Contention-Aware Lock Scheduling for Transactional Databases

Boyu Tian Jiamin Huang Barzan Mozafari Grant Schoenebeck
University of Michigan
Ann Arbor, MI, USA
{bytian, jiamin, mozafari, schoeneb}@umich.edu

ABSTRACT

Lock managers are among the most studied components in concurrency control and transactional systems. However, one question seems to have been generally overlooked: "When there are multiple lock requests on the same object, which one(s) should be granted first?"

Nearly all existing systems rely on a FIFO (first in, first out) strategy to decide which transaction(s) to grant the lock to. In this paper, however, we show that the lock scheduling choices have significant ramifications on the overall performance of a transactional system. Despite the large body of research on job scheduling outside the database context, lock scheduling presents subtle but challenging requirements that render existing results on scheduling inapt for a transactional database. By carefully studying this problem, we present the concept of contention-aware scheduling, show the hardness of the problem, and propose novel lock scheduling algorithms (LDSF and bLDSF), which guarantee a constant factor approximation of the best scheduling. We conduct extensive experiments using a popular database on both TPC-C and a microbenchmark. Compared to FIFO—the default scheduler in most database systems—our bLDSF algorithm yields up to 300x speedup in overall transaction latency. Alternatively, our LDSF algorithm, which is simpler and achieves comparable performance to bLDSF, has already been adopted by open-source community, and was chosen as the default scheduling strategy in MySQL 8.0.3+.

PVLDB Reference Format:
Boyu Tian, Jiamin Huang, Barzan Mozafari, Grant Schoenebeck. Contention-Aware Lock Scheduling for Transactional Databases. *PVLDB*, 11 (5): xxxx-yyyy, 2018.
DOI: 10.1145/3177732.3177740

1. INTRODUCTION

Lock management forms the backbone of concurrency control in modern software, including many distributed systems and transactional databases. A lock manager guarantees both correctness and efficiency of a concurrent application by solving the data contention problem. For example, before a transaction accesses a database object, it has to acquire the corresponding lock; if the transaction fails to get a lock immediately, it is blocked until the system grants it the lock. This poses a fundamental question: when multiple transactions are waiting for a lock on the same object, which should be granted first when the object becomes available? This question, which we call *lock scheduling*, has received surprisingly little attention, despite the large body of work on concurrency control and locking protocols [15, 46, 7, 68, 18, 40, 50, 54, 23]. In fact, almost all existing DBMSs[1] rely on variants of the first-in, first-out (FIFO) strategy, which grants (all) compatible lock requests based on their arrival time in the queue [2, 3, 4, 5, 6]. In this paper, we carefully study the problem of lock scheduling and show that it has significant ramifications on overall performance of a DBMS.

Related Work — There is a long history of research on scheduling in a general context [25, 42, 69, 70, 64, 41, 35, 62], whereby a set of jobs is to be scheduled on a set of processors such that a goal function is minimized, e.g., the sum of (weighted) completion times [64, 41, 39] or the variance of the completion or wait times [14, 17, 76, 49, 28]. There is also work on scheduling in a real-time database context [75, 40, 7, 36, 74], where the goal is to minimize the total tardiness or the number of transactions missing their deadlines.

In this paper, we address the problem of lock scheduling in a transactional context, where jobs are transactions and processors are locks, and the scheduling decision is about which locks to grant to which transactions. However, our transactional context makes this problem quite different than the well-studied variants of the scheduling problem. First, unlike generic scheduling problems, where at most one job can be scheduled on each processor, a lock may be held in either exclusive or shared modes. The fact that transactions can sometimes share the same resources (i.e., shared locks) significantly complicates the problem (see Section 2.4). Moreover, once a lock is granted to a transaction, the same transaction may later request another lock (as opposed to jobs requesting all of their needed resources upfront). Finally, in the scheduling literature, the execution time of each job is assumed to be known upon its arrival [52, 70, 14, 76], whereas the execution time of a transaction is often unknown *a priori*.

Although there are scheduling algorithms designed for real-time databases [55, 71, 77, 13], they are not applica-

Proceedings of the VLDB Endowment, Vol. 11, No. 5

DOI: 10.1145/3177732.3177740

[1]The only exceptions are MySQL and MariaDB, which have recently adopted our Variance-Aware Transaction Scheduling (VATS) [44] (see Section 7).

ble in a general DBMS context. For example, real-time settings assume that each transaction comes with a deadline, whereas most database workloads do not have explicit deadlines. Instead, most workloads wish to minimize latency or maximize throughput.

Challenges — Several aspects of lock scheduling make it a uniquely challenging problem, particularly under the performance considerations of a real-world DBMS.

