Investigations in Optimal Transaction Scheduling

Darren Teh
Jianzhi Wang

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2025-131
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2025/EECS-2025-131.html

May 23, 2025

Investigations in Optimal Transaction Scheduling

by Darren Teh

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Ion Stoica

Professor Ion Stoica
Research Advisor

05/21/2025

(Date)

* * * * * * *

[illegible]

Professor Matei Zaharia
Second Reader

05/21/2025

(Date)

Investigations in Optimal Transaction Scheduling

by

Darren Teh

A thesis submitted in partial satisfaction of the
requirements for the degree of
Master of Science
in
Electrical Engineering and Computer Sciences
in the
Graduate Division
of the
University of California, Berkeley

Committee in charge:

Professor Ion Stoica
Associate Professor Matei Zaharia

Spring 2025

Investigations in Optimal Transaction Scheduling

Darren Teh
University of California, Berkeley

Jianzhi Wang
University of California, Berkeley

Abstract

Transaction scheduling plays a pivotal role in optimizing the performance of database systems, especially in high-contention environments. Prior approaches rely on heuristic-based strategies, such as Shortest Makespan First (SMF) [1] and hashing to queues followed by sequential execution [2]. In this work, we first formalize the transaction scheduling problem to identify its core challenge. Following this formalism, we explore two classes of methods (optimization and neural networks) as scheduling policies. We also explore a new kind of workload in which the transactions are temporally correlated. Comparing k-SMF variants to neural net based approaches, we find that our Q-learning reinforcement learning model outperforms k-SMF in makespan at the expense of 3x, 5x, and 19x higher theoretical latency. We observe and explain the tradeoffs with the different scheduling schemes.

1 Introduction

The problem we investigate is how to best schedule a set of database transactions for OLTP databases and workloads. Transaction scheduling for high transaction execution throughput is critical for high-performance database systems where said systems are serving high volumes of latency-sensitive real-time consumers.

A *transaction* is a sequence of operations executed as a logical unit of work that must adhere to the ACID (Atomic, Consistency, Isolation, Durability) properties. For the purpose of this problem, an operation is either a read or a write operation and acts on a particular resource x, denoted as $R(x)$ and $W(x)$ respectively. An example of a transaction is $\tau = [R(x), W(x), R(y), W(y)]$. When a transaction τ is scheduled, the operations in the list are executed sequentially.

Different transactions can be executed concurrently, but if two concurrently executed transactions try to read or write on the same resource there will be consistency issues where the different transactions may interpret or modify the state of the same resource differently. This poses a problem for transaction scheduling, because any two transactions must give the illusion that they are completing in isolation of each other, without noticing the effects of the other transaction. For example, if a transaction τ performs a write operation $W(x)$, then it should expect subsequent read operations $R(x)$ to return the same value. Two transactions τ and τ' conflict if the isolation property of ACID is violated. Concretely, one can verify this by constructing a precedence graph and verifying the absence of cycles.

To maintain consistency, a database management system uses concurrency control methods such as 2PL or MVCC. For locking based concurrency control, transactions acquire and then release locks on resources when reading or writing them to prevent other transactions from reading an inconsistent state of the resources. However, if two transactions are scheduled such that they concurrently execute on the same resource, the later transaction must wait until the locks on shared resource are released before reading and writing to it. Waiting for the resource to unlock not only slows throughput of the system, but can lead to issues like deadlock or aborted transactions.

The main goal of scheduling transactions is to maximize *throughput*: the number of transactions completed (i.e. committed successfully) in a given amount of time. This must be done while minimizing the number of aborted transactions. A practical way of combining these two objectives is to generate a workload of transactions, W, and measure the number of time steps it take to complete the workload, known as makespan. For simplification of the simulation, any transaction that is scheduled and tries to access a locked resource and cannot access/modify said resource in accordance to its sequence of operations is subsequently aborted. Any aborted transaction must be continually rescheduled until it is eventually committed successfully.

Creating an optimal schedule under standard concurrency constraints is computationally complex; optimal transaction scheduling is NP-Complete [3], and due to the real-time constraint schedulers must schedule without knowing the future. Furthermore, these schedulers must be scalable which constrains intelligent methods that utilize complex computation or other mechanisms like memory-stores to track in-use resources.

2 Problem Setup and Formalism

In this section, we motivate the core challenges of the transaction scheduling problem.

Consider any particular point in time, the scheduler is presented with a transaction pool T comprising of transactions that have not been scheduled yet. The scheduler is also aware of another pool of transactions, F, that are in flight i.e. already scheduled. but have not finished execution (been committed) yet. The simplest formulation of the scheduler is as a function $S : T \rightarrow \{0,1\}$ that maps a transaction $\tau \in T$ to a decision $S(\tau) \in \{0,1\}$ denoting whether the transaction τ will be scheduled (1) or not (0).

Let the maximum number of operations in a transaction be M. M is typically a small constant in OLTP workloads. For example, $M = 6$ in the `SmallBank` workload.

The fact that M is usually small means that one can reduce the transaction pool T into a smaller filtered transaction pool T' consisting of transactions that will not conflict with those in flight F. In practice, a resource table can be constructed $R : r \rightarrow N$ that stores the latest time step a resource will be available. Upon scheduling a transaction τ', one updates the resource entries in R which τ' used. For filtering, one just checks whether each resource usage in a candidate transaction τ will occur before the last use time in R. Both the update and the query takes $O(M)$ time, which is very feasible. This assumption meant that one can disregard F when trying to model the scheduling function S.

The upshot is that the difficulty of the transaction scheduling problem lies in how to schedule a subset of transactions at any particular point in time.

An extension of the simple scheduling model is to observe that transactions can be scheduled in advance. The new format of the scheduling function is $S : T \rightarrow [[T]]^*$ where $[[T]]^* := \{0,1,2,...,T\} \cup \{-1\}$. A transaction τ scheduled at $S(\tau) = i \neq -1$ denotes that its first operation will start i steps after the current time step. A transaction with $S(\tau) = -1$ is not scheduled and is deferred to a future schedule. This formulation places greater burden on the scheduler to improve throughput.

