Towards Optimal Transaction Scheduling

Audrey Cheng
UC Berkeley
accheng@berkeley.edu

Aaron Kabcenell
Meta
akabcenell@meta.com

Jason Chan
UC Berkeley
j-chan@berkeley.edu

Xiao Shi
Unaffiliated
xiao.shi@aya.yale.edu

Peter Bailis
Google
bailis@google.com

Natacha Crooks
UC Berkeley
ncrooks@berkeley.edu

Ion Stoica
UC Berkeley
istoica@berkeley.edu

ABSTRACT

Maximizing transaction throughput is key to high-performance database systems, which focus on minimizing data access conflicts to improve performance. However, finding efficient schedules that reduce conflicts remains an open problem. For efficiency, previous scheduling techniques consider only a small subset of possible schedules. In this work, we propose systematically exploring the entire schedule space, proactively identifying efficient schedules, and executing them precisely during execution to improve throughput. We introduce a greedy scheduling policy, SMF, that efficiently finds fast schedules and outperforms state-of-the-art search techniques. To realize the benefits of these schedules in practice, we develop a schedule-first concurrency control protocol, MVSchedO, that enforces fine-grained operation orders. We implement both in our system R-SMF, a modified version of RocksDB, to achieve up to a 3.9× increase in throughput and 3.2× reduction in tail latency on a range of benchmarks and real-world workloads.

PVLDB Reference Format:
Audrey Cheng, Aaron Kabcenell, Jason Chan, Xiao Shi, Peter Bailis, Natacha Crooks, Ion Stoica. Towards Optimal Transaction Scheduling. PVLDB, 17(11): 2694 - 2707, 2024.
doi:10.14778/3681954.3681956

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/audreyccheng/transaction-scheduling.

1 INTRODUCTION

Maximizing transaction throughput is a critical objective for database systems. Inherently, transaction processing is an exercise in mediating conflicts to shared data. As such, performance differs significantly depending on how systems *schedule* transactions (i.e., which transactions to give access to contended data first).

Despite the vast amount of work on transaction processing, finding fast schedules in practical systems remains an open problem. Searching for the *optimal* schedule is infeasible in practice—Papadimitriou [60] proved in the 1970s that optimal scheduling is NP-Complete. Accordingly, the majority of existing concurrency control protocols execute requests based on arrival—or first-in, first-out (FIFO)—order (e.g., two-phase locking [15], multi-version concurrency control [14]): these approaches deal with conflicting operations after they have arrived [81] or executed [54], missing opportunities to avoid conflicts by planning ahead. To increase throughput, recent work schedules transactions more intelligently by leveraging full information about access sets (e.g., deterministic databases) and/or predicting key accesses. However, for efficiency, these approaches consider only a small subset of the schedule space (e.g., partitioning the workload on hot keys [28, 61, 63, 95] or randomly deferring requests [19])). As a result, they make suboptimal scheduling decisions. Overall, we observe a large performance gap between the schedules produced by existing methods and the best ones that can be found within the schedule space (Section 2).

In this work, we show that transaction processing systems can dramatically improve throughput by systematically exploring the schedule space, proactively identifying fast schedules, and executing them at run time. However, achieving this approach in practice raises two key challenges: (i) how to efficiently find fast schedules with partial information and (ii) how to enforce schedules during execution without violating isolation guarantees. We address both in R-SMF, a new scheduling-first transaction processing system.

Finding fast schedules. To maximize throughput, we want schedules that execute as fast as possible (e.g., for offline scheduling, the schedule that minimizes *makespan*, or the total time to execute all transactions). In analyzing real-world workloads, we observe that execution time between schedules differs due to the *cost of conflicts* across transactions—a transaction can incur high conflict costs if its conflicting operations stall the execution of other operations, causing delays that increase overall execution time. While searching for the optimal (i.e., fastest) schedule is computationally infeasible [60], we propose a greedy algorithm, Shortest Makespan First (SMF), which reduces conflict costs to find fast schedules. SMF iteratively constructs schedules by appending the transaction that leads to the least incremental increase in execution time. Crucially, SMF makes decisions based on how much each transaction conflicts with all other requests in the schedule rather than considering only the characteristics of an individual transaction.

SMF obtains fast schedules without relying on a priori knowledge of full read-write access sets by leveraging two observations. First, the small fraction of hot keys present in most workloads has an outsized impact on execution time (Section 3.2). Second, in practice, we can often infer such hot key accesses with high accuracy using the metadata contained in many applications (e.g., transaction type and initial arguments) [19]. Thus, focusing on minimizing conflicts for these keys enables SMF to capture most scheduling wins.

Proceedings of the VLDB Endowment, Vol. 17, No. 11 ISSN 2150-8097.
doi:10.14778/3681954.3681956

SMF finds competitive schedules compared to state-of-the-art search techniques, including those from job-shop scheduling (JSS). We observe that transaction scheduling can be framed as an instance of the well-known JSS problem [11] and compare SMF to the best techniques from the enormous corpus of JSS literature [8, 29, 30, 44, 47, 53, 57, 84, 89]. We find that, while state-of-the-art JSS methods obtain fast schedules, they have prohibitively high overheads (e.g., 15 transactions can take up to 30 minutes to schedule [69]). In contrast, SMF achieves linear time complexity with respect to the number of in-flight transactions and obtains schedules with performance within 10% of the best-performing JSS techniques. Furthermore, we provide statistical bounds on SMF's performance with respect to the entire schedule space (Section 3.4).

Executing schedules. Once a low-conflict schedule is selected, SMF must precisely execute it to realize its benefits—without imposing undue overheads (e.g., as a result of dependency tracking and enforcement). To avoid these overheads, existing concurrency control protocols do not proactively control the sequence in which individual operations complete [14, 15]. Systems that do enforce schedules incur high costs—either by serializing all requests in a single thread [80] or requiring developers to manually construct custom rendezvous points to coordinate dependencies [36].

To provide fine-grained control of schedule execution, we develop a new concurrency control protocol, MVSchedO. We adapt multi-version timestamp ordering (MVTSO) [14], a well known protocol that enables maximum concurrency (e.g., in contrast to two-phase locking [15], which pessimistically holds locks until commit) for this fine-grained regime. Typically, MVTSO assigns a serial order to transactions, but operations execute in parallel without restriction. Concurrent conflicting transactions abort if their timestamps do not match the execution order of their operations. In MVSchedO, we constrain database execution based on partial operation orders between transactions. In particular, we exploit the insight that scheduling hot keys has the biggest impact on performance: R-SMF leverages expected hot key accesses and maintains a scheduling queue for each of these keys to ensure that conflicting operations later in the schedule wait for preceding ones to complete. As a result, MVSchedO enables R-SMF to extract the most benefits from fast schedules with low overheads.

Impact of scheduling. We demonstrate that our scheduling approach leads to significant performance improvements in practice by evaluating it on a range of standard OLTP benchmarks and real-world workloads. In R-SMF, we augment RocksDB [32] with the SMF scheduler and MVSchedO execution mechanism to show up to a 3.9× increase in throughput over the baseline system. Moreover, we demonstrate that SMF scheduling is also extensible via a "bolt-on" approach [13]: we integrate our scheduler on top of the locking and OCC implementations in RocksDB and achieve up to 3.3× improvements in throughput, illustrating that existing systems can easily realize the benefits of SMF. Our system also improves tail latency by up to 3.2× because it executes transactions following schedules that reduce contention and thus, the likelihood of aborts. We further demonstrate that SMF is feasible for production use—providing up to a 2.5× increase in throughput and 2.1× decrease in tail latency—on TAO, Meta's social graph data store [17]. Finally, our scheduler has minimal overheads and observes a less than 5% drop in throughput on low contention workloads (Section 5).

To summarize, we make the following contributions in this work:

- We introduce a novel scheduling policy, SMF, that efficiently finds fast schedules on a range of transactional workloads.
- We propose a schedule-first concurrency control protocol, MVSchedO, that maximizes the benefits of scheduling by enforcing fine-grained operation ordering.
- We develop and evaluate R-SMF, which integrates SMF and MVSchedO in RocksDB to improve throughput by up to 3.9× and reduce tail latency by up to 3.2×.

2 SCHEDULING FOR BETTER PERFORMANCE

While it is obvious that transaction scheduling impacts throughput, there is no formal framework, to the best of our knowledge, that quantifies this performance difference. Thus, we present a model to precisely account for the impact of the schedule on throughput. Specifically, we apply the makespan metric to capture the effects of conflicts on execution time. By measuring the degree to which schedules affect performance, we confirm that different execution orders can have vastly different throughput.

2.1 Schedule Makespan

In this work, we focus on the impact of logical execution constraints (i.e., conflicts) on performance since they are the bottleneck in many transactional workloads [24]; physical resource scheduling has been addressed in prior work [55, 56, 75]. For a given schedule, transaction execution is constrained by: (i) operation dependencies (i.e., partial orders) within a transaction, and (ii) inter-transaction dependencies determined by a concurrency control protocol enforcing a specified isolation level during run time [9, 14].

To quantify the impact of these constraints, we evaluate schedule *makespan*, or the total time required for all transactions to complete. We focus on this metric because for a finite batch of transactions, minimizing makespan is equivalent to maximizing throughput. Thus, the goal of transaction scheduling is to determine execution orders that lower makespan and increase throughput.

We now describe how we compute makespan. We focus on the offline setting with the following assumptions for simplicity, but we find online systems can have even larger performance differences than what the makespan suggests (Section 5). We assume that each operation takes the same amount of time, during which any number of reads or one write can occur on a given key and that there are no execution errors that lead to aborts. We also assume "best-case" execution to focus on the *minimum* possible makespan of each schedule. While makespan is a simplified metric that does not exactly correspond to real-time execution (e.g., does not account for individual system overheads), it captures the effects of execution constraints within schedules. Accordingly, it enables us to compare the impact of conflicts on overall execution time, and we leverage it to develop an effective search policy (Section 3).

2.2 The Impact of Scheduling

In this section, we quantify different schedules to show how execution order affects throughput. First, we present a simple example in Figure 1 with a workload consisting of four transactions: $T1$ and $T3$ each read x and write z while $T2$ and $T4$ read z and write x. If we execute in FIFO order (the upper schedule), we would get

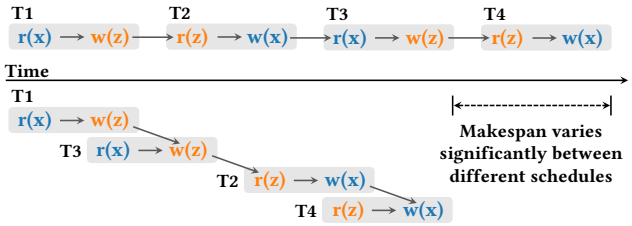

Figure 1: Two schedules of the same workload under MVTSO.

the worst-case makespan of eight time units since no concurrency is possible. Instead, if we order $T1$ and $T3$ together, we observe more concurrency as shown in the lower schedule, which has a makespan of six. Among just these four transactions, we observe a 25% decrease in makespan with the better schedule. Furthermore, this performance improvement is proportional to the number of transactions as well as the length of these transactions and becomes arbitrarily large as the workload increases in complexity. For instance, with 100 transactions of length 15 (assuming there are additional non-conflicting operations between the operations to x and z), there would be, between the two types of schedules shown in Figure 1, over a 11× (1,500 vs. 128) difference in makespan!

