6.2 Cost of policy switching

We evaluate the cost of switching the policy in terms of: 1) how long it takes to fully switch the policy 2) whether commit throughput is affected by policy switching. The result is shown in Fig. 10. We run the TPC-C 1 warehouse workload with 48 threads, and plot the throughput for each second. At the beginning, we run the workload with the OCC policy. Starting in the 15th second, we switch the policy to the one optimized for 1 warehouse. The result shows that it takes about 3 seconds to fully switch to a new policy, and switching does not negatively impact performance. In fact, because we are switching to a better policy, the performance quickly improves during switching.

7.6.3 Running a policy trained on a different workload

We also study what happens if the workload optimized by the policy differs from the actual one being executed. For these TPC-C experiments, we use fixed learned policies and measure their performance under various workloads different from those used in training.

In the first set of experiments, we use two fixed policies, which are trained using 48 threads on 1 warehouse or 4 warehouses. Fig. 12a shows the performance of fixed policies as we vary the number of warehouses, compared to existing algorithms and Polyjuice that is always trained on the correct workload. If the evaluation workload is different from the workload used for training, the fixed policies can be sub-optimal. For example, the performance of Polyjuice (1-warehouse) is 71% of Silo under 48 warehouses. However, the performance differences between fixed and optimal policies are small when the evaluation workload is not too far off from the training workload.

In the second set of experiments, we use fixed policies trained on 1 warehouse using 48 or 16 threads. Fig. 12b shows the performance of fixed policies as we vary the number of threads. The results are similar, in that a trained policy is fairly robust to training and evaluation workload mismatch.

8 Related Work

Concurrency control. We can categorize recent CC works according to their design choices. 1) Scheduling based CC: IC3 [60], Callas [65], DPR [41] and RoCoCo [42] allow ongoing transactions to expose their writes and track dependencies at runtime, then schedule the read/write operations according to the tracked dependencies. Ding et al. [11] schedules read operation after conflicting transaction's commits to avoid aborts for OCC protocol. 2) Deterministic databases: Gra-

nola [7], Deterministic CC [15,47,48,56] and Eris [32] schedule a transaction's execution according to a predetermined order. PWV [14] adds early write visibility to the deterministic CC to further improve the performance. 3) Changing the validation algorithm to avoid unnecessary aborts: TicToc [68] avoids unnecessary aborts by using logical timestamps for validation. BCC [70] changes the validation phase by detecting a special pattern. 4) Partially rolling back to reduce the abort cost [63].

In addition, there are a number of works applying MVCC into their systems. Bohm [13] combines the MVCC with deterministic CC to achieve non-blocking operations. Cicada [33] uses logical timestamps with MVCC to increase the possibility of constructing safe interleavings. Obladi [8] integrates MVCC on top of ORAM to provide security along with high performance.

All above CC protocols leverage a fixed set of design choices. Compared to them, Polyjuice is able to adapt the design choices according to the characteristics of a given workload. Some work [43, 54, 72] focus on distributed databases, which must do replication in addition to concurrency control. They propose new algorithms to handle inconsistent orderings in both concurrency control and replication.

Hybrid concurrency control. There are existing works that combine multiple concurrency control mechanisms for better performance. MOCC [59] develops a specific algorithm to combine OCC and 2PL for high-contention workloads. Sundial [69] proposes a new hybrid CC algorithm based on 2PL and OCC with logical timestamps. CormCC [55] proposes a more general hybrid method by formalizing all CC into four phases. Each operation can use any CC's policy as long as all CCs perform each phase according to the same order. Tebaldi [53] groups transactions and assigns different CC protocols to each group. However, existing algorithms are either specific for combining OCC or 2PL, or need programmers to provide heuristics to choose the execution policy for each operation. Compared to them, Polyjuice is able to automatically adapt the policy for each operation according to the workload.