A scheduler can obtain two main sources of signals: (1) a transaction τ has aborted (due to a conflict with a set of other transactions); (2) a transaction τ has completed. Signal (1) offers the information that τ should not have been scheduled the way it was, and perhaps other transactions conflicting with it should not be scheduled the way they are. Signal (2) offers the positive reinforcement that it was okay for τ to be scheduled the way it was and that it is sub-optimal to schedule τ at later time steps. Intelligent schedulers that can learn from these signals should learn over time to ultimately improve throughput.

3 Related Work

3.1 Serial

The serial scheduling strategy forms one of the baseline scheduling schemes. It schedules only one transaction when there are no transactions “in-flight”, waiting for a transaction to complete before starting execution of the next transaction. As a consequence, it achieves 0% abort rate at the expense of the worst throughput.

3.2 Random

The random scheduling strategy forms another one of the baselines. The order in which transactions are selected to be scheduled is random.

In the Shortest Makespan First scheduling scheme, the schedules are constructed by first randomly selecting a small sample of k transactions (ie, k = 5) from the unscheduled transaction pool, then scheduling the optimal one of the k sampled transactions [1]. Optimality is determined as the transaction that increases the total schedule’s predicted time to execute, or makespan, the least. This process of repeatedly picking the next transaction to schedule from a random sample is shown to be consistently better than truly random schedules which don’t do any makespan calculations. This greedy algorithm, unlike prior methods that rely on full access set knowledge, can be paired with a classifier trained on a particular workload’s hotkeys and transaction types to predict the makespan of each of the k randomly selected transactions (known as R-SMF) [1].

3.3 Queue-Based

In a queue-based strategy, transactions are hashed into Q buckets, with transactions in each bucket scheduled sequentially. For example, locally-sensitive hashing (LSH) can be done on the `WHERE` clauses of SQL statements to obtain a binary vector within a shared vector space [2]. Subsequently, buckets can generally represent the centroid of clusters within this space, so the binary vector should be assigned to the bucket with the closest corresponding centroid in this vector space. Ideally, transactions that are a smaller distance apart in this space would have similar where clauses and thus be more likely to conflict. By putting similar transactions into the same cluster, the hope is to schedule transactions within a cluster sequentially as to avoid scheduling conflicts. This inter-queue serial schedule is paired with intra-queue parallelism by having these queues operate separately and with general disregard of one another (no communication between the queues to enforce there aren’t conflicts in the final result).

4 Scheduling Approaches

We also observe that if the order of transactions is fixed, then scheduling becomes significantly easier. If we can only see and schedule transactions one by one, they can only conflict with in-flight transactions in F, so scheduling would reduce to picking the earliest possible time to schedule). However, interleaving the transaction into the existing schedule can be difficult if $|F|$ is large; this is partially alleviated by the fact that the length of transactions in OLTP workloads, M, tends to be small. However, by maintaining a resource mapper to track in-use resources, we only select transactions from the filtered set T' that cannot conflict with transactions in F. Subsequently, picking the earliest possible time to schedule a transaction becomes trivial, as the earliest possible time would be stored in the resource mapping.

For a scheduler that could make more complex scheduling decisions that consider both ordering and interleaving, we introduce the idea of a *kernel* that simultaneously makes scheduling decisions for a batch of transactions $B \subseteq T$ instead of sequentially making scheduling decisions for each individual transaction. Then, machine learning methods such as a neural network and reinforcement learning can be applied to these kernels to produce an optimal schedule for each transaction $\tau \in B$. To constrain the size of B, we define $N = |B| < 50$. We then represent the input kernel as a matrix of size $N \times N \times (2T+1)$ known as a conflict matrix $C \in \{0,1\}^{N\times N\times(2T+1)}$. $C_{i,j,\Delta t} \to \{0,1\}$ represents whether transaction τ_i and τ_j conflict if τ_j is scheduled $\Delta t \subset (-T,...,0,...,T)$ time steps ahead of τ_i. Here, $C_{i,j,\Delta t} = 0$ means the transactions will not conflict at that timestep difference and $C_{i,j,\Delta t} = 1$ means they will conflict.

From this, we postulate that conflict matrix C is a sufficient statistic for some kernel-based scheduler S_θ. Given this statistic, intelligent schedulers should be able to schedule in a way that both avoids conflicts and maximizes parallelism by either deciding time steps / interleaving (linear neural net) or ordering (reinforcement learning).

4.1 Optimization Problem

With the assumptions that we operate in time steps and that each operation takes 1 timestep, an optimal schedule can be defined as an integer optimization problem. Given the conflict matrix $C_{i,j,\Delta t}$, define the optimization variables as indicators: $x_{i,t} = 1\{\tau_i \text{ scheduled at } t\}$. The objective is to maximize the number of transactions scheduled subject to conflict constraints. The optimization problem is thus: $\max_{x_{i,t}\in\{0,1\}} \sum_{i\in T}\sum_{t=0}^{T} x_{i,t}$ subject to $\sum_{t=0}^{T} x_{i,t} \le 1, \forall i$ and $x_{i,t_1} + x_{j,t_2} \le 1, \forall C_{i,j,t_2-t_1} = 1$.

This approach gives guaranteed optimality. However, implementing a real-time scheduler that constantly solves this integer optimization problem is too computationally intensive for a real-time scheduling scheme. Therefore, taking inspiration from LSH [2], we try to do queue-based scheduling where transactions are assigned to queues where inter-queue transactions are likely to conflict. This allows integer optimization to shine, where the kernel can try to interleave the transactions even when the transactions conflict. Ultimately, this only partially alleviates the scheduling latency issue, but still does not make this viable. Thus, integer optimization serves as a benchmark that attains optimality only on a local 'queue' scale. The kernel is referenced as $Opt_{int-q=q,n=n}$ for q queues and n-sized kernel.