This difference persists on real-world benchmarks and workloads. Table 1 show that there can be over a 100% increase in makespan when using FIFO instead of optimized schedules on real-world workloads. Furthermore, scheduling can have an even bigger impact in online systems than what makespan suggests because aborts can be common and hamper system throughput (Section 5).

We observe that these schedules have varying makespan because they have different *cost of conflicts*, or how much conflicts delay the execution of other operations. In the previous example, ordering $T2$ before $T3$ results in higher conflict costs due to their contending operations to x. More generally, reducing the cost of conflicts lowers overall makespan. To maximize throughput, we need a scheduling policy that decreases these costs of conflicts to minimize makespan.

3 SEARCHING FOR FAST SCHEDULES

To find fast schedules, we introduce a new scheduling policy, Shortest Makespan First (SMF), that greedily minimizes the impact of conflicts and thus, total makespan. We provide intuition for how SMF is able to find schedules with low makespan and argue why adversarial scenarios for this policy are unlikely on real-world workloads. Note that in this section, we focus on the offline setting (where we assume a finite batch of transactions and known access sets) and extend our work to the online setting in Section 4.

3.1 SMF: An Effective Search Policy

We present our search policy, Shortest Makespan First (SMF), that greedily finds fast schedules. Since optimal scheduling for transactions is NP-Hard [60], we must rely on heuristics to develop a practical policy. While there are a plethora of greedy algorithms in scheduling literature [46, 67, 68, 72], they are not designed for transactional workloads, which have complex conflict patterns. Our key intuition in designing our policy is to minimize the cost of conflicts as we construct a schedule: SMF places transactions with high conflict costs far apart. Concretely, we evaluate the incremental *makespan* increase when a given transaction is added to the schedule because this metric accounts for the cost of all potential conflicts that an unscheduled transaction has with the current ordering.

Table 1: Percent increase in makespan between the best known schedule for each workload and FIFO (avg).

Epinions	SmallBank	TAOBench	TPC-C	YCSB
43.2%	8.0%	96.2%	101.6%	99.3%

Policy description. SMF starts the schedule with a random transaction.[1] At each iteration, SMF finds the transaction that increases makespan the least among k randomly sampled unscheduled requests and appends this transaction to the schedule. In the case of a tie, SMF randomly chooses one out of the best candidates. The policy repeats this process until all transactions have been scheduled and has a linear runtime of $O(n \times k)$ run time, where n is the number of transactions to be scheduled and k is a constant representing the sample size. We find in Section 5 that a small sample size (e.g., $k = 5$) is adequate for finding fast schedules since the chance of sampling a transaction with low conflict costs is high in real-world workloads, which have diverse contention patterns.

3.2 Why SMF Finds Fast Schedules

To provide intuition for why SMF is effective, we present a case study on TPC-C [27], a standard OLTP benchmark, for which our policy finds close-to-optimal schedules; Section 5.4 provides further empirical results. We analyze a workload of 500 `New-Order` and `Payment` transactions (assuming 10 Warehouses) since most conflicts for this benchmark occur on the Warehouse and District keys between these two transaction types. SMF (with a sample size of five and assuming all key accesses are known) finds a schedule 1.7× lower in makespan compared to the average obtained under FIFO. SMF is able to find fast schedules by minimizing conflict costs: it places `New-Order` and `Payment` transactions that do not conflict (e.g., access different Warehouse and/or District keys) together, creating "pockets" of concurrency with low costs of conflicts. As a result, it finds close-to-optimal schedules by allowing `New-Order` and `Payment` transactions that do not conflict to run in parallel.

Upon further analysis, SMF captures most of its scheduling wins by making the right ordering decisions for hot keys. Since transactional workloads tend to conflict on a small subset of keys (i.e., hot keys), these keys have an outsized impact on makespan. To demonstrate this, we run a version of SMF that is only aware of the Warehouse and District keys. The schedule produced by this version of the policy has almost equal performance (less than 6% difference in makespan) to the one found by the version of the policy that knows all key accesses apriori, showing that scheduling with only knowledge of hot keys is sufficient to find fast schedules. We find that the importance of hot keys for scheduling persists across different workloads (Section 5.4).

3.3 Adversarial Cases for SMF

While SMF finds low makespan schedules on most real-world workloads, we can construct adversarial scenarios for this algorithm. In general, greedy policies make "mistakes" by only considering local options since they do not look ahead when making decisions. In the case of SMF, it cannot change the position of a scheduled transaction. As a concrete example, we consider the following workload:

- Type A transactions ($T1, T2$): $\{r(x), w(x)\}$
- Type B transactions ($T3, T4$): $\{r(x), r(z), w(z)\}$

[1] We empirically find that the starting transaction does not significantly impact overall makespan once the workload is reasonably large (e.g., more than 20 transactions).

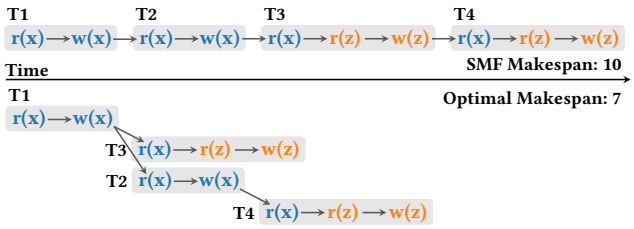

Figure 2: SMF can make suboptimal scheduling decisions under an adversarial workload with few conflict patterns.

Figure 2 shows the schedule obtained by SMF as well as the optimal schedule under MVTSO. Assuming SMF starts the schedule with $T1$ and has a sample size of three, SMF would choose to add $T2$, the other type A transaction, next because $T2$ results in the least increase in makespan (of two units, while either of the type B transactions would increase makespan by three). $T3$ and $T4$ are subsequently added to the schedule, yielding a total makespan of ten. However, the optimal schedule alternates type A and type B transactions (e.g., by executing $T3$ before $T2$). With this ordering, all operations of $T2$ can run in parallel with $T3$ because $T3$ only reads x while $T2$ reads and writes to x. SMF is not aware that interspersing type A's and B's leads to greater overall concurrency in the long run since it considers only the *next* transaction to add to the schedule (rather than multiple transactions at once).

Adversarial scenarios like the above example are unlikely in more realistic workloads. In the example, SMF has only two choices: either mix type A's and B's or schedule them separately. Since our policy makes the same (wrong) decision at each iteration, it finds a suboptimal schedule. However, real-world workloads have diverse conflict patterns (e.g., more transaction types, many hot keys, etc.), so SMF is unlikely to repeatedly make poor choices in the long run. For instance, on the TPC-C workload from Section 3.2, it is highly unlikely that SMF encounters only transactions with high conflict costs among its random samples at each iteration. We consider `New-Order` and `Payment` transactions accessing the same Warehouse or `New-Order` transactions accessing the same Warehouse and District to have high conflict costs since they cannot proceed in parallel. Among the iterations that SMF performs on this batch of requests, less than 5% observe all samples having high conflict costs with respect to the last 20 transactions in the schedule (we assume these transactions have not yet committed).

3.4 Statistical Performance Bounds

How well does SMF compare to all possible (serializable) schedules for a given workload? To answer this question, we must analyze the entire schedule space, which is inherently challenging given the exponential number of potential schedules. To do so, we uniformly sample the schedules of a given workload to construct a representative distribution of schedule performance. This enables us to not only understand the tradeoffs between different scheduling policies but also provide statistical bounds on performance with respect to the *entire* space (e.g., the 99^{th} percentile "fastest" schedule).

Constructing schedule space distributions. Since exhaustively evaluating all schedules of a space is prohibitively expensive (e.g., for 20 transactions with one read and one write operation each, there are $20! = 2.4 \times 10^{18}$ possible schedules), we consider a representative subset by sampling different schedules of the same workload. To provide statistical bounds, we need to sample across the space of valid (i.e., serializable) schedules *uniformly*, which requires some care. We focus on serializability since it is widely used and prevents data anomalies for real-world applications.

For uniform sampling, we represent serializable schedules as graphs and leverage existing graph sampling techniques. Each schedule maps to one corresponding acyclic serialization graph [14], which has nodes representing transactions and directed edges representing the order of conflicting operations. For a given workload, the serialization graph of every possible schedule has the same underlying (undirected) graph "structure"—same nodes and edges—and differs only in the *direction* of the edges. Thus, these serialization graphs are *acyclic orientations* of the undirected graph.

Consequently, sampling over all possible serializable schedules for a workload is equivalent to sampling over all of its acyclic orientations. We employ an existing algorithm, Interval-Reversal (IR) Random Walk [43], that uniformly samples over acyclic orientations of an undirected graph. Specifically, we form a Markov chain via the IR random walk until the chain converges to its stationary uniform distribution [43]. Thereafter, we can draw samples by continuing the random walk. Each step of the IR random walk costs $O(n+m)$, where n is the number of nodes (transactions) in the graph and m is the number of edges. The random walk takes on the order of $n \log n$ steps to converge [12]. Thus, the overall run time of this approach is $O(n \log n * (n+m) + k * (n+m))$, where k is the number of uniform samples we want to obtain.

This sampling technique is powerful: we can bound the performance of a given schedule with respect to the *entire* schedule space. Specifically, by using nonparametric, one-sided tolerance intervals [40], we can bound the proportion of schedules that have lower makespan than our best random sample with a specified confidence level. These statistical intervals do not make any assumptions about the underlying distribution (nonparametric), and the number of samples to achieve a desired bound and confidence level does not grow with the number of possible schedules. For instance, with 299 samples, we know the makespan of our best schedule from random sampling is better than 99% of all possible schedules with 95% confidence.[2] Note that these intervals do not bound the makespan difference of schedules, only the proportion of the distribution that falls beyond the intervals.

Contextualizing SMF's performance. We apply our uniform sampling approach to analyze SMF. While we do not always know the makespan of the *optimal* schedule, we can quantify the proportion of schedules that SMF outperforms. For instance, on our workloads in Section 5, we evaluate 100K random samples, the best of which is in the 99.999^{th} percentile in terms of makespan and better than 99.99% of all possible schedules with 99.99% confidence; the schedule found by SMF has even lower makespan.

Our uniform sampling approach can be applied to provide statistical guarantees on makespan in both offline and online systems. For systems in which all accesses are known apriori (e.g., deterministic DBs), this technique can be used as the default policy to ensure that the selected schedules have lower makespan than the majority of the *entire* schedule space. For online systems, this sampling

[2] For 99% of the distribution and 99% confidence, we need 459 samples; for 99.9% of the distribution and 99% confidence, we need 4,603 samples [40].