Learned systems. Many system optimizations can be done by machine learning models trained from historical data. In the area of databases, examples include cardinality estimation [25, 29, 45, 62], join order planning [27, 37, 44] and configuration tuning [58]. Besides databases, works have been done to improve buffer management systems [4], sorting algorithms [73], memory page prefetching [19, 71] and memory control [21], task scheduling [26], CPU scheduling [51], locking priority [12] and cache replacement [52]. Although these works try to leverage machine learning to make systems self-aware, but none of them targets on the concurrency control. Thus, they have different model design from Polyjuice.

9 Discussion

As a first attempt on learnable CC, Polyjuice has limitations, some of which we hope to address in the future.

Not suitable for rapidly changing workloads. In our experience, training takes on the order of several hundred seconds. Thus, Polyjuice is not suitable for scenarios in which workload changes quicker than every few minutes.

Inaccurate workload emulation. Training reissues executed transactions with their logged inputs. However, since transaction interleavings during training differ from that of the original execution, a transaction's outputs also differ. Polyjuice works only if such emulation inaccuracies do not significantly affect the workload access pattern.

Large state space. Polyjuice represents a^s potential policies in a table format, where s is the number of different states and a is the number of different actions per state. As the number of transactions and the number of accesses in each transaction increase in the workload, both s and a increase. The resulting much enlarged search space will make training via EA less effective. One potential solution is to follow the breakthrough of deep reinforcement learning, and use a function approximator like a deep neural network to approximate the policy table with parameters far fewer than the number of table cells. It is a well-known challenge to make deep RL work effectively.

More expressive policy space. There are several interesting directions to expand the policy space, such as supporting multi-version databases, explicit CPU scheduling of execution, fine-grained instead of binary contention levels.

Weaker and mixed isolation levels. Polyjuice currently only guarantees serializability. Some applications can work with weaker or mixed isolation levels [9, 23, 34, 38, 64]. It is an interesting extension to generalize to these scenarios.

Acknowledgements

Chien-chin Huang and Minjie Wang contributed valuable ideas in the early stage of this project. We thank the anonymous reviewers, esp. our shepherd Deniz Altınbüken, for the helpful comments. Jiachen Wang, Huan Wang, Zhaoguo Wang and Haibo Chen were supported by National Key Research and Development Program of China (No. 2020AAA0108500), National Natural Science Foundation of China (No. 61902242), and the High-Tech Support Program from Shanghai Committee of Science and Technology (No. 19511121100). Ding Ding, Conrad Christensen, and Jinyang Li were supported by NSF grant 1816717, and a gift from NVIDIA and AMD. Zhaoguo Wang (zhaoguowang@sjtu.edu.cn) and Jinyang Li (jinyang@cs.nyu.edu) are the corresponding authors.

References

[1] Atul Adya. Weak consistency: A generalized theory and optimistic implementations for distributed transactions. *Ph.D. Thesis*, 1999.

[2] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *Computing Surveys*, 13(2), 1981.

[3] Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware. In *International Conference on Learning Representations (ICLR)*, 2019.

[4] Xinyun Chen. Deepbm: A deep learning-based dynamic page replacement policy.

[5] The Transaction Processing Council. TPC-C Benchmark. http://www.tpc.org/tpcc/.

[6] The Transaction Processing Council. TPC-E Benchmark. http://www.tpc.org/tpce/.

[7] James Cowling and Barbara Liskov. Granola: low-overhead distributed transaction coordination. In *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, pages 223–235, 2012.

[8] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious serializable transactions in the cloud. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 727–743, 2018.

[9] Natacha Crooks, Youer Pu, Nancy Estrada, Trinabh Gupta, Lorenzo Alvisi, and Allen Clement. Tardis: A branch-and-merge approach to weak consistency. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1615–1628, 2016.

[10] Lawrence Davis. Handbook of genetic algorithms. 1991.