4.2 Linear Neural Networks

Given the conflict matrix C as statistic for the scheduling model S_θ, we denote a linear neural network with the following scheme.

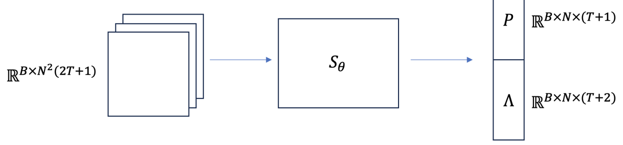
$\mathbb{R}^{B\times N^2(2T+1)}$
S_θ
P
$\mathbb{R}^{B\times N\times(T+1)}$
Λ
$\mathbb{R}^{B\times N\times(T+2)}$

Figure 1: A schematic of the neural network for the scheduling function S_θ.

To take into account the scheduling signals, the neural network was designed to output two tensors $P \in R^{N\times(T+1)}$ and $\Lambda \in R^{N\times(T+2)}$ 1. The i-th row of the scheduling tensor Λ is a probability distribution which denotes how transaction τ_i will be scheduled. The (i,j)-th element of the probability tensor P denotes the probability that transaction i will commit successfully without conflicts given that it is scheduled at the j-th time step after the current time step, conditional on all other information.

$$P_{i,j} \approx P[\tau_i \text{ commits}|C] \tag{1}$$

Assuming an optimal scheduling policy Λ^*, the probability of witnessing a set of signals $\{t_i, y_i\}_{i=1}^{L}$ indicating that transaction i committed ($y_i = 1$) or aborted ($y_i = 0$) if it is scheduled at time step t_i from now is:

$$\begin{aligned}
&P[\{t_i,y_i\}_{i=1}^{L}] \\
=&\Pi_{i=1}^{N}\Lambda_i^*(t_i)P_{i,t_i}^{y_i}(1-P_{i,t_i})^{1-y_i} \\
&\log P[\{t_i,y_i\}_{i=1}^{L}] \\
=&\Sigma_{i=1}^{N}\log\Lambda_i^*(t_i)+y_i\log P_{i,t_i} \\
&+(1-y_i)\log(1-P_{i,t_i})
\end{aligned}$$

This justifies the use of the logistic loss for signals relating to P, assuming the optimal scheduling policy Λ^* is known.

The loss for the scheduling policy Λ is more empirical and uses KL divergence. Consider a transaction τ_i with initial scheduling policy Λ_i that is scheduled at time t. If τ_i commits successfully, it gives the signal that it is suboptimal to schedule this transaction at a later time step. This prompts a target distribution $\tilde{\Lambda}_i$ where $\tilde{\Lambda}_{i,t'} = c\Lambda_{i,t'} \forall t' \leq t$ and $\tilde{\Lambda}_{i,t'} = 0 \forall t' \geq t$. Here, c is a normalization constant. On the other hand, if τ_i aborts, then it is not possible to schedule the transaction at the current time step. This prompts a target distribution $\tilde{\Lambda}_i$ where $\tilde{\Lambda}_{i,t'} = c\Lambda_{i,t'} \forall t' \neq t$ and $\tilde{\Lambda}_{i,t} = 0$. Again, c is a normalization constant.

The total loss is therefore:

$$\begin{aligned}\mathcal{L}(\theta) = -\Sigma_{i=1}^{N}(y_i \log P_{i,t_i}^{\theta} \\ + (1-y_i)\log(1-P_{i,t_i}^{\theta})) \\ + \Sigma_i \mathrm{KL}(\Lambda_i^{\theta} || \tilde{t}_i)\end{aligned}$$

4.3 Reinforcement Learning

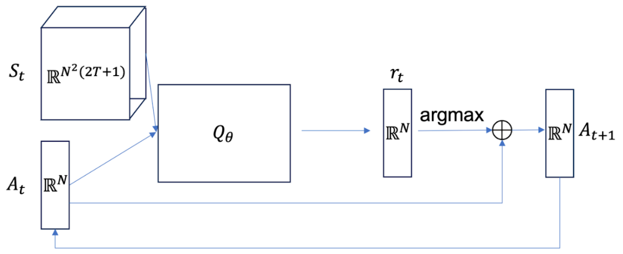

Figure 2: A schematic of the reinforcement learning Q-net for the scheduling function S_θ.

Similarly, we try a reinforcement learning (Q-learning) approach to transaction scheduling (*QKernel*) partially inspired by Marcus [4]. Each action is the transaction to be scheduled next, so unlike the neural network this approach does a pass through the Q-net for every transaction to schedule. The action (next transaction to schedule) $\alpha_i; 1 \leq i \leq N$ is determined by the q-net Q_θ which takes the state tensor (conflict matrix) S_t and a bitmask A_t that masks the previously scheduled transactions and prevents them from being scheduled again. The timestep is then determined by some scheduling scheme such as k-SMF.

By letting the reinforcement learning model handle the ordering of the transactions, the model should make more informed decisions on how to schedule transactions to maximize throughput. Ideally, it would schedule all non-conflicting transactions immediately and serially schedule conflicting transactions. This behavior is promoted by first immediately scheduling any transaction τ_i that has no conflicts, so $\Sigma_{j=1}^{N}\Sigma_{\Delta t=-T}^{T} C_{i,j,\Delta t} = 0$, reducing the load on the RL scheduler.

There is also another action α_{stop} to stop scheduling and put the unscheduled transactions back into the transaction pool. This action might be desired if the remaining unscheduled transactions conflict with already scheduled transactions and there's little to no room for interleaving.

Finally, the reward is expressed as $R_t = Q_\theta(S_t, A_t)$. The reward function used was tuned to promote scheduling transactions that interleave while still potentially having conflicts. For the stop action α_{stop} , the scheduler is rewarded for unscheduled transactions that would have aborted if scheduled within the time limit T and punished for those that would not have aborted.

