A MODELING TRANSACTION SCHEDULING AS A JSS PROBLEM

We model transaction scheduling by adding constraints to the job-shop scheduling optimization problem with an objective of minimizing makespan. From the original formulation [74], we have jobs as transactions, tasks as operations, and machines as data items. We assume 2PL is used to ensure serializability. The new constraints we add to model transactions are in blue.

Parameters:

n: Number of txns

m: Number of keys

$O_{j,h}$ for $h = 1, ..., h_j$: hth operation of txn j

$O_{i,j,h}$: Operation $O_{j,h}$ on key i

$p_{i,j,h}$: Processing time of $O_{i,j,h}$

M: A large number

$$y_{i,j,h} = \begin{cases} 1 & \text{if } O_{j,h} \text{ can be performed on key } i \\ 0 & \text{otherwise} \end{cases}$$

$$r_{j,h} = \begin{cases} 1 & \text{if } O_{j,h} \text{ is a read} \\ 0 & \text{otherwise} \end{cases}$$

$$u_{j,h} = \begin{cases} 1 & \text{if } O_{j,h} \text{ is a read in a txn with a write to the same key} \\ 0 & \text{otherwise} \end{cases}$$

Decision variables:

$$x_{i,j,h,k,l} = \begin{cases} 1 & \text{if } O_{i,j,h} \text{ immediately precedes } O_{i,k,l} \\ 0 & \text{otherwise} \end{cases}$$

$t_{j,h}$: Start time of the processing of operation $O_{j,h}$

$f_{j,h}$: Finish time of the processing of operation $O_{j,h}$

$s_{i,j,h}$: Release time of shared lock for key i based on ongoing txns

C_{max}: Makespan of a schedule

Optimization problem:

Min C_{max}

s.t.

(1) $t_{j,h} + y_{i,j,h} * p_{i,j,h} \leq f_{j,h}, \forall i = 1, ..., m; j = 1, ..., n; h = 1, ..., h_j$

(2) $t_{j,h} + p_{i,j,h} \leq t_{j,h+1}, \forall i = 1, ..., m; j = 1, ..., n; h = 1, ..., h_j$

(3) $f_{j,h} \leq f_j, \forall h = 1, ..., h_j$

(4) $f_j \leq C_{max}$

(5) $f_j - r_{j,h} * r_{k,l} * M \leq t_{k,l} + (1 - x_{i,j,h,k,l}) * M, \forall i = 1, ..., m; j = 0, ..., n; h = 1, ..., h_j; k = 1, ..., n; l = 1, ..., h_k; j \neq k$

(6) $t_{j,h} + p_{i,j,h} - r_{j,h} * r_{k,l} * M \leq t_{k,l} + (1 - x_{i,j,h,k,l}) * M, \forall i = 1, ..., m; j = 0, ..., n; h = 1, ..., h_j; k = 1, ..., n; l = 1, ..., h_k$

(7) $f_j - (1 - r_{j,h}) * M \leq s_{i,k,l} + (1 - x_{i,j,h,k,l}) * M, \forall i = 1, ..., m; j = 0, ..., n; h = 1, ..., h_j; k = 1, ..., n; l = 1, ..., h_k; j \neq k$

(8) $s_{i,j,h} - (1 - r_{j,h}) * (1 - r_{k,l}) * M \leq s_{i,k,l} + (1 - x_{i,j,h,k,l}) * M, \forall i = 1, ..., m; j = 0, ..., n; h = 1, ..., h_j; k = 1, ..., n; l = 1, ..., h_k; j \neq k$

(9) $s_{i,j,h} - r_{k,l} * M \leq t_{k,l} + (1 - x_{i,j,h,k,l}) * M, \forall i = 1, ..., m; j = 0, ..., n; h = 1, ..., h_j; k = 1, ..., n; l = 1, ..., h_k; j \neq k$

(10) $\sum_i y_{i,j,h} = 1, \forall j = 0, 1, ..., n; h = 1, ..., h_j$

(11) $\sum_j \sum_h x_{i,j,h,k,l} = y_{i,k,l}, \forall i = 1, ..., m; k = 1, ..., n; l = 1, ..., h_k$

(12) $\sum_k \sum_l x_{i,j,h,k,l} = y_{i,j,h}, \forall i = 1, ..., m; j = 0, 1, ..., n; h = 1, ..., h_j$

(13) $x_{i,j,h,j,h} = 0, \forall i = 1, ..., m; j = 0, 1, ..., n; h = 1, ..., h_j$