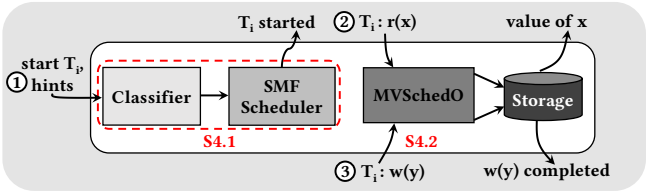

Figure 3: R-SMF architecture: we trace a transaction through its start (1) and several operations (2, 3).

technique can be used to quality check the schedules produced by SMF. We can periodically compare SMF's schedule with our best random sample from a given batch of transactions that has already executed. This process runs in the background separate from the main scheduler, so while it consumes CPU resources, it does not have a noticeable impact on run time performance (Section 5.2). With this technique, we can avoid worst-case behavior from SMF: if SMF's makespan deviates significantly to the right of the mean of the samples over a period of time, we can fallback to FIFO.

4 AN ONLINE SCHEDULING-FIRST DATABASE

Our analysis in the previous section highlights the potential impact transaction scheduling can have on performance. To realize these gains in practice, we need to adapt our scheduling policy to the online setting. In doing so, there are two main challenges we must address: (i) making scheduling decisions without full information and with low overheads and (ii) ensuring correct execution of schedules. We address both with R-SMF, a system that searches for and executes fast schedules to improve throughput. R-SMF consists of three main components (Figure 3): (i) a classifier that predicts hot key conflict patterns, (ii) a SMF scheduler, which determines how transactions are ordered, and (iii) a schedule-first concurrency control protocol, MVSchedO, which enforces fine-grained operation execution. In the rest of this section, we explain each in detail.

4.1 Online Scheduling

While we assume in the offline setting that all key accesses are known in advance, this information is not available for many workloads in online systems. To deal with this, we leverage our observation that scheduling hot key conflict patterns has an outsized impact on performance (Section 3.2). Rather than relying on all operations to make scheduling decisions, we focus on *predicting* hot key conflicts using a classifier and small number of application "hints." We then apply our predicted hot key access patterns to approximate makespans and optimize SMF for online scheduling.

4.1.1 Application Hints. We leverage two types of application hints in our system: transaction type and hot key accesses. Together, this information enables us to effectively predict hot key conflict patterns, which have the largest impact on schedule makespan. First, transaction type is readily available: most applications along with nearly all standard benchmarks [31] execute a pre-defined (and small) set of transactions, either hand-written or generated through ORM frameworks [91]. Second, we find that many applications provide information about hot keys upfront, as input to the transaction itself. For example, all benchmarks in OLTPBench [31] place hot key accesses at the beginning of transactions to reduce contention (e.g., avoid deadlocks). These keys are known upon transaction instantiation (e.g., the Warehouse and District keys for TPC-C and the user and item ids for Epinions). Some workloads, such as Meta's social

```
DepositChecking(cName, amt):
    START TRANSACTION(type = 0, keys = {cName});
    id = SELECT cId FROM ACCOUNTS WHERE name = cName;
    UPDATE CHECKING SET bal = bal + amt WHERE cId = id;
    COMMIT TRANSACTION;
```

Listing 1: SmallBank `DepositChecking` transaction passing transaction type and hot key information to R-SMF.

Classifier

```
/* Data structures */
class Op: { op_type: read/write; position_in_txn: int }
class Txn: { txn_type: int; ops: List[Op] }
class MetadataVec: List[int] /* txn_type at vec[0],
                                * hot keys as ints at vec[1:] */
/* Shared function */
txn_to_md_vec(txn: Txn) → MetadataVec

/* Training */
find_clusters(trace: List[Txn]) → List[<MetadataVec, /* cluster_label */ int>],
                                  /* num_clusters */ int
train(txns: List[<MetadataVec, /* cluster_label */ int>], /* num_clusters */ int)

/* Prediction */
predict(md_vec: MetadataVec) → /* cluster_ label */ int
get_hot_key_ops(cluster_label: int) → List[Op]
```

Figure 4: Our classifier provides a simple API.

graph transactions [23], provide the full read-write set at the start of each transaction. At scale, application awareness of hot keys is crucial in preventing one program from affecting the performance of others that share the same data [10]. We note that this information does not have to be provided by the application explicitly—it can be inferred automatically from other metadata (e.g., application stack traces, client endpoints, etc.) [63, 74, 78, 85, 91].

Annotating and passing along this information is straightforward: the only change required in the application code is to assign a unique value to each transaction type and send it to the database (e.g., modify the START statement, as shown in the SmallBank `DepositChecking` transaction in Listing 1). Consequently, minimal changes are needed on existing applications and systems.

4.1.2 Classifier. To make good scheduling decisions, we need information on hot key access patterns, which are used to calculate conflict costs. R-SMF uses a classifier (Figure 4) to predict these access patterns. We train our classifier on a trace of transactions for each workload, and this process involves three parts: (i) mapping transactions to metadata vectors, (ii) finding the optimal number of clusters, and (iii) determining a canonical set of hot key operations for each cluster. First, we take a given trace (which we assume includes transaction type and known hot keys at transaction instantiation) and map each transaction to a metadata vector that represents this information as integers. We set aside a portion of the transactions for validation. Second, we find the optimal number of clusters (i.e., k) for these metadata vectors. On the training dataset, we cluster the metadata vectors based on Euclidean distance, which is widely applied for clustering and classification [48]. Specifically, we pick the k that gives the lowest validation error on our validation set. Finally, we determine a canonical set of hot key operations for each cluster. We note that widely differing sets of hot key operations among transactions in a cluster are rare on most workloads since many hot keys have correlated accesses (e.g., each Warehouse and District key pair has its own cluster in the TPC-C workload). Our classifier chooses the most frequently occurring set of hot key operations among the transactions in a cluster as

the canonical access set, and we experimentally confirm this is sufficient for standard workloads (Section 5.3).

At run time, we generate a metadata vector based on the application hints. Our classifier then predicts a cluster label for this vector and gets the canonical set of hot key operations corresponding to this label. For instance, a `Payment` transaction in TPC-C sends along its type as well as its Warehouse and District keys. Our classifier uses this information to infer a label that corresponds to the predicted hot key operations (a read and a write to the Warehouse key followed by a read and a write to the District key).

We find that a KNN-classifier provides high accuracy on a range of workloads (Section 5.3). Furthermore, the classifier is simple (no model or parameters), has low overhead, and can easily be updated with new data. While most workloads are relatively stationary over time (e.g., most standard benchmarks), we retrain our classifier periodically (and redetermine the optimal number of clusters) with recent traces to account for changes in hot key accesses. Classifier accuracy is dependent on the quality of application hints, and we view predicting contention as an interesting avenue for future work.

4.1.3 Approximating Makespan. Finally, to adapt our SMF scheduler to an online setting, we apply the predicted hot key access patterns from our classifier to calculate makespans and employ additional optimizations to make this calculation efficient. With the predicted hot key operations of each transaction (including read-/write and position information), we compute how the makespan changes (via the process described in Section 2.1) assuming this transaction is added to the schedule. For example, if the last transaction in the schedule is a `Payment` transaction, SMF would find that adding another `Payment` transaction that writes to the same Warehouse leads to a higher total makespan than adding a `Payment` transaction that writes to a different Warehouse. To reduce run time overheads, we only consider in-flight transactions to be a part of the schedule, and we assume that transactions which do not access hot keys have no impact on makespan (they execute immediately). Finally, for each hot key, we consider only the conflicting operation of the latest transaction that accesses this key. We can ignore earlier transactions that conflict on the same keys because these conflicts were already accounted for when the latest transaction was added to the schedule. SMF uses limited memory by design since we do not need to store incremental makespan results past an iteration. As a result, our scheduler has minimal performance and storage overheads while finding fast schedules (Section 5.2).

4.2 Schedule-First Concurrency Control

Now that we know which hot key conflicts are likely to occur and how to schedule them with SMF, we turn to the problem of executing transactions correctly while maximizing the benefits of fast schedules. Since scheduling and concurrency control are complementary techniques, we can incorporate SMF directly without changing existing concurrency control protocols by starting transactions based on the determined schedule, which significantly improves performance (Section 5.1). While this is useful for existing systems, most concurrency control mechanisms limit what schedules are allowed and do not treat the transaction schedule as a first-class component (i.e., cannot precisely execute operations according to a predetermined order). To maximize the benefits of scheduling, we introduce a novel schedule-centric concurrency control protocol, MVSchedO, that extracts the most benefits from a fast schedule. MVSchedO augments MVTSO [14], a well-known and performant concurrency control protocol, by enforcing fine-grained control on operation execution with low overheads.

Algorithm 1: MVSchedO

```
1  Data structures
2  hot_keys: set of high conflict keys
3  pred_ops: set of predicted operations, per hot key
4
5  procedure START_TXN(H : application hints):
6      // Assign unique timestamp to each transaction
*7     ts ← SMF_SCHEDULER(H)
*8     pred_hot_key_ops ← PREDICT_HOT_KEY_OPS(H)
*9     for op ∈ pred_hot_key_ops do
*10        pred_ops[op.k].add(op)
11     return ts
12
*13 procedure SCHED_KEY(k : key, ts : timestamp):
*14    await min(pred_ops[k].get_all_ts()) ≥ ts
15
16 procedure READ_KEY(k : key, ts : timestamp):
17     // Delay op until all conflicting ops with lower
          timestamps on this key have executed
*18    if k ∈ hot_keys then
*19        SCHED_KEY(k, ts)
20     val ← MVTSO_READ(k, ts)
*21    pred_ops[k].remove_read(ts)
22     return val
23
24 procedure WRITE_KEY(k : key, v : value, ts : timestamp):
*25    if k ∈ hot_keys then
*26        SCHED_KEY(k, ts)
27     MVTSO_WRITE(k, ts, v)
*28    pred_ops[k].remove_write(ts)
29
*30 procedure FREE_HOT_KEY_DEPS(ts : timestamp):
*31    pred_ops.remove_ops(ts)
32
33 procedure COMMIT(ts : timestamp):
*34    FREE_HOT_KEY_DEPS(ts)
35     success ← MVTSO_COMMIT(ts)
36     return success
```

4.2.1 Making MVTSO Schedule-First. We design a schedule-first concurrency control protocol, MVSchedO, by adapting MVTSO, which we choose as a starting point for two main reasons. First, MVTSO enables high performance: it imposes the minimal execution constraints required to ensure serializability in contrast to pessimistic protocols like two-phase locking (2PL) [15], which artificially increase run time by holding locks until transaction commit. Second, MVTSO assigns a schedule to transactions and enforces serializability based on this schedule. Thus, we can leverage existing correctness mechanisms with minimal changes while replacing the schedule with one determined by SMF. We briefly describe MVTSO and its known flaws before presenting MVSchedO, which overcomes these issues by following transaction schedules precisely.