5 Temporally Correlated Workloads

We postulate that if we fix the order of transactions, then scheduling becomes significantly easier. However, this would lose out on the potential of intelligently selecting the order of some batch of **M** transactions rather than relying on FIFO or randomness for ordering.

Notably, k-SMF which selects one transaction at a time may face adversarial examples where a reordering the selected transactions would reduce the makespan and thus increase throughput. However, real-world workloads have diverse conflict patterns (e.g., more transaction types, many hotkeys, etc.), so k-SMF is unlikely to repeatedly make poor choices in the long run [2]. For instance, on the TPC-C workload it is highly unlikely that k-SMF encounters only transactions with high conflict costs among its random samples at each iteration. But in the event that the incoming transactions in a short period of time all conflict with each other, k-SMF may perform poorly because it can't rely on randomness as much to avoid high conflict costs. This motivates the need for tuneable contention workloads.

To simulate high contention workloads, we developed correlated versions of benchmarking workloads such as Smallbank and TPC-C. These workloads may transition from a normal state $R|normal$ to a correlated state $R|corr$ based on workload parameters λ_1 and λ_2. The amount of time steps the workload is in the normal and correlated state respectively are $t_{normal} \sim Expo(\lambda_1)$ and $t_{corr} \sim Expo(\lambda_2)$. Within a correlated state / "burst", transactions are much more likely to interact with the same sticky resource (with probability p) and can be thought of as drawing from an alternate distribution R_{sticky}. More formally, $R|corr = R$ with probability $1-p$ and $R|corr = R_{sticky}$ with probability p.

6 Simulation

In the simulation infrastructure, we assume that operations take the same amount of time regardless of type (read or write) and resource.

The `Scheduler` class is in the exact same form as the formalism. It takes in T and F and returns $\{0,1\}^{|T|}$ denoting whether a transaction is scheduled or not. The `Simulator` class takes in a scheduler and runs a workload through it. The `Workload` class is an abstract class, which `SmallBank` and `TPC-C` implements. The workload generation follows the respective benchmarks' guidelines [5]. Additionally, resources were generated for the transaction operations based on a `zipf` distribution for more custom hotkey patterns.

The research infrastructure and schedulers are implemented and available on GitHub. [1]

In our experimentation, we implemented variants of k-SMF [2]: $kSMF_{twisted}$ (faithful replication), $kSMF_{normal}$ (without interleaving), $kSMF_{2-phase}$ (k-SMF first executed on correlated resources, then normal).

7 Results

After some experimentation, a bug in the conflict matrix adding more 1s (conflicts) than necessary led us to find that our original $kSMF_{normal}$ implementation was outperformed by $kSMF_{rapid}$ which simply schedules all k transactions as opposed to just 1. This led us to finding an inefficiency where $kSMF_{normal}$ was scheduling only on the latest time a resource is available and not considering scheduling before the resource is used. The corrected $kSMF_{twisted}$ significantly outperforms other scheduling approaches.

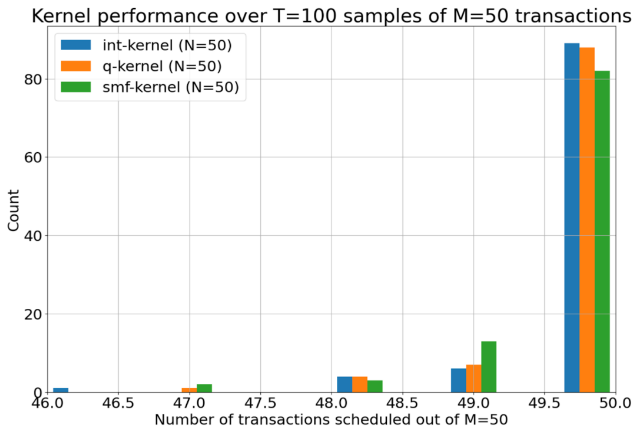

Figure 3: *Number of commits for each kernel's batch of B,* $|B| = N = 50$*,* R_{normal}

In Figure 3, the batch B of $M = 50$ transactions is scheduled with each kernel 100 times. It displays that the integer optimization and Q-learning kernel perform similarly to $kSMF$. The increase in performance is marginal because the conflict matrix C for B is sparse; as observed in [1], it's uncommon to find transactions that conflict from a small random sample. Furthermore in Figure 4, the unsupervised linear model with heuristic based loss function doesn't significantly learn good scheduling schemes. The neural network model converges on naive solutions and prioritizes scheduling transactions at timestep $ts_{step} + 0$ regardless of the conflict; even with the high loss of just the commit probability tensor P the limited loss heuristic and sparse conflict matrix makes it difficult

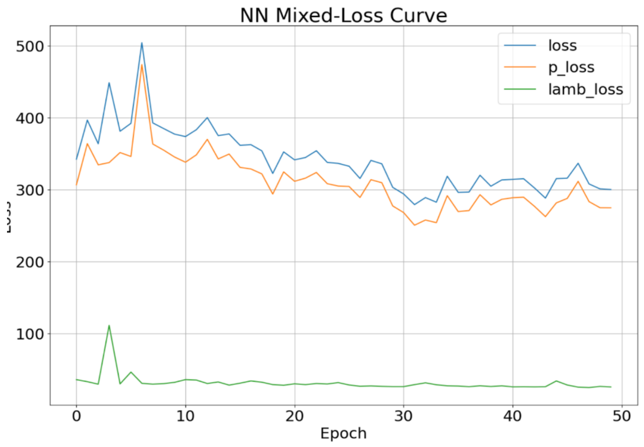

Figure 4: *Training loss for linear neural net model*

to prioritize avoiding aborts. Alternatively, different model architectures that approach scheduling with an interleaving-forward approach could be explored in the future, but because the scheduling latency of the neural net was already high (only coming second to last to integer optimization), we felt that the neural net approach was ultimately not worth pursuing.