(14) $t_{j,h} \geq 0, \forall j = 0, 1, ..., n; h = 1, ..., h_j$

(15) $f_{j,h} \geq 0, \forall j = 0, 1, ..., n; h = 1, ..., h_j$

(16) $y_{i,j,h} \in \{0, 1\}, \forall i = 1, ..., m; j = 0, 1, ..., n; h = 1, ..., h_j$

(17) $x_{i,j,h,k,l} \in \{0, 1\}, \forall i = 1, ..., m; j = 0, 1, ..., n; h = 1, ..., h_j; k = 1, ..., n; l = 1, ..., h_k$

We describe the purpose of each constraint. (1) defines the processing time of each operation. (2) requires that operations in each transaction must follow a specified partial order. (3) ensures that no locks are released until the end of each transaction. (4) defines the makespan. (5) enables shared and read-for-update locks. (6) ensures only one operation can execute at a time on a key unless both are reads. (7) defines the release time of if this operation is a read. (8) defines the release time of a lock if the next operation is a read. (9) ensures a write can only occur after all read locks are released. (10) requires each operation to execute on only one key. (11) and (12) define circular permutations of operations on each machine. They eliminate alternative operations that are excluded in a final schedule. (11) selects one operation $O_{i,j,h}$ that immediately precedes a scheduled alternative operation $O_{i,k,l}$. (12) selects one operation $O_{i,k,l}$that immediately follows a scheduled alternative operation$O_{i,j,h}$. A circular permutation of operations on a machine yields a scheduled sequence of the operations on the same machine. (13) requires that each operation only occurs once. (14) and (15) ensure that operation and transaction execution times cannot be negative. (16) and (17) ensure all operations in all transactions are scheduled.

B EXPLORING THE SCHEDULE SPACE

In this section, we present an approach to systematically investigate the schedule space of transactional workloads. Analyzing the entire space of schedules is inherently challenging due to the exponential number of schedules possible. As a result, we introduce a new framework to characterize the space of schedules for a given workload. We sample the (serializable) schedules of a given workload to construct an empirical distribution of schedule performance (e.g., the 95th percentile "fastest" schedule). This enables us to not only understand the tradeoffs between different scheduling policies but also provide statistical bounds on schedule performance with respect to the *entire* space. While simple, we have not previously observed this kind of workload-dependent empirical characterization in the transaction processing literature.

To capture the performance distribution of a schedule space, we need to: 1) to quantify schedule performance based on an isolation level and a concurrency control protocol and 2) sample schedules in the space. In this section, we focus on the offline setting (assuming a finite batch of transactions and known access sets).

Transactions, schedules, and serialization. We adopt standard transactional formalisms [9, 14]: A *transaction* T_i is a set of operations on data items with ordering constraints. We use $o_i(x)$ to represent an operation of T_i on data item x, which could be either a read $r_i(x)$ or a write $w_i(x)$. $\sum_i$ denotes the set of all operations in T_i, and $<_i$ represents the partial order of the operations. Following past notation [14], we define a transaction *schedule* S over transactions $\{T_1, ..., T_n\}$ as a partially ordered set $(\sum, <_S)$, where:

- $\sum = \bigcup_{i=1}^{n} \sum_i$, i.e., the schedule contains all and only those operations submitted by $T_1 ..., T_n$
- $<_S \supseteq \bigcup_{i=1}^{n} <_i$, i.e., if transaction T_i specifies the order of its operations, they must appear in that order in the schedule

- $\forall r_j(x), \exists i \leq j$ such that $w_i(x) <_S r_j(x)$, i.e., every read of a data item must be preceded by at least one write to the same item, possibly in the same transaction (each item in the database starts with an empty value)
- for any two conflicting operations $o_i, o_j \in S$, either $o_i <_S o_j$ or $o_j <_S o_i$ (two operations *conflict* if they operate on the same copy of a data item, and at least one is a write)

To characterize the schedule space of a transactional workload, we must consider schedules with respect to an isolation level as this determines which schedules are permissible. We focus on serializability for our evaluation since it is widely used and prevents data anomalies for real-world applications. A *serializable schedule* is one that is equivalent to some *serial schedule* (i.e., by reordering non-conflicting operations, we can obtain a schedule in which transactions execute sequentially). A schedule is serializable if its *serialization graph* is acyclic [14]. A serialization graph SG for schedule S, denoted $SG(S)$, is a directed graph in which nodes are transactions in S and edges are all $T_i \rightarrow T_j (i \neq j)$ such that some operation of T_i precedes and conflicts with some operation of T_j. Note that each serializable schedule corresponds with exactly one acyclic serialization graph (assuming $<_i$ is specified for each T_i).