1. **An online problem.** At the time of granting a lock to a transaction, we do not know when the lock will be released, since the transaction's execution time will only be known once it is finished.
2. **Dependencies.** In a DBMS, there are dependencies among concurrent transactions when one is waiting for a lock held by another. In practice, these dependencies can be quite complex, as each transaction can hold locks on several objects and several transactions can hold shared locks on the same object.
3. **Non-uniform access patterns.** Not all objects in the database are equally popular. Also, different transaction types might each have a different access pattern.
4. **Multiple locking modes.** The possibility of granting a lock to one writer exclusively or to multiple readers is a source of great complexity (see Section 2.4).

Contributions — In this paper, to the best of our knowledge, we present the first formal study of lock scheduling problem with a goal of minimizing transaction latencies in a DBMS context. Furthermore, we propose a contention-aware transaction scheduling algorithm, which captures the contention and the dependencies among concurrent transactions. The key insight is that a transaction blocking many others should be scheduled earlier. We carefully study the difficulty of lock scheduling and the optimality of our algorithm. Most importantly, we show that our results are not merely theoretical, but lead to dramatic speedups in a real-world DBMS. Despite decades of research on all aspects of transaction processing, lock scheduling seems to have gone unnoticed, to the extent that nearly all DBMSs still use FIFO. Our ultimate hope is that our results draw attention to the importance of lock scheduling on the overall performance of a transactional system.

In summary, we make the following contributions:

1. We propose a contention-aware lock scheduling algorithm, called Largest-Dependency-Set-First (LDSF). We prove that, in the absence of shared locks, LDSF is optimal in terms of the expected mean latency (Theorem 2). With shared locks, we prove that LDSF is a constant factor approximation of the optimal scheduling under certain regularity constraints (Theorem 3).
2. We propose the idea of granting only *some* of the shared lock requests on an object (as opposed to granting them *all*). We study the difficulty of the scheduling problem under this setting (Theorem 5), and propose another algorithm, called bLDSF (batched Largest-Dependency-Set-First), which improves upon LDSF in this setting. We prove that bLDSF is also a constant factor approximation of the optimal scheduling (Theorem 6).
3. In addition to our theoretical analysis, we use a real-world DBMS and extensive experiments to empirically evaluate our algorithms on the TPC-C benchmark, as well as a microbenchmark. Our results confirm that, compared to the commonly-used FIFO strategy, LDSF and bLDSF reduce mean transaction latencies by up to 300x and 290x, respectively. They also increase throughput by up to 6.5x and 5.5x. As a result, LDSF (which is simpler than bLDSF) has already been adopted as the default scheduling algorithm in MySQL [1] as of 8.0.3+.

2. PROBLEM STATEMENT

In this section, we first describe our problem setting and define dependency graphs. We then formally state the lock scheduling problem.

2.1 Background: Locking Protocols

Locks are the most commonly used mechanism for ensuring consistency when a set of shared objects are concurrently accessed by multiple transactions (or applications). In a locking system, there are two main types of locks: shared locks and exclusive locks. Before a transaction can read an object (e.g., a row), it must first acquire a shared lock (a.k.a. read lock) on that object. Likewise, before a transaction can write to or update an object, it must acquire an exclusive lock (a.k.a. write lock) on that object. A shared lock can be granted on an object as long as no exclusive locks are currently held on that object. However, an exclusive lock on an object can be granted only if there are no other locks currently held on that object. We focus on the strict 2-phase locking (strict 2PL) protocol: once a lock is granted to a transaction, it is held until that transaction ends. Once a transaction finishes execution (i.e., it commits or gets aborted), it releases all of its locks.

2.2 Dependency Graph

Given the set T of transactions currently in the system, and the set O of objects in the database, we define the dependency graph of the system as an edge-labeled graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$. The vertices of the graph $\mathcal{V} = T \cup O$ consist of the current transactions and objects. The edges of the graph $\mathcal{E} \subseteq T \times O \cup O \times T$ describe the locking relationships among the objects and transactions. Specifically, for transaction $t \in T$ and object $o \in O$,

- $(t, o) \in \mathcal{E}$ if t is waiting for a lock on o;
- $(o, t) \in \mathcal{E}$ if t already holds a lock on o.

The label $\mathcal{L} : \mathcal{E} \to \{S, X\}$ indicates the lock type:

- $\mathcal{L}(t, o) = X$ if t is waiting for an exclusive lock on o;
- $\mathcal{L}(t, o) = S$ if t is waiting for a shared lock on o;
- $\mathcal{L}(o, t) = X$ if t already holds an exclusive lock on o;
- $\mathcal{L}(o, t) = S$ if t already holds a shared lock on o.

We assume that deadlocks are rare and are handled by an external process (e.g., a deadlock detection and resolution module). Thus, for simplicity, we assume that the dependency graph $\mathcal{G}$ is always a directed acyclic graph (DAG).