In Figure 5, $kSMF_{twisted}$ outperforms $kSMF_{normal}$ (k=50-smf) and $kSMF_{rapid}$ by a bit. The $kSMF$ and $Opt_{int-q=10,n=50}$ both perform well at the beginning exemplifying its ability to schedule non-conflicting transactions well. Throughput slows down near the tail-end as all non-conflicting transactions are scheduled and the schedulers wait on the serially scheduled conflicting transactions. The queue-based schedulers are simple schedulers that assign a transaction to the queue that has the latest contention among the transactions in that queue already. Notably, the number of aborts for $kSMF_{twisted}$ and $kSMF_{normal}$, as well as the queue-based schedulers, remain 0 because this is their intentional design (to schedule the next transaction at the next available time step that won't conflict). $kSMF_{rapid}$ experience significantly high number of aborts despite its apparent initial high throughput because it blindly schedules all k transactions at the current timestep instead of just 1. The impute (I) scheduler versions only experience aborts initially because they switch to the $kSMF_{twisted}$ scheduling policy at $ts = 40$ (denoted by the vertical line in the figure). This impute is done under the belief that $kSMF$ does well to schedule under high contention and schedule conflicting transactions serially, but doesn't have the highest initial throughput. The queue-based schedulers (including $Opt_{int-q=10,M=50}$) all perform poorly because they are only optimal locally and not globally, showing the lack of viability for transactions that operate on row level (unlike SQL queries done in LSH) [2]. Transactions can often conflict with more than one bucket, so naively scheduling each bucket without consideration of the others leads to compounding conflicts as aborted transactions are rescheduled; this explains the stagna-

[1]https://github.com/jianzhi-1/db-workload-sim

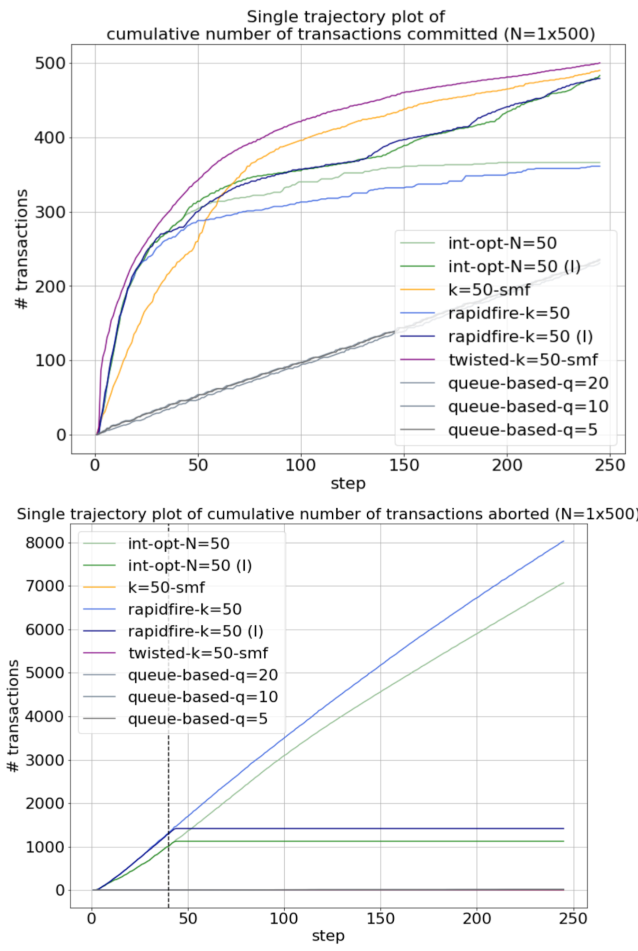

Figure 5: *(Above/below) Number of transaction commits / aborts, (I) signifies a switch to the $kSMF_{twisted}$ scheduler (at $ts = 40$). $N = 1 \times 500$ indicates the workload is 1 batch of 500 transactions.*

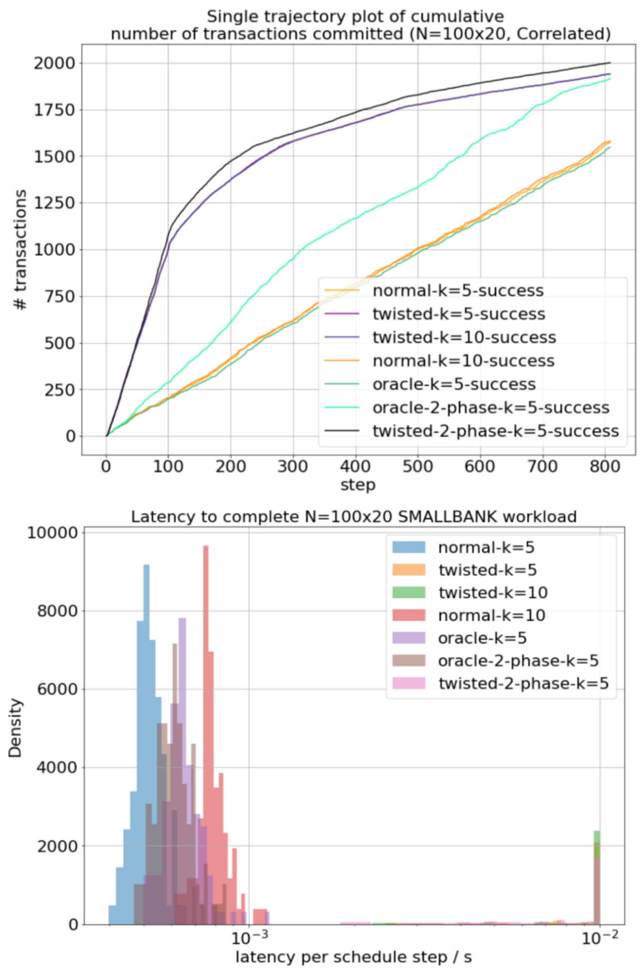

Figure 6: *(Above) Number of commits on* $R|corr$. $N = 100 \times 20$ *means workload that adds 20 transactions at each timestep for 100 timesteps. (Below) Latency to complete workloads*

tion of $Opt_{int-q=10,M=50}$.