Serializable execution. The schedule space is further constrained by concurrency control protocols, which specify how conflicts are mediated during execution. For instance, under two-phase locking (2PL), a conflicting operation must instead wait until all operations in the conflicting transaction have completed. In contrast, under MVTSO, an operation can execute as soon as the conflicting operation preceding it in the partial order has executed. These execution constraints are captured via a partial ordering on the operations.

Formally, we define $(\sum, <_{CC})$ as the partially ordered set of operations determined by concurrency control protocol CC over schedule S – this set is a superset of $(\sum, <_S)$. We define a *serializable execution graph SEG*, denoted $SEG(S, CC)$, to be a directed graph with nodes representing operations in S and edges as $o_i \rightarrow o_j (i \neq j)$ such that $(\sum, <_{CC})$ requires o_i to precede o_j. For example, Figure 1 shows the serialization graphs and serialization execution graphs for two different serializable schedules of the same workload under MVTSO ($T1 : \{r(x), r(y), w(z)\}$, $T2 : \{r(y), w(z)\}$, $T3 : \{r(a), r(b)\}$). Focusing on the leftmost schedule, we see that the serialization execution graph requires a partial order between the $w(z)$ operations of $T1$ and $T2$. Compared to the serialization graph, the serialization execution graph has operations rather than transactions as nodes. Note that each serializable schedule has one serializable execution graph, which must be acyclic.

Quantifying schedule performance. With a serializable execution graph, we can quantify the performance of the schedule it represents. We evaluate each schedule based on its *makespan*, or the total time required for all transactions to complete. This metric is standard in scheduling research; for a given set of transactions, minimizing makespan is equivalent to maximizing throughput because the faster a batch completes, the higher the throughput.

To compute makespan, we assume "best-case" execution and calculate the *minimum* time it takes to execute a given schedule under a specific concurrency control protocol. That is, we assume that there are no resource constraints (i.e., operations run as early as possible) and that there are no execution errors that lead to aborts. Additionally, to simplify our calculations, we assume each operation takes the same amount of time, during which any number of reads or one write can occur on a given key.

Given these inputs, finding the makespan of a serialization execution graph is straightforward: since the graph must be acyclic, the makespan is simply the length of the longest path. Each serializable schedule maps to a serializable execution graph, which has only one possible makespan. We provide two example makespan calculations in Figure 1. The schedule on the left has $T1$ ordered before $T2$, and its serializable execution graph gives a makespan of four. In contrast, the schedule on the right has $T2$ ordered before $T1$, resulting in a makespan of three.

The makespan enables us to compare the impact of conflicts on overall execution time, and we leverage it to develop an effective search policy (Section 1). 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. We find that scheduling can have an even bigger impact in online systems than what our makespan analysis suggests because aborts can be common and therefore hamper system throughput (Section 4).

Constructing schedule space distributions. We now turn to the problem of empirically constructing the distribution of a space of schedules, with statistical bounds. Exhaustively evaluating all schedules of a space would be prohibitively expensive (e.g., for 20 transactions with one write operation each, there are $20! = 2.4 \times 10^{18}$ possible schedules).

To make this problem tractable, we sample from space of serializable schedules. Doing so uniformly requires some care. We observe that each such schedule corresponds to a serialization graph, so sampling over all serialization graphs produces equivalent outcomes. Each serialization graph has the same underlying (undirected) graph "structure"—nodes represent transactions and edges represent conflicts between transactions—and differs only in the *direction* of the edges. Formally, these serialization graphs are "acyclic orientations" of the underlying undirected graph.

Thus, uniformly sampling over the serialization graphs of a workload is equivalent to uniformly sampling over the acyclic orientations of the workload's graph structure. We employ an existing algorithm [47] that achieves the latter by forming a Markov chain via a random walk. The mixing time of this approach is $n \log n$, where n is the number of transactions in the workload [12]. For example, we would need to take 200 steps along our random walk to converge on a stationary (uniform) distribution for a batch of 100 transactions.

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 [43], we can bound the proportion of schedules that have lower makespan than our best random sample with a given 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.[3] Note that these intervals do not bound the makespan difference of schedules, only the proportion of the distribution that falls beyond the intervals. For a given schedule, we can evaluate its performance with respect to the makespan distribution and thus, the entire schedule space.

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 before execution (e.g., deterministic databases), this technique can be used as the default scheduling policy to ensure that selected schedules have lower makespan than the majority of the *entire* schedule space. For online systems, this sampling technique can be used as a way 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. If SMF performs worse than the best random sample, this may indicate a workload shift (e.g., hot keys have changed), and the system can leverage this information to respond accordingly. We can also use this technique to 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 arrival order.

[3] 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 [43].