2.3 Lock Scheduling

A lock scheduler makes decisions about which transactions are granted locks upon one or both of the following events: (i) when a transaction requests a lock, and (ii) when a lock is released by a transaction.[2] Let $\mathbb{G}$ be the set of all possible

[2]These are the only situations in which the dependency graph changes. If a scheduler grants locks at other times, the same decision could have been made upon the previous

Notation	Description
T	the set of transactions in the system
O	the set of objects in the database
$\mathcal{G}$	the dependency graph of the system
$\mathcal{V}$	vertices in the dependency graph
$\mathcal{E}$	edges in the dependency graph
$\mathcal{L}$	labels of the edges indicating the lock type
$\mathcal{A}$	a scheduling algorithm
$l_{\mathcal{A}}(t)$	the latency of transaction t under $\mathcal{A}$
$\bar{l}_{\mathcal{A}}(t)$	the expectation of $l_{\mathcal{A}}(t)$
$\bar{l}(\mathcal{A})$	the expected transaction latency under $\mathcal{A}$

Table 1: Table of Notations.

dependency graphs of the system. A scheduling algorithm $\mathcal{A} = (\mathcal{A}_{req}, \mathcal{A}_{rel})$ is a pair of functions $\mathcal{A}_{req}, \mathcal{A}_{rel} : \mathbb{G} \times O \times T \times \{S, X\} \to 2^T$. For example, when transaction t requests an exclusive lock on object o, $\mathcal{A}_{req}(\mathcal{G}, o, t, X)$ determines which of the transactions currently waiting for a lock on o (including t itself) should be granted their requested lock on o, given the dependency graph $\mathcal{G}$ of the system. (Note that the types of locks requested by transactions other than t are captured in $\mathcal{G}$.) Likewise, when transaction t releases a shared lock on object o, $\mathcal{A}_{rel}(\mathcal{G}, o, t, S)$ determines which of the transactions currently waiting for a lock on o should be granted their requested lock, given the dependency graph $\mathcal{G}$. When all transactions holding a lock on an object o release the lock, we say that o has *become available*. When the lock request of a transaction t is granted, we say that t is scheduled.

Since the execution time of each transaction is typically unknown in advance, we model their execution time using a random variable with expectation R. Given a particular scheduling algorithm $\mathcal{A}$, we define the latency of a transaction t, denoted by $l_{\mathcal{A}}(t)$, as its execution time plus the total time it has been blocked waiting for various locks. Since $l_{\mathcal{A}}(t)$ is a random variable, we denote its expectation as $\bar{l}_{\mathcal{A}}(t)$. We use $\bar{l}(\mathcal{A})$ to denote the expected transaction latency under algorithm $\mathcal{A}$, which is defined as the average of the expected latencies of all transactions in the system, i.e., $\bar{l}(\mathcal{A}) = \frac{1}{|T|} \sum_{t \in T} \bar{l}_{\mathcal{A}}(t)$.

Our goal is to find a lock scheduling algorithm under which the expected transaction latency is minimized. To ensure consistency and isolation, in most database systems $\mathcal{A}_{req}$ simply grants a lock to the requesting transaction only when (i) no lock is held on the object, or (ii) the currently held lock and the requested lock are compatible and no transaction in the queue has an incompatible lock request. This choice of $\mathcal{A}_{req}$ also ensures that transactions requesting exclusive locks are not starved. The key challenge in lock scheduling, then, is choosing an $\mathcal{A}_{rel}$ such that the expected transaction latency is minimized.

2.4 NP-Hardness

Minimizing the expected transaction latency under the scheduling algorithm is, in general, an NP-hard problem. Intuitively, the hardness is due to the presence of shared locks, which cause the system's dependency graph to be a DAG, but not necessarily a tree.

event, i.e., a transaction was unnecessarily blocked. A lock scheduler is thus an event-driven scheduler.

THEOREM 1. *Given a dependency graph $\mathcal{G}$, when a transaction t releases a lock (S or X) on object o, it is NP-hard to determine which pending lock requests to grant, in order to minimize the expected transaction latency. The result holds even if all transactions have the same execution time, and no transaction requests additional locks in the future.*[3]

Given the NP-hardness of the problem in general, in the rest of this paper, we propose algorithms that guarantee a constant-factor approximation of the optimal scheduling in terms of the expected transaction latency.

3. CONTENTION-AWARE SCHEDULING

We define contention-aware scheduling as any algorithm that prioritizes transactions based on their impact on the overall contention in the system. First, we study several heuristics for comparing the contribution of different transactions to the overall contention, and illustrate their shortcomings through intuitive examples. We then propose a particular contention-aware scheduling that formally quantifies this contribution, and guarantees a constant-factor approximation of the optimal scheduling when shared locks are not held by too many transactions. (Later, in Section 4, we generalize this algorithm for situations where this assumption does not hold.)