For both uncorrelated and correlated workloads displayed in Figure 6, $kSMF_{twisted}$ dominates other scheduling methods at the expense of high scheduling latency (by a factor of 10 for correlated). Notably, $kSMF_{normal}$ performs significantly worse than $kSMF_{twisted}$, signifying that interleaving is much more important for higher contention workloads R_{corr}.

$kSMF_{2-phase}$ first schedules all transactions with correlated resources, then the rest of the transactions. $kSMF_{2-phase,oracle}$ additionally makes a suggestion to schedule some transaction based on makespan calculations for only non-correlated transactions, then the scheduler probabilistically chooses from this suggestion over picking the transaction for shortest makespan from all transactions. From Figure 6, it shows that $kSMF_{2-phase}$ schedulers outperforms their non 2-phase counterparts and that having prior knowledge on highly contested resources (hotkeys) can lead to minor scheduling improvements.

In Figure 7 and Figure 8, *kSMF* schedules are compared to different *QKernels* under `SmallBank` and `TPCC` workloads. The `n` parameter controls the size of the kernel conflict matrix, and `filterT` filters the transaction pool $T \to T'$ based on whether the resources of are being accessed by the in-flight transactions **F**.

After the *QKernel* decides the transaction scheduling order, the scheduler decides each transaction's scheduled timestep using either SMF (best possible timestep, $QKernel_{SMF}$) or using the conflict matrix (optimal only within B, $QKernel_{memory}$). When scheduling S, $QKernel_{SMF}$ will simply find the earliest available time step to schedule regardless of T, which could be much larger than T for a transaction that uses a hot resource. On the other hand, $QKernel_{memory}$ either delays transactions for later scheduling or tries to schedule them within the next T time steps (from the current time step). Transactions that are delayed don't count as aborts, but transactions that are chosen and cannot be

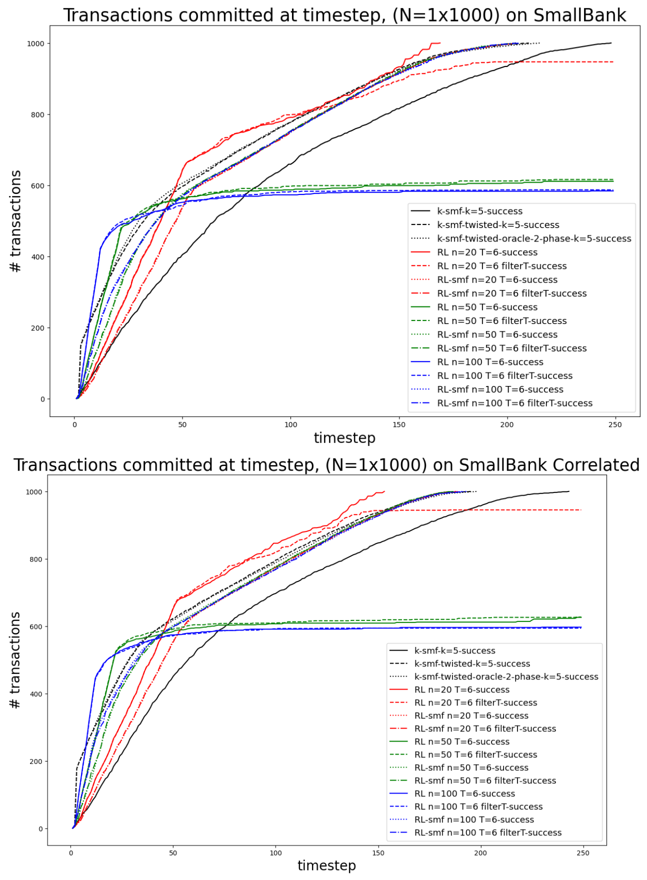

Figure 7: *Number of successful commits on SmallBank workloads*

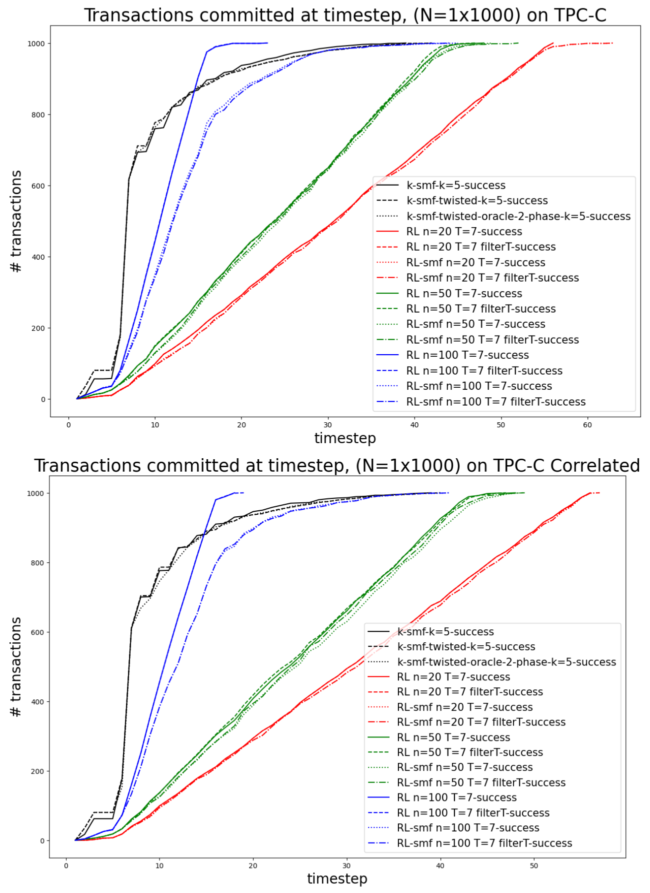

Figure 8: *Number of successful commits on TPC-C workload*

scheduled within the next T time steps are aborted. Notably, the $QKernel_{memory, filterT}$ and $QKernel_{SMF}$ fall behind $QKernel_{memory}$, showing that these scheduler variations that employ $QKernel$ don't decrease makespan much.