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

[12] Jonathan Eastep, David Wingate, Marco D Santambrogio, and Anant Agarwal. Smartlocks: lock acquisition scheduling for self-aware synchronization. In *Proceedings of the 7th international conference on Autonomic computing*, pages 215–224, 2010.

[13] Jose M Faleiro and Daniel J Abadi. Rethinking serializable multiversion concurrency control. In *VLDB*, 2014.

[14] Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. High performance transactions via early write visibility. *Proceedings of the VLDB Endowment*, 10(5):613–624, 2017.

[15] Jose M Faleiro, Alexander Thomson, and Daniel J Abadi. Lazy evaluation of transactions in database systems. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 15–26, 2014.

[16] David E Goldberg and John H Holland. Genetic algorithms and machine learning. *Machine learning*, 3(2), 1988.

[17] J. N. Gray, R. A Lorie, and G. R. Putzolu. Granularity of locks in a shared data base. In *VLDB*, 1975.

[18] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1992.

[19] Milad Hashemi, Kevin Swersky, Jamie A Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. Learning memory access patterns. *arXiv preprint arXiv:1803.02329*, 2018.

[20] John Henry Holland et al. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[21] Engin Ipek, Onur Mutlu, José F Martínez, and Rich Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 39–50. IEEE Computer Society, 2008.

[22] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. A deep reinforcement learning perspective on internet congestion control. In *International Conference on Machine Learning*, pages 3050–3059, 2019.

[23] Gowtham Kaki, Kartik Nagar, Mahsa Najafzadeh, and Suresh Jagannathan. Alone together: Compositional reasoning and inference for weak isolation. In *45th Symposium on Principles of Programming Languages (POPL)*, 2018.

[24] Michael Kechinov. ecommerce behavior data from multi category store. https://www.kaggle.com/mkechinov/ecommerce-behavior-data-from-multi-category-store.

[25] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677*, 2018.

[26] Tim Kraska, Mohammad Alizadeh, Alex Beutel, E Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. Sagedb: A learned database system. CIDR, 2019.

[27] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196*, 2018.

[28] H. T. Kung and John Robinson. On optimistic methods for concurrency control. In *ACM Transactions on Database Systems (TODS)*, 1981.

[29] M Seetha Lakshmi and Shaoyu Zhou. Selectivity estimation in extensible databases-a neural network approach. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 623–627. Morgan Kaufmann Publishers Inc., 1998.

[30] Justin Levandoski, David Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. High performance transactions in deuteronomy. In *Conference on Innovative Data Systems Research (CIDR)*, 2015.

[31] Jialin Li, Ellis Michael, and Dan Ports. Eris: Coordination-free consistent transactions using network multi-sequencing. In *SOSP*, 2017.

[32] Jialin Li, Ellis Michael, and Dan RK Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 104–120, 2017.

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

[34] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416, 2011.

[35] H. Mao, R. Netravali, and M. Alizadeh. Neural adaptive video streaming with pensieve. In *SIGCOMM*, 2017.

[36] H. Mao, M. Schwarzkopf, S. Venkatakrishnan, Z. Meng, and M. Alizadeh. Learning scheduling algorithms for data processing clusters. In *SIGCOMM*, 2019.

[37] Ryan Marcus and Olga Papaemmanouil. Deep reinforcement learning for join order enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, page 3. ACM, 2018.

[38] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. I can't believe it's not causal! scalable causal consistency with no slowdown cascades. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 453–468, 2017.

[39] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V Le, and Jeff Dean. A hierarchical model for device placement. 2018.

[40] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pages 2430–2439. JMLR.org, 2017.

[41] Shuai Mu, Sebastian Angel, and Dennis Shasha. Deferred runtime pipelining for contentious multicore software transactions. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

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

[43] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating concurrency control and consensus for commits under conflicts. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 517–532, 2016.

[44] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S Sathiya Keerthi. Learning state representations for query optimization with deep reinforcement learning. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*, page 4. ACM, 2018.