MVTSO. MVTSO assigns each transaction a unique timestamp corresponding to its serialization order. To ensure serializability, MVTSO records the read and write timestamps of transactions. Any transaction containing a write operation with a smaller timestamp than the highest read timestamp on a key is aborted so that no

read ever fails to observe a write from a transaction earlier in the serialization order. Transactions keep track of any write dependencies (uncommitted writes they observe) and commit once all such transactions commit or abort if any abort. The key benefit of MVTSO is that it makes uncommitted writes immediately visible to simultaneously executing transactions, enabling more concurrency.

While MVTSO offers significant benefits, it has two flaws that hamper performance: (1) it assigns timestamps in arrival order, missing opportunities to benefit from faster schedules, and (2) it does not delay transactions ordered later in the schedule from executing before earlier ones, potentially causing unnecessary aborts. For example, MVTSO does not prevent a transaction with a higher timestamp T_2 from reading a key x before a transaction with a lower timestamp T_1 attempts to write to this key. As a result, T_1 must abort. This abort can have cascading effects: if other transactions have write-read dependencies on T_1, they must also abort.

MVSchedO. Our protocol MVSchedO overcomes these issues by extending MVTSO in two main ways. Algorithm 1 shows our protocol, and our changes to MVTSO are marked with asterisks. First, MVSchedO assigns timestamps using SMF rather than FIFO (line 7). Second, we proactively enforce this order on hot keys by ensuring that operations with later timestamps do not execute until conflicting ones earlier in the schedule have completed. Specifically, we predict the set of hot keys operations for each transaction at its start (lines 8–10) and use this information to decide whether to delay an operation before its execution (lines 13–14, 18–19, 25–26). Since we only maintain this information per hot key, the overheads of our approach remain low (Section 5.2). If an expected hot key access never occurs (i.e., the prediction was wrong), any queued transactions will be able to proceed once the transactions they are waiting on commit or abort (line 34). Reads, writes, and commit validation (which checks the write dependencies of a transaction) otherwise execute identically to MVTSO (lines 20, 27, 35).

Correctness and optimality. MVSchedO ensures serializability because all executions it permits are also possible under MVTSO (albeit under a different arrival order). Since timestamp assignment in MVTSO is arbitrary, the schedule chosen by SMF does not affect serializability guarantees. For reads and writes to hot keys, MVSchedO physically delays operations that are free to execute in any order under MVTSO (subject to versioning constraints, which apply for both protocols). Otherwise, MVSchedO uses the same validation mechanisms as MVTSO, so it provides serializability.

Furthermore, MVSchedO executes a given schedule with the maximum allowable concurrency under serializability.[3] That is, in the absence of resource constraints and aborts, it is impossible for another serializable protocol to extract more concurrency from the execution of the schedule. This property holds with early write visibility [36], which is enabled by both MVTSO and MVSchedO.

5 EVALUATION

In this section, we evaluate R-SMF on a range of different workloads. Specifically, we aim to answer the following questions:

- What are the benefits of scheduling in a real-world system?
- What are the overheads of our approach?
- How does SMF compare to alternative search techniques?

[3] MVSchedO ensures optimality of schedule *execution*, not the optimality of the *schedule*.

5.1 Scheduling in Practice

Our first set of experiments focuses on evaluating R-SMF on RocksDB, Meta's transactional key-value store [32]. We compare against a state-of-the-art scheduling policy that probabilistically defers transactions [19] as well as several standard concurrency control protocols. We find that R-SMF increases throughput by up to 3.9× and also decreases tail latency by up to 3.2×.

Experimental setup. We implement R-SMF and various baselines in RocksDB (8.5) [34]. We run our database and clients on separate c5ad.16xlarge EC2 instances with 64 vCPUs, 128GB RAM, and local NVMe-based SSDs in the same region. Clients run in a closed-loop fashion with exponential backoff (we account for aborts and retries when measuring latency), and we report the average of three 60 second runs with 30 seconds of warm-up each. For each workload, we tune the number of client and worker threads to ensure system saturation. We follow the standard tuning guide [5] for RocksDB. We compare R-SMF to the following baselines:

1. **RocksDB Optimistic Concurrency Control (OCC).** RocksDB's Optimistic transactions [34] provide up to Snapshot Isolation (SI) using Optimistic Concurrency Control.
2. **RocksDB Locking (Lock).** RocksDB's Pessimistic transactions use a locking protocol [34] that holds only write locks and reads from snapshots to provide SI.
3. **RocksDB Multi-Version Timestamp Ordering (MVTSO).** We implement MVTSO [14] in RocksDB to provide serializability.
4. **Aria**. We implement Aria [54], which includes a reordering mechanism that reduces conflicts, in RocksDB.
5. **`TsDEFER` (Defer).** This protocol checks for conflicts on two keys from the predicted access set of each transaction. If there are in-flight requests already operating on these keys, Defer chooses to queue the transaction with a probability of 60% [19].
6. **`TsDEFER`-MVTSO (Defer-MVTSO).** We extend Defer to delay timestamp assignment for MVTSO when queuing a transaction.

R-SMF. We implement R-SMF in RocksDB, making several modifications to the transaction manager to support SMF and MVSchedO. First, we modify the transaction `START` function to take in application hints, which are passed into our classifier to predict hot key access patterns. Second, we allow SMF to queue transactions at their start until it schedules them. We use a sample size of five transactions at each iteration (we find more samples do not substantially improve performance). Once a transaction is scheduled, we add its predicted hot keys accesses to MVSchedO's scheduling queues. We also measure the performance of MVSchedO as a baseline: we queue transactions based on predicted hot key accesses without using SMF (transactions execute in FIFO order). To prevent starvation, we place barriers in the scheduling queue that ensure requests in front of the barrier will execute before those after it.

Bolt-on schedulers. Since scheduling is complementary to concurrency control, we can combine our scheduling policy, SMF, with existing concurrency control protocols, though the benefits will naturally be smaller than using MVSchedO. However, this enables existing systems to easily realize the benefits of scheduling. We add SMF as a layer above RocksDB's OCC and Lock protocols [34]. Our implementations, SMF-OCC and SMF-Lock, queue transactions at their start until they are scheduled. Once a transaction begins

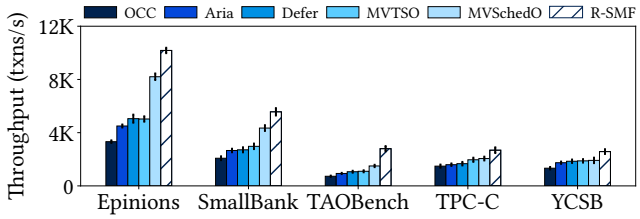

Figure 5: R-SMF performance on application benchmarks.

to execute, it runs under the original concurrency control protocol without fine-grained operation scheduling. Even with limited control over execution, these implementations are able to achieve significant performance improvements (Section 5.1.2).

TAO prototype. We also implement a SMF prototype on TAO, Meta's social graph data store [17]. Our prototype applies the bolt-on approach, similar to the RocksDB bolt-on schedulers, by queuing transactions at their start (TAO implements a variant of 2PL for concurrency control [24]). The prototype is implemented as an adapter layer in C++ that sends requests to a TAO (cache and database) deployment in a testing environment. Our experimental setup imitates that of production: each thread acts as an individual TAO client, mirroring how Meta's applications access this system [24].

Benchmarks. We evaluate the performance of our schedulers on a range of different standard benchmarks and real-world workloads. **Epinions** [31] consists of nine transaction types that represent behavior observed on a consumer reviews website. We run the benchmark with 2M users and 1M items (total data size of 50GB). **SmallBank** [79] contains six types of transactions that model a simple banking application. We run the benchmark with 1M accounts (total data size of 50GB). **TPC-C** [27], a standard OLTP benchmark, simulates the business logic of e-commerce suppliers with five types of transactions. We use a separate table as a secondary index on the Order table to locate a customer's latest order in the `Order-Status` transaction and on the Customer table to look up customers by last names (for the `Order-Status` and `Payment` transactions) [22]. We run the workload with 10 Warehouses (total data size of 2GB). **TAOBench** [23] is a social network benchmark based on Meta's production traces. We run Workload T, which captures the full transactional workload on TAO, Meta's social graph database. We run the benchmark with 10M objects (total data size of 100GB). **YCSB** is a microbenchmarking suite that generates read and write operations, which we place into groups of 16 [19]. We use Workload B (95% reads, 5% writes) with a Zipfian distribution and load 1M objects (total data size of 10GB).

5.1.1 R-SMF Results. R-SMF outperforms all baselines on the five application benchmarks since it executes transactions following schedules that minimize conflict costs (Figure 5).

Epinions. R-SMF improves throughput by 3.1× compared to OCC (Figure 5). This workload centers around user interactions and item reviews, containing five read-only transactions and four read-write transactions. Given the skewed access to popular users and items, scheduling prevents many aborts that arise under the baselines. Since most transactions are short, reducing wasted work caused by aborts has an outsized impact on throughput. This is further confirmed by the fact that there is only a 24% difference in throughput between MVSchedO and R-SMF. These results demonstrate that executing schedules precisely with MVSchedO can significantly improve throughput, even under FIFO (which can be an effective fallback strategy). We observe a 2.0× throughput improvement compared to MVTSO; this baseline has higher throughput compared to OCC since it exposes uncommitted writes. Aria also has higher throughput than OCC since its reordering mechanism reduces some aborts. [54] However, since this protocol is single-versioned, it produces more conservative schedules than those allowed by MVTSO and results in more aborts. Defer avoids some conflicts by delaying requests but not as many as R-SMF, which achieves 2.0× higher throughput. On the other hand, Defer-MVTSO (omitted from the graph due to space constraints) has nearly equal performance to MVTSO because delaying timestamp assignment does not address potential race conditions during execution (i.e., after the timestamp has been assigned). R-SMF improves tail latency by 29% and reduces latency variance by 14% compared to 23% and 11% for Defer, respectively (Table 2). Request times do not vary significantly in this workload since most transactions are short. With scheduling, we also observe lower abort rates: OCC aborts 6.0% of transactions while R-SMF has a 0.2% abort rate.

Table 2: Defer (D) and R-SMF (R) latency compared to OCC.