From observing training reward and scheduling decisions of the kernels, the large kernels $QKernel_{N=50}$ and $QKernel_{N=100}$ struggle with contentious conflict matrices. The reward function for scheduling a transaction is based on whether it can schedule the transaction within T timesteps of the current timestep, as well as recognizing that it should defer scheduling due to too much conflict. Lots of conflicting transactions with a large kernel make it exponentially difficult to generate good schedules. However, $QKernel_{memory}$ tries to schedule the batch of transactions within the next T timesteps, so for conflicting transactions the kernel must decide whether they can interleave and be scheduled within the next T time steps or if some should be delayed for some future B; incorrectly deciding the former case leads to aborts.

This decision is exemplified in Figure 7 where $QKernels_{SMF}$ perform better than $QKernels_{memory}$ on the `SmallBank` workloads where conflicts are much more common. In line with observations made during training, $QKernels_{memory}$ struggles with high contention and even stagnates near the end of the workloads; transactions that are delayed and thus likely operate on hot keys are put at the back of the transaction pool, so near the end of the workloads conflicts are higher. As a result, for SmallBank, $QKernel_{memory}$ of N=20, 50, and 100 ended with ~ 2500, ~ 10000, and ~ 20000 aborts at 250 time steps. The smaller $QKernel_{N=20}$ which doesn't suffer from as much contention (because of a smaller B) was able to schedule with a somewhat low makespan for `SmallBank`, but still suffers from a high number of aborts.

Smaller kernels don't benefit as much from the potential high throughput of $QKernel_{N=50}$ and $QKernel_{N=100}$ observed in figures 5b and 5d. The `TPC-C` workloads have a lot less contention compared to our implementation of `SmallBank` (typically varying from 0-5 conflicts per B, compared to >20 for `SmallBank`), so generally the large kernels didn't have many conflicts and could optimally order easily while simultaneously benefiting from high throughput that comes from the large kernel size. As a result, $QKernel_{N=100}$ even outperforms $kSMF_{twisted}$ in both `TPC-C` workloads.

To allow contentious transactions finish and thus avoids aborts at the potential expense of slower throughput, we imple-

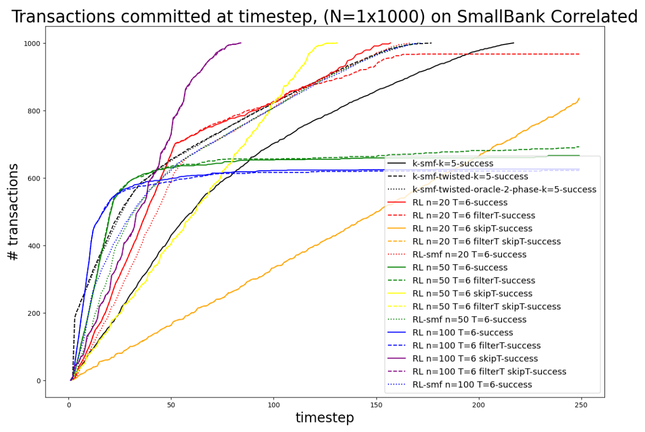

Figure 9: *Number of commits for N=(1x1000) on $SmallBank_{correlated}$*

ment a `skipT` parameter makes the scheduler make a scheduling decision every T time steps instead of every timestep.

Interestingly in Figure 9, with the `skipT` parameter active within high contention workloads in-flight transactions are allowed more time to complete; the `skipT` parameter lets contentious transactions finish and thus avoids aborts, as we find $QKernels_{skipT}$ have near-0 aborts. Although the initial throughput is slower because $QKernels_{skipT}$ make scheduling decisions much more infrequently, they appear to not suffer as much during the more contentious tail-ends of the workloads. As a result, there is not as much contention between transactions in **F** and B, allowing for $QKernel_{N=50,skipT}$, and $QKernel_{N=100,skipT}$ to surpass $kSMF_{twisted}$. However, $QKernels_{filterT,skipT}$ behaves similarly to $QKernels_{skipT}$, meaning that just the `filterT` parameter is not sufficient.

Makespan		Schedulers					
		$kSMF_{normal}$	$kSMF_{twisted}$	$kSMF_{oracle, 2-phase}$	$QKernel_{M=20}$	$QKernel_{M=50}$	$QKernel_{M=100}$
Workload	SB	249	211	**216**	**170**	DNF	DNF
	SB_{Corr}	244	196	199	**154**, 301	DNF,**132**	N/A, **85**
	TPC-C	40	43	43	57	**29**	**24**
	$TPC\text{-}C_{Corr}$	40	41	41	57	49	**19**

Figure 10: *Scheduler makespans (in ms) for $N = (1 \times 1000)$ on SmallBank and TPC-C.* ***Bold*** *indicates best out of the schedulers for the workload. Underlined indicates the worst. Comma separated entries indicate $QKernel_N$ and $QKernel_{N,skipT}$ respectively*

From Figure 10 we observe that in almost all scenarios there is some *QKernel* that outperforms all *kSMF* variant makespans. However, different sized kernels perform differently and thus it's necessary to tune the model and scheduler parameters (`skipT`, `filterT`, size of the kernel, reward function); this is on top of already individually training a model for each workload. Larger kernels can benefit from high throughput especially only low contention workloads. Large kernels struggle with workloads with high contention so much so that they can effectively stall out, but combined with the correct scheduler customization they can outperform *kSMF* .