[45] Yongjoo Park, Shucheng Zhong, and Barzan Mozafari. Quicksel: Quick selectivity learning with mixture models. *arXiv preprint arXiv:1812.10568*, 2018.

[46] Dan Ports and Kevin Grittner. Serializable snapshot isolation in postgresql. In *VLDB*, 2012.

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

[48] Kun Ren, Dennis Li, and Daniel J Abadi. Slog: serializable, low-latency, geo-replicated transactions. *Proceedings of the VLDB Endowment*, 12(11):1747–1761, 2019.

[49] L. Sha, J.P. Lehoczky, and E.D. Jensen. Modular concurrency control and failure recovery. *IEEE transactions on Computers*, 37(2), 1988.

[50] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. Transaction chopping: Algorithms and performance studies. *ACM Transactions on Database Systems (TODS)*, 20(3), 1995.

[51] Yangjun Sheng, Anthony Tomasic, Tieying Sheng, and Andrew Pavlo. Scheduling oltp transactions via machine learning. *arXiv preprint arXiv:1903.02990*, 2019.

[52] Zhenyu Song, Daniel S Berger, Kai Li, and Wyatt Lloyd. Learning relaxed belady for content distribution network caching. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 529–544, 2020.

[53] Chunzhi Su, Natacha Crooks, Cong Ding, Lorenzo Alvisi, and Chao Xie. Bringing modular concurrency control to the next level. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 283–297, 2017.

[54] Adriana Szekeres, Michael Whittaker, Jialin Li, Naveen Kr Sharma, Arvind Krishnamurthy, Dan RK Ports, and Irene Zhang. Meerkat: multicore-scalable replicated transactions following the zero-coordination principle. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–14, 2020.

[55] Dixin Tang and Aaron J Elmore. Toward coordination-free and reconfigurable mixed concurrency control. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 809–822, 2018.

[56] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12, 2012.

[57] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, 2013.

[58] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1009–1024. ACM, 2017.

[59] Tianzheng Wang and Hideaki Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores (extended version). *Hewlett Packard Labs Technical Report HPE-2016*, 58, 2016.

[60] Zhaoguo Wang, Shuai Mu, Yang Cui, Han Yi, Haibo Chen, and Jinyang Li. Scaling multicore databases via constrained parallel execution. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1643–1658, 2016.

[61] Ronald Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8, 1992.

[62] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. Towards a learning optimizer for shared clouds. In *Proceedings of the 45th International Conference on Very Large Data Bases (VLDB), page to appear*, 2019.

[63] Yingjun Wu, Chee-Yong Chan, and Kian-Lee Tan. Transaction healing: Scaling optimistic concurrency control on multicores. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1689–1704, 2016.

[64] Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan. Salt: Combining {ACID} and {BASE} in a distributed database. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 495–509, 2014.

[65] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. High-performance acid via modular concurrency control. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 279–294, 2015.

[66] Maysam Yabandeh and Daniel Gómez Ferro. A critique of snapshot isolation. 2012.

[67] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. In *PVLDB*, 2014.

[68] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1629–1642, 2016.

[69] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sanchez, Larry Rudolph, and Srinivas Devadas. Sundial: Harmonizing concurrency control and caching in a distributed oltp database management system. In *PVLDB*, 2018.

[70] Yuan Yuan, Kaibo Wang, Rubao Lee, Xiaoning Ding, Jing Xing, Spyros Blanas, and Xiaodong Zhang. Bcc: reducing false aborts in optimistic concurrency control with low cost for in-memory databases. *Proceedings of the VLDB Endowment*, 9(6):504–515, 2016.

[71] Yuan Zeng and Xiaochen Guo. Long short term memory based hardware prefetcher: a case study. In *Proceedings of the International Symposium on Memory Systems*, pages 305–311. ACM, 2017.

[72] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. Building consistent transactions with inconsistent replication. *ACM Transactions on Computer Systems (TOCS)*, 35(4):1–37, 2018.