Workload	Var. (D)	P99 Latency (D)	Var. (R)	P99 Latency (R)
Epinions	11%	23%	14%	29%
SmallBank	13%	27%	19%	35%
TAOBench	43%	107%	65%	323%
TPC-C	10%	28%	20%	47%
YCSB	33%	81%	45%	101%

SmallBank. We observe a 2.7× increase throughput comparing R-SMF to OCC (Figure 5). This workload consists mainly of short read-write transactions, and R-SMF is able to prevent conflicts between transactions accessing the same user accounts. Reducing aborts on this workload has a large impact since most transactions are short (aborts and restarts take comparatively longer). R-SMF has a smaller improvement in throughput of 1.9× compared to MVTSO since this baseline enables more concurrency. However, R-SMF still achieves better performance because it executes based on fast schedules and reduces aborts that are caused by race conditions during MVTSO's timestamp assignment. MVSchedO achieves 28% lower throughput compared to R-SMF. Against Defer, R-SMF has 2.0× higher throughput, since the former delays transactions with a fixed probability (missing some conflicts). Defer-MVTSO has similar performance to MVTSO since both observe race conditions after timestamp assignment. For latency, R-SMF improves tail latency by 35% and reduces latency variance by 19% compared to 27% and 13% by Defer (Table 2), respectively. Both policies are able to avoid repeated aborts though SMF attains better schedules by intelligently scheduling requests rather than randomly deferring them. R-SMF has a 0.2% abort rate compared to 5.7% under OCC.

TAOBench. We observe a 3.9× throughput increase with R-SMF compared to OCC and a 2.6× increase compared to MVTSO. This workload is read-heavy and skewed, typical of most social networks. There are many short read transactions, some shorter read-write transactions (under 10 operations), and a small portion of longer read-write transactions (up to 60 operations). Since key accesses are drawn from probability distributions in this workload, hot keys are not requested in a fixed order, as in many of the other benchmarks. Consequently, naively ordering requests in FIFO order results in slow schedules and many aborts. This is further evidenced by the fact that MVSchedO has 87% lower throughput than R-SMF. Our system is able to avoid placing long transactions accessing many

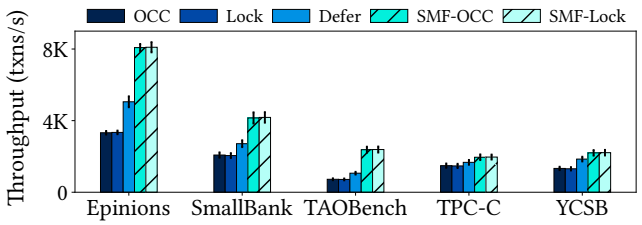

Figure 6: Bolt-on performance on application benchmarks.

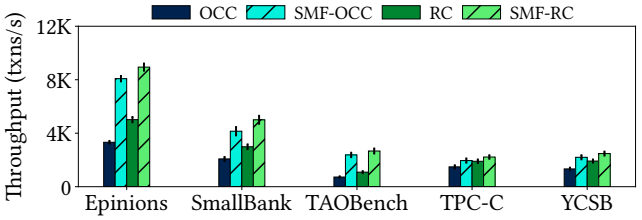

Figure 7: Performance under varying isolation levels.

Table 3: Performance difference from TAO baseline.

Workload	Throughput	P99 Latency
TAOBench	252%	-208%

hot keys together with conflicting requests. By reducing aborts (from 12.0% under OCC to 0.5% under R-SMF), R-SMF is also able to reduce tail latency by 323% and latency variance by 65% (Table 2). Defer has moderate success in avoiding aborts (R-SMF has 2.6× higher throughput); since the access sets of these transactions are larger, Defer has lower probability of detecting potential conflicts.

TPC-C. R-SMF improves throughput on TPC-C by 1.8× compared to OCC and 1.3× compared to MVTSO. The latter has noticeably higher performance than the other baselines because this protocol allows writes to be pipelined (`New-Order` and `Payment` transactions can access Warehouse and District keys prior to the commit of preceding requests). However, since MVTSO schedules transactions in arrival order and there are race conditions between operations after timestamp assignment, `New-Order` and `Payment` transactions can execute out of order, leading to aborts. In contrast, R-SMF ensures that transactions with later timestamps will wait for conflicting requests with earlier timestamps to complete. R-SMF also schedules transactions with lower conflict costs together, resulting in 31% higher throughput compared to MVSchedO. Our system improves tail latency by 47% and reduces latency variance by 20% (Table 2). Compared to Defer, R-SMF achieves a 1.2× improvement in throughput; since the Warehouse and District keys (hot keys) are known upon transaction instantiation, Defer has a high likelihood of delaying conflicting requests. R-SMF reduces the abort rate to 2.2% compared to OCC, which has a 10.1% abort rate.

YCSB. R-SMF improves throughput by 2.0× compared to OCC and 1.4× compared to MVTSO (Figure 5). R-SMF also has 35% higher throughput compared to MVSchedO, demonstrating the impact of SMF. YCSB has a large pool of warm keys and higher variance in accesses since keys are chosen from a Zipfian distribution ($\theta = 0.90$). Since there are diverse contention patterns, there are also more schedules with low makespan (which SMF identifies). Defer performs well on this workload, showing a 1.4× increase compared to OCC, since it finds and delays conflicting transactions with high probability (due to the skewed access patterns). On the other hand, Defer-MVTSO and MVTSO have similar performance since both observe aborts from concurrent conflicting transactions that have already been assigned timestamps. R-SMF is able to decrease tail latency by 101% and reduce latency variance by 45% (Table 2) because it prevents transactions accessing hot keys from repeatedly aborting (as does Defer by 81% and 33%, respectively). R-SMF reduces the abort rate to 2.0% compared to 7.7% under OCC.

5.1.2 Bolt-On Results. We also evaluate SMF layered on top of OCC (SMF-OCC) and Lock (SMF-Lock), showing that we can achieve significant scheduling wins with minimal changes to the existing concurrency control implementations. Across the five benchmarks, OCC and Lock have similar performance because they encounter similar conflicts when processing requests in FIFO order and abort at nearly equal rates. SMF-OCC and SMF-Lock are able to prevent many of these conflicts and have comparable performance (Figure 6). On Epinions, we observe a 2.4× increase in throughput, comparing SMF-OCC to OCC and SMF-Lock to Lock. For SmallBank, these schedulers improve throughput by 2.0× compared to their respective baselines. We see an even larger improvement of 3.3× on TAOBench. We note that the relative increase by SMF-OCC and SMF-Lock over their respective baselines is larger than that between R-SMF and MVTSO (2.0×, 1.9×, and 2.6× for these three workloads) since MVTSO enables greater concurrency and achieves higher throughput. On the other hand, the improvements of the schedulers compared to their respective baselines are smaller than R-SMF compared to OCC/Lock because SMF-OCC and SMF-Lock can only delay the transaction start (they do not impact operation execution once a transaction begins) while R-SMF utilizes fine-grained operation scheduling to extract bigger wins.

5.1.3 Weaker Isolation Levels. Scheduling provides benefits across isolation levels. We implement SMF on top of the default concurrency control protocols in RocksDB that provide SI (SMF-OCC) and RC (SMF-RC). Figure 7 shows that scheduling improves throughput across all workloads under RC (up to 2.5x), though the relative improvement is less than that under SI (between OCC and SMF-OCC); this is because fewer conflicts occur under RC compared to SI.

5.1.4 TAO Prototype Results. SMF achieves up to 252% higher throughput and 208% lower tail latency on the TAOBench workload (Table 3) by intelligently ordering transactions to reduce the cost of conflicts. Furthermore, we find that SMF has minimal overheads on TAO, demonstrating that our policy can be feasibly applied in production to realize the benefits of transaction scheduling.

5.2 Scheduling Overheads

To quantify the overheads of scheduling, we measure performance under low ($\theta = 0.10$) and medium ($\theta = 0.50$) contention with the YCSB workload (Figure 8). We compare a read-dominant workload (95% reads) as well as a write-intensive workload (80% writes). Throughput is higher in general on the read-dominant workload though scheduling has a bigger impact on the write-intensive one since its operations are more likely to conflict. Under low contention, our scheduler imposes minimal overhead. R-SMF and SMF-OCC throughput are within 5% of that of MVTSO and OCC, respectively, for both workloads. Since transactions that do not access hot keys are allowed to execute immediately, the only overheads we impose are the classification step and makespan calculations. SMF has linear time complexity with respect to the number of in-flight transactions (which is typically bounded), and makespan calculations depend on the number of hot keys (which is also bounded). Once

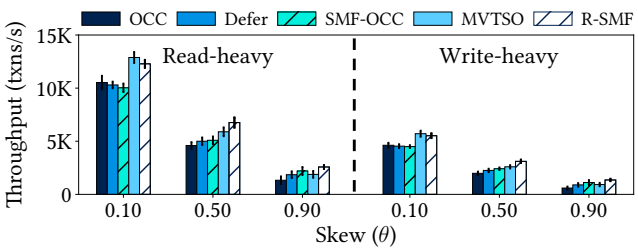

Figure 8: Performance under varying contention.

there is some contention (θ = 0.5), our schedulers show improvements in throughput compared to the baselines (1.4× improvement comparing R-SMF to OCC for the read-dominant workload and 1.6× improvement for the write-dominant workload). Figure 9b shows the latency breakdown for the TPC-C workload: scheduling has limited impact on overall execution time.

Furthermore, we find R-SMF scales effectively: as we increase the number of Warehouses in the TPC-C workload, contention decreases, and throughput grows linearly (Figure 9a). By 40 Warehouses, low contention causes scheduling to have minimal impact. However, both R-SMF and SMF-OCC do not impose undue overheads and have nearly equal performance with the other baselines.

5.3 Classifier Accuracy

We run a series of additional experiments to understand the effects of classifier error, and we find that, as expected, prediction accuracy directly impacts throughput. For our application benchmarks, which include a representative subset of OLTP workloads [31], our classifier has high accuracy since hot keys are easily predictable with application hints. For instance, prediction on TPC-C is always correct since the transaction type and hot keys (Warehouse and District keys) are provided. This information is sufficient to determine hot key access patterns (e.g., `Payment` always reads and writes to both Warehouse and District keys). Accordingly, we use this workload to construct several scenarios in which we deterministically set classifier accuracy and measure its impact (Table 4) .

Decreasing classifier accuracy detrimentally impacts throughput. At one extreme, we assume no application hints are available: the classifier cannot make predictions, so R-SMF is unable to provide any performance benefits and suffers from a small throughput drop (2%) as a result of scheduling overhead. When the hints are wrong 10% of the time (i.e., incorrect Warehouse/District keys), throughput decreases, but we still observe benefits from scheduling (25% improvement in throughput and 29% reduction in tail latency), though this is lower than when all hints are provided (33% improvement in throughput and 35% reduction in tail latency). However, when classifier accuracy drops significantly (50% of hints are wrong), we find that scheduling harms performance because it leads to false positives (delaying transactions that do not conflict) and false negatives (missing potential conflicts). To avoid this, users can choose to forgo scheduling if classifier accuracy drops below a given threshold (e.g. via post-execution sampling analysis, as described in Section 3.4).