8 Limitations and Future Work

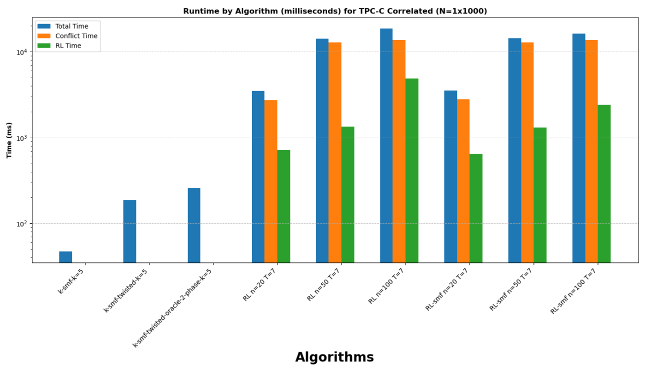

Figure 11: *Runtimes of algorithms for TPC-C correlated $N = (1 \times 1000)$ workload Conflict time is time to compute conflict matrix C, RL time is the Q_θ inference time*

In terms of trade-offs between *QKernel* and *kSMF*, *QKernels* total time to execute is a factor of 10x-100x longer than $kSMF_{twisted}$ and 100x-1000x longer than kSMF. Fortunately, most of the total scheduler decision time consists of computing the pairwise conflict matrix which is parallelizable; we do not implement this parallelization. When ignoring conflict matrix computation latency, *QKernels* outperform k-SMF in makespan through batched scheduling (higher throughput) at the expense of 3x, 5x, and 19x higher model inference latency.

Additionally, $QKernel_{memory}$ in high conflict workloads suffer from high aborts (although this could theoretically be reduced with more model training or better reward model heuristics). However, in addition to the displayed maximum throughput benefits, the `skipT` parameter can be used to avoid forcing the kernel to schedule high contention to both achieve high throughput and avoid aborts. In the future, this can be employed as a tunable parameter which controls how many time steps to wait before making another scheduling decision.

We show that the parameter `filterT` doesn't significantly improve makespan. However, the implementation and viability of a resource mapper necessary for $QKernels_{filterT}$ depends on a number of factors. The size of the in-flight transactions **F** and their correlated resources scales with the throughput of the scheduler. Consequently, the size of the resource mapping would scale linearly with the |**F**| which constantly changes as resources are locked and freed. Thus, |**F**| should be small enough to fit in the device the scheduler operates on or has access to (i.e. memory). A large map size could hinder latency, and a constrained |**F**| would hurt throughput. Fortunately, this linear scaling relationship can be alleviated if the mapper only maintains hotkeys (highly contentious resources), which can be done as future work. This idea of only operating on hotkeys rather than the entire key-set can be furthered by only considering hotkeys when calculating conflicts and scheduling, which could reduce the required computation and storage.

Initially, we tried using reinforcement learning and neural networks to sequentially schedule transactions with in-flight transactions as an input to the model and the output being the timestep to schedule. A reinforcement Q-learning proved difficult to produce a good schedule, but because the workload tested on (`SmallBank`) was simple we generated training input-output data pairs and saw resounding success. However, the way the input was compacted was heavily workload dependent. The input was formatted as a vector of size $n =$ the number of unique transaction types (which is dependent on workload). and without the benefit of batching like in the kernels the latency would be too high, so this approach was abandoned.

An alternate avenue to explore could be to use supervised learning for the linear neural network kernel. To obtain training data, we can adapt the Opt_{int} kernel to produce optimal schedules as training data. Then, some sort of distribution distance based loss such as KL divergence or a sum of timestep differences can be done. Furthermore, because we operate on the conflict matrix and are blind to metadata of the workload, a model that's seen high and low conflict B should be generalizable to new workloads.

The general viability of intelligent schedulers is constrained by computational complexity which subsequently affects scheduling latency. Although the statistic we chose (conflict matrix) to represent the transactions can be computed in parallel, there are definitely ways to simplify or modify this metric. For example, constructing a dependency graph and operating on a graph structure could be explored. Another approach could be avoiding computing conflicts for known non-conflicting transactions which would be less computationally intense.

Different model architectures that still employ a conflict matrix could be explored. Given that model inference time is a theoretical time latency bound, a model that can parallelize inference and remain relatively small would make for a lower bound for computation latency.

9 Conclusion

This report investigates and explores the viability of applying reinforcement learning and optimization-based techniques for transaction scheduling with OLTP workloads. We formalize the core scheduling challenge and evaluate different approaches including k-SMF variants, integer optimization, neural networks, and reinforcement learning with Q-learning. We also use simulations of workloads like SmallBank and TPC-C and develop correlated versions for higher conflicts.

We demonstrate that our reinforcement learning models *QKernels* can outperform both traditional *kSMF* and our own *kSMF* variants. However, these gains come with increased latency and computational costs due to the conflict matrix computation, which can be parallelized, and model inference time. The *QKernels* struggle under high contention workloads but with variation to the scheduling scheme (`skipT` and `filterT`) the kernels can still outperform *kSMF*.

Latency, abort rates under high conflict, and model generalization are further refinements that can be before real-world deployment. The problem of scheduling transactions is similar to the problem of scheduling instructions in VLIW compilers; in this vein, employing similar algorithms and heuristics done in similar contexts can inspire practical solutions.

References

[1] Audrey Cheng, Aaron Kabcenell, Jason Chan, Xiao Shi, Peter Bailis, Natacha Crooks, and Ion Stoica. Towards optimal transaction scheduling. *Proc. VLDB Endow.*, 17(11):2694–2707, July 2024.

[2] Yangjun Sheng, Anthony Tomasic, Tieying Zhang, and Andrew Pavlo. Scheduling oltp transactions via learned abort prediction. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, aiDM '19, New York, NY, USA, 2019. Association for Computing Machinery.

[3] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, October 1979.

[4] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: a learned query optimizer. *Proceedings of the VLDB Endowment*, 12(11):1705–1718, July 2019.

[5] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, 2013.