[73] Hanqing Zhao and Yuehan Luo. An $o(n)$ sorting algorithm: Machine learning sorting. *arXiv preprint arXiv:1805.04272*, 2018.

Algorithm 1 Transaction execution in Polyjuice

```
1:  function PUT(k, v, T, tx-type, acc-id)
2:      // k: key, v: value, T: transaction object
3:      // tx-type: transaction type, acc-id: access-id
4:      r ← db.Lookup(k)
5:      action ← policy.Lookup(tx-type, acc-id, ACCESS)
6:      WaitUntil(action.waits)
7:      T.buffer.append(k,v, WRITE)
8:      T.wset.append(k,v)
9:      if action.write_visible == PUBLIC then
10:         if Early_Validate(T.buffer, acc-id) fails then
11:             rollback T.wset, T.rset
12:             T.buffer ← {}
13:             goto last point of successful validation
14:         else AppendToAccessList(T.buffer)
15:         end if
16:         T.buffer ← {}
17:     end if
18: end function
19: function GET(k, T, tx-type, acc-id)
20:     r ← db.Lookup(k)
21:     action ← policy.Lookup(tx-type, acc-id, ACCESS)
22:     WaitUntil(action.waits)
23:     if action.read_version == DIRTY_READ then
24:         v ← FindLastWrite(r.acc_list)
25:     else v ← r.data
26:     end if
27:     T.buffer.append(k, v, READ)
28:     T.rset.append(k,v)
29:     if action.early_validate then
30:         if Early_Validate(T.buffer, acc-id) fails then
31:             rollback T.wset, T.rset
32:             T.buffer ← {}
33:             goto last point of successful validation
34:         else AppendToAccessList(T.buffer)
35:         end if
36:         T.buffer ← {}
37:     end if
38: end function
39: function EARLY_VALIDATE(T.buffer, acc-id)
40:     action ← policy.Lookup(tx-type, acc-id, VALID)
41:     WaitUntil(T.deps)
42:     Validate(T.rset, T.wset)
43: end function
44: function COMMIT(T, tx-type)
45:     // Add all dirty reads in T.buffer to T.deps.
46:     WaitUntil(T.deps)
47:     if Validate(T.rset, T.wset) fails then
48:         retry from beginning
49:     else atomically write T.wset to db
50:     end if
51: end function
```

A Proof

We give a sketch of the correctness argument.

1) Polyjuice never commits a transaction that has read data from some aborted transactions.

This is because if a read operation passes validation, the read version must be committed (Lemma-1).

2) All committed transactions are serializable.

We use proof-by-contradiction to show there cannot be cycles in any serialization graph (Lemma-3).

Lemma 1 ***commit read validation:*** *for transaction T_r, if T_r reads the value of R written by T_w with version v_w (both committed or uncommitted), then T_r can successfully pass the validation on R if only if T_w finally commits version v_w of R.*

Proof Assume v_w hasn't been successfully committed (including Subcase-1: T_w has aborted itself, Subcase-2: T_w has been successfully committed, but T_w commits another version v'_w (different from v_w) on R.). There doesn't exist another version v' of R which shares the same version-id as v_w (Lemma-2), and thus T_r cannot pass the validation on R.

□

Lemma 2 ***unique version-id:*** *for any record, there doesn't exist two different versions(including both committed and uncommitted version) v_1 != v_2, such that v_1's version-id == v_2's version-id.*

Proof Assume v_i and v_j are version-ids of two different versions to some record, created by T_i and T_j. Specifically, v_i includes the txn-id of T_i and the seqno if T_i publics this version before commit, ditto for v_j.

- Case-1 T_i != T_j, we never assign a txn-id twice.
- Case-2 T_i == T_j, since a txn never commits twice, there must be a version published before T_i's commit. For each version published before commit, we assign a unique seqno.

□

For the proof of serializability, we also use the following definitions:

Committed read/write: the read/write value associated with a committed txn.