Finally, we consider the scenario in which we have partial information for prediction—only transaction type is known. We know the probability of conflict between requests (10% between `Payment` transactions, 1% between `New-Order` transactions, and 10% between `New-Order` and `Payment`), but we do not know exactly which requests will contend since we lack hot key information. As a result,

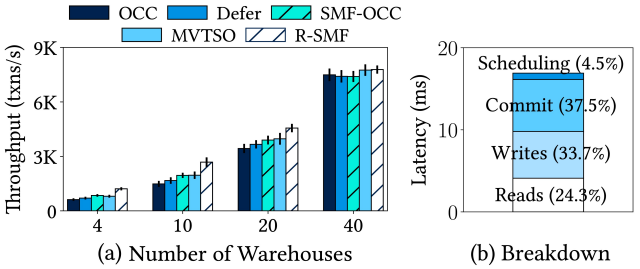

Figure 9: R-SMF scalability and overheads on TPC-C.

Table 4: Performance difference from MVTSO on TPC-C.

Policy	Throughput	P99 Latency
No Hints	-2%	-2%
10% Wrong	25%	29%
50% Wrong	-11%	-5%
Only Types	-5%	-3%
R-SMF (All Hints)	33%	35%

we have R-SMF probabilistically delay requests based on the likelihood of conflict from their type (assuming an equal proportion of each type, we have a 10% chance of conflict for `Payment` and 5.5% for `New-Order`). This approach leads to slightly lower performance compared to MVTSO. Since the conflict rate is low (≤10%) and R-SMF also delays requests at a low rate (≤10%), the probability that R-SMF correctly delays a transaction to prevent a conflict is even lower (≤10% × ≤10% = ≤1%). Thus, most of the scheduling delays imposed by R-SMF are false positives and harm throughput. These results demonstrate a fundamental requirement of scheduling: sufficient information about access patterns (especially hot key accesses) must be available to find fast schedules.

5.4 Evaluating SMF's Search Quality

To evaluate SMF's effectiveness, we compare SMF to state-of-the-art JSS search techniques as well as scheduling policies developed for transactional databases. SMF's schedule makespan is within 10% of that of the best-performing JSS policy, which has much higher overheads. Against transactional policies, SMF provides up to 55% lower makespan and 164% lower variance. We also analyze the impact of our SMF optimizations, which reduce run time overheads but have minimal detrimental impact on search results.

5.4.1 Comparison with Search Policies. We compare SMF to a range of search techniques developed for job-shop scheduling (JSS), which can be used to model transaction scheduling. JSS techniques are designed to navigate complex schedule spaces and consequently, are able to find good schedules on transactional workloads in the offline setting. However, the overheads of these techniques are too high (e.g., in the order of seconds for each scheduling decision [69]) for us to obtain meaningful results in a real system, so we evaluate these policies in our makespan simulator.

Job-Shop Scheduling (JSS). Transaction scheduling can be framed as an instance of the well-studied JSS problem. JSS is a classic scheduling problem that focuses on assigning jobs to machines to minimize makespan [45]: each job consists of multiple tasks that must be executed in a given order on specific machines. We model the offline transaction scheduling problem (known batch of transactions and access sets) as a variation of JSS in which jobs are transactions, tasks are operations, and machines are data items. Transactions require two additional classes of constraints. First, there are different operation types (namely, read and write) for

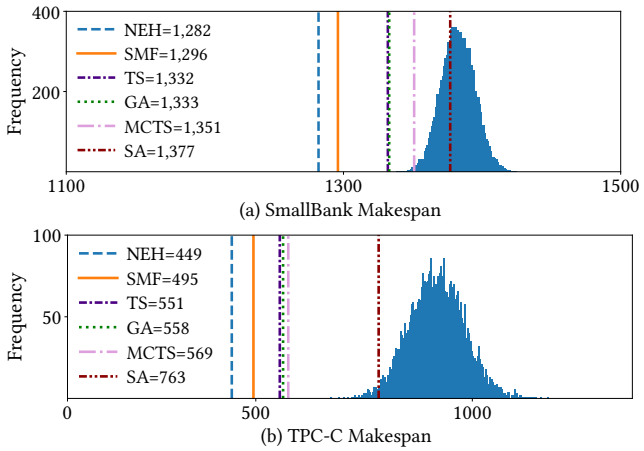

Figure 10: NEH and SMF outperform other JSS search techniques on SmallBank and TPC-C (500 transactions each).

each key, which can be modeled via parallel machines [69]. Second, to account for isolation guarantees, we adapt sequence-dependent execution requirements used by some JSS problems [73]. Interested readers can find the formal optimization problem in the extended version of our paper [2] and further discussion of JSS in Section 6.

With this framework in hand, we now apply JSS search techniques to transaction scheduling. Many JSS techniques have been developed to obtain fast schedules, and we evaluate a range of representative techniques on our application benchmarks.

Algorithms for comparison. We focus on JSS approximation techniques since they are suited for larger workloads (e.g., more than 20 transactions). These strategies are can be categorized as either *iterative* approaches, which randomly perturb a schedule to find faster ones, or *constructive* approaches, which build up a schedule from scratch by leveraging workload features [45]. From iterative approaches, we evaluate the performance of techniques from the three most popular categories: genetic algorithms (GA) [25, 29], simulated annealing (SA) [20, 84], and tabu search (TS) [16, 30, 42, 69]. Among construction heuristics, we focus on the Nawaz–Enscore–Ham (NEH) algorithm [57], which is known to be high performing [44, 53]. This algorithm first sorts transactions based on expected execution time and then iterates through all unscheduled transactions, adding each to the position in the partial schedule that minimizes makespan. Note that NEH *inserts* transactions (following a fixed order) into various positions of the schedule while SMF *appends* a transaction (chosen among a sample of candidates) to the end of a schedule and does not sort requests.

There also has been recent work applying reinforcement learning (RL) to scheduling problems. We consider Monte Carlo Tree Search (MCTS), a popular algorithm that expands a search tree using random sampling, since it has been applied to JSS [50, 70].

Results. We evaluate JSS policies assuming MVTSO on 500 transactions of each of our benchmarks. Table 5 shows the makespan obtained by SMF and the best-performing JSS policy. Figures 10a and 10b show the makespan of all policies on the SmallBank and TPC-C workloads, respectively. SMF uses a sample size of five and considers only hot keys for makespan calculations.

SMF has robust performance on all workloads, finding schedules with less than 10% difference compared to the best JSS techniques. In general, JSS policies are able to find fast schedules: NEH is the highest-performing JSS method on all workloads except Epinions

Table 5: Makespan distribution statistics (100K samples) and schedule makespans under MVTSO.

Workload	Mean	Var.	SMF	Best JSS Policy
Epinions	106	27	74	74 (All)
SmallBank	1,385	131	1,296	1,282 (NEH)
TAOBench	1,703	22,419	959	868 (NEH)
TPC-C	905	12,290	495	449 (NEH)
YCSB	2,587	3,150	1,434	1,298 (NEH)

(for which all the techniques find schedules with the same makespan since this workload has many read-only transactions). This is because NEH's strategy of inserting requests at positions that reduce makespan leads to fast schedules. SMF follows a similar intuition by appending transactions that minimize overall makespan. We note that both policies perform better than the best random sample among 100K samples, achieving makespans in the 99.999^{th} percentile. Furthermore, the best random sample among 100K samples (and consequently, the schedules found by SMF) is better than 99.99% of all possible schedules with 99.99% confidence.

The other JSS techniques have moderate success in scheduling transactions. These strategies are iterative, searching in the local neighborhood of a starting schedule by randomly swapping transactions. However, since hot key conflict patterns impact makespan the most (Section 3.2), arbitrarily reordering transactions usually takes longer to find a fast schedule. Similarly, since MCTS relies heavily on random search, it does not converge quickly.

We find that all JSS techniques have higher overheads than SMF. We measure the latency of each technique in our offline Python simulator to provide a rough estimate of how expensive JSS policies are since they cannot be feasibly implemented in a real database. For a batch of 500 transactions, our single-threaded, unoptimized implementation of SMF finds a schedule in less than five seconds. In contrast, NEH takes 50 seconds, and the other policies take over 500 seconds to run. Furthermore, JSS methods are not compatible with interactive systems (e.g., NEH inserts transactions into arbitrary positions of the schedule, which can violate serializability).

5.4.2 Comparison with Scheduling Policies. We also compare SMF against scheduling policies designed for transactional databases. Specifically, we evaluate three state-of-the-art policies from recent work. Largest-Dependency-Set-First (LDSF) [81] is an online lock scheduling algorithm that gives priority to transactions that block more requests; we follow the order determined by LDSF but evaluate makespan under MVTSO (rather than Lock) for a fair comparison. We also evaluate Defer and $\text{TS}_{\text{KD}}[\text{C}]$, which partitions the workload based on key accesses and is the best-performing scheduler for deterministic databases [19]. For each policy, we measure the makespan and variance of the best schedule over 100 random arrival orders of 500 transactions from each of our benchmarks.

Compared to SMF, these policies find slower schedules (up to 55% higher makespan) and are more sensitive to arrival order (up to 164% higher variance). As Table 6 shows, makespan differences are smaller on Epinions and SmallBank, which contain many short, read-only transactions. $\text{TS}_{\text{KD}}[\text{C}]$ has the best performance out of the three policies, which is expected since it is designed for the offline setting (i.e., deterministic DBs) and uses key access information to lower conflict costs. However, it schedules requests with coarse granularity: a transaction must fit cleanly into a concurrently executing partition or it must run sequentially with other "residual" transactions after all partitions have committed. In contrast,

Table 6: Increase in makespan between best schedule from Defer (D), LDSF (L), and $TS_{KD}[C]$ (T) compared to SMF. Increase in variance compared to SMF (D-V, L-V, T-V).

Workload	D	L	T	D-V	L-V	T-V
Epinions	0%	0%	0%	109%	52%	41%
SmallBank	6.6%	5.8%	4.4%	121%	96%	61%
TAOBench	55%	47%	31%	153%	147%	103%
TPC-C	49%	37%	18%	164%	159%	42%
YCSB	52%	39%	28%	129%	87%	67%

SMF determines schedules on a per-transaction basis. LDSF shows some improvements, but it is highly dependent on arrival order as evidenced by its high variance. Finally, Defer returns schedules of similar makespan to random sampling since Defer essentially follows arrival order in our simulator, which calculates best-case makespans and assumes no aborts.

These results illustrate that existing policies are highly dependent on arrival order. As a comparison, on a small workload (e.g., 20 TAOBench transactions), the makespan and variance differences between the three policies and SMF are less than 25% and 40%, respectively. In real-world workloads that have more conflict patterns and possible schedules, these policies are less likely to encounter arrival orders that lead to fast schedules.

5.4.3 SMF Optimizations. Finally, we investigate the impact of SMF's optimizations (random sampling and using only hot key information), which reduce run time overheads, on schedule makespan. The default policy uses a sample size of five and considers only hot keys. We evaluate a no sampling variant (NS), an all keys variant (AK) that has information about all key accesses, and a combination of these two policies (NS + AK). We measure the makespan decrease (i.e., performance improvement) compared to default SMF on 500 transactions of our benchmarks assuming MVTSO (Table 7).

SMF's run time optimizations have limited impact on makespan. For instance, there is only a 6.2% difference between the schedules produced by default SMF and SMF with no sampling and all key access information (NS + AK) on TPC-C. Makespan differences are minimal on Epinions and SmallBank because many schedules share the same conflict patterns. On TAOBench, all key access information is available upfront, so there is no difference between default SMF and AK. In contrast, AK has better performance on YCSB because there is a long tail of cold key accesses. Since most transactional workloads have diverse access patterns, a small sample size is sufficient to encounter a transaction with low conflict costs. Furthermore, these findings confirm our observation in Section 3.2 that hot key conflicts have the greatest impact on makespan.

6 RELATED WORK

Transaction scheduling. Kung and Papadimitriou proved early on that transaction scheduling is NP-Hard [60]. Subsequent research has mainly centered around concurrency control mechanisms, such as locking [15], to improve performance. Most of these techniques operate within the implicit constraint of arrival (FIFO) order and deal with conflicts as they appear [1, 3, 4, 39, 51, 58, 62, 77, 82, 85, 87, 88, 94]. Other techniques address the transaction schedule more explicitly by assigning a schedule based on arrival order [14] or reordering the schedule after transaction commit [54] and/or abort [18, 33]. Deterministic databases, which assume apriori access to read-write sets, also schedule batches of requests explicitly. While many use FIFO order [35–37, 80], some approaches find better schedules by partitioning workloads based on hot keys [28, 61, 63–65, 95]. Recent research proposes scheduling transactions without assuming full knowledge of key accesses by predicting hot keys and probabilistically delaying requests [19] or by learning abort patterns between pairs of transactions [74]. Most techniques focus on improving throughput, though there is a line of work that reduces latency by scheduling transactions based on dependency set sizes [41, 81] or assigned priorities [21, 92, 93]. Overall, these approaches consider only a small subset of the schedule space for efficiency. In contrast, we systematically study the entire schedule space to develop SMF, which efficiently finds fast schedules. R-SMF applies SMF with predicted hot key accesses and a schedule-centric concurrency control protocol, MVSchedO, to improve throughput.

Real-time databases (RTDB). There is some work on transaction scheduling for RTDBs, which assume each transaction comes with a pre-specified deadline [6, 7, 52, 59, 66, 76, 83, 90]. These scheduling algorithms (e.g., Earliest-Deadline-First [52, 76, 90]) typically focus on minimizing the number of missed deadlines and are not applicable to general DBMSs, which do not have workloads with deadlines and focus mainly on maximizing throughput.

Job-shop scheduling (JSS). We observe that transaction scheduling can be framed as an instance of JSS [11]. JSS optimization methods provide exact solutions through full enumeration [49] but can only be applied to small problems. On the other hand, approximation methods are used for larger problems [8, 44, 47, 53, 57, 89], and a range of policies have been developed to find low makespan schedules, including genetic algorithms [25, 29], simulated annealing [20, 84], tabu search [16, 30, 42, 69], and hybrids of different approaches [26, 38, 71, 86, 96]. While these methods find fast schedules, their overheads are too high for them to be applied directly to DBMSs. SMF is able to match the makespan of the best JSS techniques while having much lower run time overheads.

Table 7: Decrease in makespan of variants of SMF.

Workload	NS	AK	NS + AK
Epinions	0%	0%	0%
SmallBank	2.6%	3.0%	5.1%
TAOBench	7.0%	0%	7.0%
TPC-C	5.3%	1.2%	6.2%
YCSB	3.0%	5.8%	8.4%

7 CONCLUSION

In this work, we demonstrate the power of searching through the schedule space for fast schedules and precisely executing them. R-SMF leverages a greedy policy, SMF, and a schedule-first concurrency control protocol, MVSchedO, to significantly improve performance. The benefits of our approach provide compelling evidence that search-based scheduling is a promising direction for extracting higher throughput from database systems.

ACKNOWLEDGMENTS

We thank Dave Cecere, Shilpa Lawande, John Hugg, Nathan Bronson, and the VLDB anonymous reviewers for their insightful feedback. This work is supported by a Meta Next-Generation Infrastructure award, and gifts from Accenture, AMD, Anyscale, Google, IBM, Intel, Mohamed Bin Zayed University of Artificial Intelligence, Samsung SDS, SAP, and VMware.

REFERENCES

[1] 2020. MySQL Transactional and Locking Statements. https://dev.mysql.com/doc/refman/8.0/en/sql-transactional-statements.html

[2] 2023. DariusDB. https://github.com/audreyccheng/DariusDB

[3] 2024. CockroachDB Transaction Layer. https://www.cockroachlabs.com/docs/stable/architecture/transaction-layer

[4] 2024. PostgreSQL. https://www.postgresql.org/

[5] 2024. RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide

[6] Robert Abbott and Hector Garcia-Molina. 1988. Scheduling Real-Time Transactions. *Acm Sigmod Record* 17, 1 (1988), 71–81.

[7] Robert K Abbott and Hector Garcia-Molina. 1992. Scheduling Real-Time Transactions: A Performance Evaluation. *ACM Transactions on Database Systems (TODS)* 17, 3 (1992), 513–560.

[8] Joseph Adams, Egon Balas, and Daniel Zawack. 1988. The Shifting Bottleneck Procedure for Job Shop Scheduling. *Management Science* 34, 3 (1988), 391–401.

[9] Atul Adya, Barbara Liskov, and Patrick O'Neil. 2000. Generalized Isolation Level Definitions. (2000), 67–78.

[10] Phillipe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. 2015. Challenges to Adopting Stronger Consistency at Scale. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. USENIX Association, Kartause Ittingen, Switzerland. https://www.usenix.org/conference/hotos15/workshop-program/presentation/ajoux

[11] Sheldon B. Akers, Jr. and Joyce Friedman. 1955. A Non-Numerical Approach to Production Scheduling Problems. *Journal of the Operations Research Society of America* 3, 4 (1955), 429–442.

[12] Christos A Athanasiadis and Persi Diaconis. 2010. Functions of random walks on hyperplane arrangements. *Advances in Applied Mathematics* 45, 3 (2010), 410–437.

[13] Peter Bailis, Alan Fekete, Joseph M. Hellerstein, Ali Ghodsi, and Ion Stoica. 2014. Scalable Atomic Visibility with RAMP Transactions. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 27–38.

[14] Philip A Bernstein and Nathan Goodman. 1983. Multiversion Concurrency Control—Theory and Algorithms. *ACM Transactions on Database Systems (TODS)* 8, 4 (1983), 465–483.

[15] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley.

[16] Paolo Brandimarte. 1993. Routing and scheduling in a flexible job shop by tabu search. *Annals of Operations research* 41, 3 (1993), 157–183.

[17] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *2013 USENIX Annual Technical Conference (USENIX ATC '13)*. 49–60.

[18] Matthew Burke, Florian Suri-Payer, Jeffrey Helt, Lorenzo Alvisi, and Natacha Crooks. 2023. Morty: Scaling Concurrency Control with Re-Execution. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 687–702.

[19] Yang Cao, Wenfei Fan, Weijie Ou, Rui Xie, and Wenyue Zhao. 2023. Transaction Scheduling: From Conflicts to Runtime Conflicts. *Proc. ACM Manag. Data* 1, 1, Article 26 (may 2023), 26 pages.

[20] Shouvik Chakraborty and Sandeep Bhowmik. 2015. An Efficient Approach to Job Shop Scheduling Problem Using Simulated Annealing. *International Journal of Hybrid Information Technology* 8, 11 (2015), 273–284.

[21] Youmin Chen, Xiangyao Yu, Paraschos Koutris, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. 2022. Plor: General Transactions with Predictable, Low Tail Latency. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 19–33.

[22] Audrey Cheng, David Chu, Terrance Li, Jason Chan, Natacha Crooks, Joseph M. Hellerstein, Ion Stoica, and Xiangyao Yu. 2023. Take Out the TraChe: Maximizing (Tra)nsactional Ca(che) Hit Rate. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 419–439.

[23] Audrey Cheng, Xiao Shi, Aaron Kabcenell, Shilpa Lawande, Hamza Qadeer, Jason Chan, Harrison Tin, Ryan Zhao, Peter Bailis, Mahesh Balakrishnan, Nathan Bronson, Natacha Crooks, and Ion Stoica. 2022. TAOBench: An End-to-End Benchmark for Social Network Workloads. *Proc. VLDB Endow.* 15, 9 (may 2022), 1965–1977.

[24] Audrey Cheng, Xiao Shi, Lu Pan, Anthony Simpson, Neil Wheaton, Shilpa Lawande, Nathan Bronson, Peter Bailis, Natacha Crooks, and Ion Stoica. 2021. RAMP-TAO: Layering Atomic Transactions on Facebook's Online TAO Data Store. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3014–3027.

[25] Runwei Cheng, Mitsuo Gen, and Yasuhiro Tsujimura. 1996. A tutorial survey of job-shop scheduling problems using genetic algorithms—I. Representation. *Computers & industrial engineering* 30, 4 (1996), 983–997.

[26] Runwei Cheng, Mitsuo Gen, and Yasuhiro Tsujimura. 1999. A tutorial survey of job-shop scheduling problems using genetic algorithms, part II: hybrid genetic search strategies. *Computers & Industrial Engineering* 36, 2 (1999), 343–364.

[27] The Transaction Processing Performance Council. 2021. TPC-C. http://www.tpc.org/tpcc/

[28] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: A Workload-Driven Approach to Database Replication and Partitioning. *Proc. VLDB Endow.* 3, 1–2 (sep 2010), 48–57.

[29] Lawrence Davis. 2014. Job Shop Scheduling with Genetic Algorithms. In *Proceedings of the first International Conference on Genetic Algorithms and their Applications*. Psychology Press, 136–140.

[30] Mauro Dell'Amico and Marco Trubian. 1993. Applying tabu search to the job-shop scheduling problem. *Annals of Operations research* 41, 3 (1993), 231–252.

[31] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proceedings of the VLDB Endowment* 7, 4, 277–288.

[32] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB.. In *CIDR*, Vol. 3. 3.

[33] Zhiyuan Dong, Zhaoguo Wang, Xiaodong Zhang, Xian Xu, Changgeng Zhao, Haibo Chen, Aurojit Panda, and Jinyang Li. 2023. Fine-Grained Re-Execution for Efficient Batched Commit of Distributed Transactions. *Proc. VLDB Endow.* 16, 8 (apr 2023), 1930–1943.

[34] Facebook. 2023. RocksDB Github. https://github.com/facebook/rocksdb

[35] Jose M. Faleiro and Daniel J. Abadi. 2015. Rethinking Serializable Multiversion Concurrency Control. *Proc. VLDB Endow.* 8, 11 (jul 2015), 1190–1201.

[36] Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. 2017. High Performance Transactions via Early Write Visibility. *Proceedings of the VLDB Endowment* 10, 5 (2017).