Serialization graph: the graph consists of only committed transactions. The edges in the graph have 3 types, based on conflicts between committed reads/writes.

$T_i \xrightarrow{ww} T_j$: T_j's committed write overwrites T_i's committed write to the same record in the data store.

$T_i \xrightarrow{wr} T_j$: T_j's committed read is the same as T_i's committed write to the same record in the data store.

$T_i \xrightarrow{rw} T_j$: T_j's committed write overwrites the version which is T_i's committed read to the same record in the data store.

Serializability: there is no cycle in the serialization graph.

Proof sketch: Suppose there exists some violation of serializability, thus there exists a cycle in some serialization graph. Suppose the cycle is $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$. We use a lemma (proven later) that if $T_i \rightarrow T_j$, then T_i should acquire all locks (end-of-lock-stage) in the commit phase before T_j acquire all locks (end-of-lock-stage). Let's use < to indicate happens-before relationship. Thus, T_1's end-of-lock-stage < T_2's end-of-lock-stage < ... < T_1's end-of-lock-stage, which forms a contradiction.

Lemma 3 *Given an edge $T_i \rightarrow T_j$ in the serialization graph, then T_i's end-of-lock-stage < T_j's end-of-lock-stage, a.k.a. T_i acquires all locks before T_j acquires all locks.*

We prove this case by case, according to 3 edge types: $\xrightarrow{ww}$, $\xrightarrow{wr}$ or $\xrightarrow{rw}$. For concreteness, let's assume the conflict edge is associated with record R.

- Case-1 is for $T_i \xrightarrow{ww} T_j$. This implies "T_i acquires R's lock < T_i writes to R < T_i releases R's lock < T_j acquires R's lock", and thus we can have "T_i's end-of-lock-stage < T_j's end-of-lock-stage".
- Case-2 is for $\xrightarrow{wr} T_j$.
  - Subcase-2-1 is for T_j reads the clean (committed) value of R, which is committed by T_i. This implies T_i's acquires R's lock < T_i's write < T_j's read, which happens before T_j starts its commit phase, and thus T_i's end-of-lock-stage < T_j's end-of-lock-stage.
  - Subcase-2-2 is for T_j reads the dirty (uncommitted) value of R, which is finally committed by T_i. Since T_j will wait for all the direct dependent transactions to commit/abort before it enters the commit phase, T_j needs to wait for T_i to finish commit before T_j starts its commit phase, and thus T_i's end-of-lock-stage < T_j's end-of-lock-stage.
- Case-3 is for $T_i \xrightarrow{rw} T_j$. Let's assume T_i read the version installed by T_c. To prove "T_i's end-of-lock-stage < T_j's end-of-lock-stage", we first prove that "T_i's version check on R < T_j acquires R's lock".
  - Subcase-3-1 is for T_j releases R's lock < T_i's version check on R, which means T_j commits its writes to R before T_i validates R. According to the definition, $T_i \xrightarrow{rw} T_j$ implies that T_j overwrites T_c's committed write, which is read by T_i (Lemma-1). Thus, we have "T_c's write < T_j's write < T_i's version-check". For this order, T_i will fail to validate R, this is because T_j installs a version id different from T_c (Lemma-2) before T_i's validation.
  - Subcase-3-2 is for T_i's version check on R < T_j releases R's lock. This includes the following two cases:
    * First, "T_j acquires R's lock < T_i's version check on R".
    * Second, "T_i's version check on R < T_j acquires R's lock".

    Then we prove that the first case is impossible: if "T_j acquires R's lock < T_i's version check on R", then we have T_j acquires R's lock < T_i's version check < T_j releases R's lock. For this order, T_i will fail to validate R, this is because T_i would abort itself upon encountering T_j's lock on R.

Therefore, we prove that T_i's version check on R < T_j acquires R's lock, which implies T_i's end-of-lock-stage < T_j's end-of-lock-stage.

□