[37] Jose M Faleiro, Alexander Thomson, and Daniel J Abadi. 2014. Lazy Evaluation of Transactions in Database Systems. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 15–26.

[38] José Fernando Gonçalves, Jorge José de Magalhães Mendes, and Mauricio GC Resende. 2005. A Hybrid Genetic Algorithm for the Job Shop Scheduling Problem. *European journal of operational research* 167, 1 (2005), 77–95.

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

[40] G.J. Hahn and W.Q. Meeker. 1991. *Statistical Intervals.* Wiley & Sons, Inc, New York, NY.

[41] Jiamin Huang, Barzan Mozafari, Grant Schoenebeck, and Thomas F Wenisch. 2017. A Top-Down Approach to Achieving Performance Predictability in Database Systems. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 745–758.

[42] Johann Hurink, Bernd Jurisch, and Monika Thole. 1994. Tabu search for the job-shop scheduling problem with multi-purpose machines. *Operations-Research-Spektrum* 15 (1994), 205–215.

[43] Benjamin Iriarte Giraldo. 2015. *Combinatorics of acyclic orientations of graphs: algebra, geometry and probability.* Ph. D. Dissertation. Massachusetts Institute of Technology.

[44] R. Leisten J. M. Framinan and C. Rajendran. 2003. Different initial sequences for the heuristic of Nawaz, Enscore and Ham to minimize makespan, idletime or flowtime in the static permutation flowshop sequencing problem. *International Journal of Production Research* 41, 1 (2003), 121–148.

[45] A.S. Jain and S. Meeran. 1999. Deterministic job-shop scheduling: Past, present and future. *European Journal of Operational Research* 113, 2 (1999), 390–434.

[46] E Douglas Jensen, C Douglass Locke, and Hideyuki Tokuda. 1985. A Time-Driven Scheduling Model for Real-Time Operating Systems. In *Rtss*, Vol. 85. 112–122.

[47] John J Kanet and Jack C Hayya. 1982. Priority dispatching with operation due dates in a job shop. *Journal of operations Management* 2, 3 (1982), 167–175.

[48] Aman Kataria and MD Singh. 2013. A Review of Data Classification Using K-Nearest Neighbour Algorithm. *International Journal of Emerging Technology and Advanced Engineering* 3, 6 (2013), 354–360.

[49] Eugene L Lawler, Jan Karel Lenstra, Alexander HG Rinnooy Kan, and David B Shmoys. 1993. Sequencing and scheduling: Algorithms and complexity. *Handbooks in operations research and management science* 4 (1993), 445–522.

[50] Kexin Li, Qianwang Deng, Like Zhang, Qing Fan, Guiliang Gong, and Sun Ding. 2021. An effective MCTS-based algorithm for minimizing makespan in dynamic flexible job shop scheduling problem. *Computers & Industrial Engineering* 155 (2021), 107211.

[51] Hyeontaek Lim, Michael Kaminsky, and David G Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 21–35.

[52] C. L. Liu and James W. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM* 20, 1 (jan 1973), 46–61.

[53] Weibo Liu, Yan Jin, and Mark Price. 2017. A new improved NEH heuristic for permutation flowshop scheduling problems. *International Journal of Production Economics* 193 (2017), 21–30.

[54] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: A Fast and Practical Deterministic OLTP Database. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2047–2060.

[55] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. 2015. Retro: Targeted Resource Management in Multi-Tenant Distributed Systems. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Oakland, CA) *(NSDI'15)*. USENIX Association, USA, 589–603.

[56] Vivek Narasayya, Sudipto Das, Manoj Syamala, Badrish Chandramouli, and Surajit Chaudhuri. 2013. SQLVM: Performance Isolation in Multi-Tenant Relational Database-as-a-Service. In *CIDR 2013* (cidr 2013 ed.). 6th Biennial Conference on Innovative Data Systems Research.

[57] Muhammad Nawaz, E Emory Enscore Jr, and Inyong Ham. 1983. A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *Omega* 11, 1 (1983), 91–95.

[58] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 677–689.

[59] Gultekin Ozsoyoglu and Richard T Snodgrass. 1995. Temporal and real-time databases: A survey. *IEEE Transactions on Knowledge and Data Engineering* 7, 4 (1995), 513–532.

[60] Christos H. Papadimitriou. 1979. The Serializability of Concurrent Database Updates. *J. ACM* 26, 4, 631–653. https://doi.org/10.1145/322154.322158

[61] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. 2012. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 61–72.

[62] Dan R. K. Ports and Kevin Grittner. 2012. Serializable Snapshot Isolation in PostgreSQL. *Proc. VLDB Endow.* 5, 12 (aug 2012), 1850–1861.

[63] Guna Prasaad, Alvin Cheung, and Dan Suciu. 2020. Handling Highly Contended OLTP Workloads Using Fast Dynamic Partitioning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 527–542.

[64] Thamir M Qadah and Mohammad Sadoghi. 2018. QueCC: A Queue-oriented, Control-free Concurrency Architecture. In *Proceedings of the 19th International Middleware Conference*. 13–25.

[65] Dai Qin, Angela Demke Brown, and Ashvin Goel. 2021. Caracal: Contention Management with Deterministic Concurrency Control. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 180–194.

[66] Krithi Ramamritham. 1993. Real-time databases. *Distributed and parallel databases* 1 (1993), 199–226.

[67] Rubén Ruiz, Quan-Ke Pan, and Bahman Naderi. 2019. Iterated Greedy methods for the distributed permutation flowshop scheduling problem. *Omega* 83 (2019), 213–222.

[68] Rubén Ruiz and Thomas Stützle. 2007. A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. *European journal of operational research* 177, 3 (2007), 2033–2049.

[69] Mohammad Saidi-Mehrabad and Parviz Fattahi. 2007. Flexible job shop scheduling with tabu search algorithms. *International Journal of Advanced Manufacturing Technology* 32, 5-6 (2007), 563–570.

[70] M Saqlain, S Ali, and JY Lee. 2023. A Monte-Carlo tree search algorithm for the flexible job-shop scheduling in manufacturing systems. *Flexible Services and Manufacturing Journal* 35, 2 (2023), 548–571.

[71] DY Sha and Cheng-Yu Hsu. 2006. A hybrid particle swarm optimization for job shop scheduling problem. *Computers & Industrial Engineering* 51, 4 (2006), 791–808.

[72] Gaurav Sharma, Ravi R Mazumdar, and Ness B Shroff. 2006. On the complexity of scheduling in wireless networks. In *Proceedings of the 12th annual international conference on Mobile computing and networking*. 227–238.

[73] Liji Shen, Stéphane Dauzère-Pérès, and Janis S Neufeld. 2018. Solving the flexible job shop scheduling problem with sequence-dependent setup times. *European journal of operational research* 265, 2 (2018), 503–516.

[74] Yangjun Sheng, Anthony Tomasic, Tieying Zhang, and Andrew Pavlo. 2019. Scheduling OLTP Transactions via Learned Abort Prediction. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management* (Amsterdam, Netherlands) *(aiDM '19)*. Association for Computing Machinery, New York, NY, USA, Article 1, 8 pages.

[75] David Shue, Michael J. Freedman, and Anees Shaikh. 2012. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 349–362.

[76] John A Stankovic, Marco Spuri, Krithi Ramamritham, and Giorgio Buttazzo. 1998. *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*. Vol. 460. Springer Science & Business Media.

[77] Chunzhi Su, Natacha Crooks, Cong Ding, Lorenzo Alvisi, and Chao Xie. 2017. Bringing Modular Concurrency Control to the Next Level. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 283–297.

[78] Dixin Tang, Hao Jiang, and Aaron J. Elmore. 2017. Adaptive Concurrency Control: Despite the Looking Glass, One Concurrency Control Does Not Fit All. In *CIDR*, Vol. 2.

[79] The H-Store team. 2013. SmallBank Benchmark. http://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/

[80] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 1–12.

[81] Boyu Tian, Jiamin Huang, Barzan Mozafari, and Grant Schoenebeck. 2018. Contention-Aware Lock Scheduling for Transactional Databases. *Proceedings of the VLDB Endowment* 11, 5 (2018), 648–662.

[82] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 18–32.

[83] Özgür Ulusoy and Geneva G Belford. 1993. Real-time transaction scheduling in database systems. *Information Systems* 18, 8 (1993), 559–580.

[84] Peter JM Van Laarhoven, Emile HL Aarts, and Jan Karel Lenstra. 1992. Job Shop Scheduling by Simulated Annealing. *Operations research* 40, 1 (1992), 113–125.

[85] Jia-Chen Wang, Ding Ding, Huan Wang, Conrad Christensen, Zhaoguo Wang, Haibo Chen, and Jinyang Li. 2021. Polyjuice: High-Performance Transactions via Learned Concurrency Control. In *OSDI*. 198–216.

[86] Ling Wang and Da-Zhong Zheng. 2001. An effective hybrid optimization strategy for job-shop scheduling problems. *Computers & Operations Research* 28, 6 (2001), 585–596.

[87] Zhaoguo Wang, Shuai Mu, Yang Cui, Han Yi, Haibo Chen, and Jinyang Li. 2016. Scaling Multicore Databases via Constrained Parallel Execution. In *Proceedings of the 2016 International Conference on Management of Data*. 1643–1658.

[88] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. 2015. High-Performance ACID via Modular Concurrency Control. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 279–294.

[89] Jin Xie, Liang Gao, Kunkun Peng, Xinyu Li, and Haoran Li. 2019. Review on flexible job shop scheduling. *IET collaborative intelligent manufacturing* 1, 3 (2019), 67–77.

[90] Ming Xiong, Qiong Wang, and Krithi Ramamritham. 2008. On earliest deadline first scheduling for temporal consistency maintenance. *Real-Time Systems* 40 (2008), 208–237.

[91] Cong Yan and Alvin Cheung. 2016. Leveraging Lock Contention to Improve OLTP Application Performance. *Proc. VLDB Endow.* 9, 5 (jan 2016), 444–455.

[92] Linguan Yang, Xinan Yan, and Bernard Wong. 2022. Natto: Providing Distributed Transaction Prioritization for High-Contention Workloads. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 715–729.

[93] Chenhao Ye, Wuh-Chwen Hwang, Keren Chen, and Xiangyao Yu. 2023. Polaris: Enabling Transaction Priority in Optimistic Concurrency Control. *Proc. ACM Manag. Data* 1, 1, Article 44 (may 2023), 24 pages.

[94] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2016 International Conference on Management of Data*. 1629–1642.

[95] Erfan Zamanian, Julian Shun, Carsten Binnig, and Tim Kraska. 2020. Chiller: Contention-centric Transaction Execution and Data Partitioning for Modern Networks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 511–526.

[96] Rui Zhang and Cheng Wu. 2010. A hybrid immune simulated annealing algorithm for the job shop scheduling problem. *Applied Soft Computing* 10, 1 (2010), 79–89.