3.1 Capturing Contention

The degree of contention in a database system is directly related to the number of transactions concurrently requesting conflicting locks on the same objects.

For example, a transaction holding an exclusive lock on a popular object will naturally block many other transactions requesting a lock on that same object. If such a transaction is itself blocked (e.g., waiting for a lock on a different object), it will negatively affect the latency of many transactions, increasing overall contention in the system. Thus, our goal in contention-aware scheduling is to determine which transactions have a more important role in reducing the overall contention in the system, so that they can be given higher priority when granting a lock. Next, we discuss heuristics for measuring the priority of a transaction in reducing the overall contention.

Number of locks held — The simplest criterion for prioritizing transactions is the number of locks they currently hold. We refer to this heuristic as Most Locks First (MLF). The intuition is that a transaction with more locks is more likely to block other transactions in the system. However, this approach does not account for the popularity of objects in the system. In other words, a transaction might be holding many locks but on unpopular objects, which are unlikely to be requested by other transactions. Prioritizing such a transaction will not necessarily reduce contention in the system. Figure 1 demonstrates an example where transaction t_1 holds the most number of locks, but on unpopular objects. It is therefore better to keep t_1 waiting and instead schedule t_2 first, which holds fewer but more popular locks.

Number of locks that block other transactions — An improvement over the previous criterion is to only count those locks that have at least one transaction waiting on them. This approach disregards transactions that hold many

[3]All missing proofs can be found in our technical report [72].

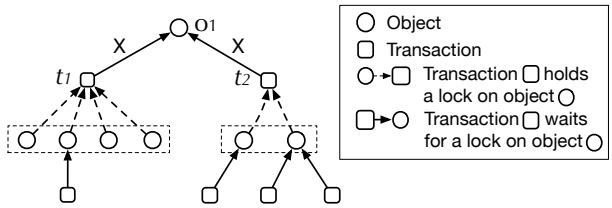

Figure 1: Transaction t_1 holds the greatest number of locks, but many of them on unpopular objects.

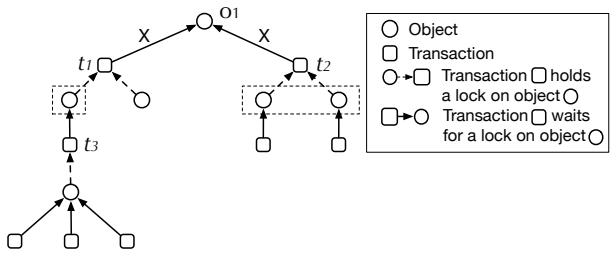

Figure 2: Transaction t_2 holds two locks that are waited on by other transactions. Although only one of t_1's locks is blocking other transactions, the blocked transaction (i.e., t_3) is itself blocking three others.

locks, but on these locks no other transactions are waiting. We call this heuristic Most Blocking Locks First (MBLF). The issue with this criterion is that it treats all blocked transactions as the same, even if they contribute unequally to the overall contention. Figure 2 shows an example in which the scheduler must decide between transactions t_1 and t_2 when the object o_1 becomes available. Here, this criterion would choose t_2, which currently holds two locks, each at least blocking one other transaction. However, although t_1 holds only one blocking lock, it is blocking t_3 which itself is blocking three other transactions. Thus, by scheduling t_2 first, t_3 and its three subsequent transactions will remain blocked in the system for a longer period of time than if t_1 had been scheduled first.

Depth of the dependency subgraph — A more sophisticated criterion is the depth of a transaction's dependency subgraph. For a transaction t, this is defined as the subgraph of the dependency graph comprised of all vertices that can reach t (and all edges between such vertices). The depth of t's dependency subgraph is characterized by the number of transactions on the longest path in the subgraph that ends in t. We refer to this heuristic as Deepest Dependency First (DDF). Figure 3 shows an example, where the depth of the dependency subgraph of transaction t_1 is 3, while that of transaction t_2 is only 2. Thus, based on this criterion, the exclusive lock on object o_1 should be granted to t_1. The idea behind this heuristic is that a longer path indicates a larger number of transactions sequentially blocked. Thus, to unblock such transactions sooner, the scheduling algorithm must start with a transaction with deeper dependency graph. However, considering only the depth of this subgraph can limit the overall degree of concurrency in the system. For example, in Figure 3, if the exclusive lock on o_1 is granted to t_1 , upon its completion only one transaction in its dependency subgraph will be unblocked. On the other hand, if the lock is granted to t_2, upon its completion two other transactions in its dependency subgraph will be unblocked, which can run concurrently.

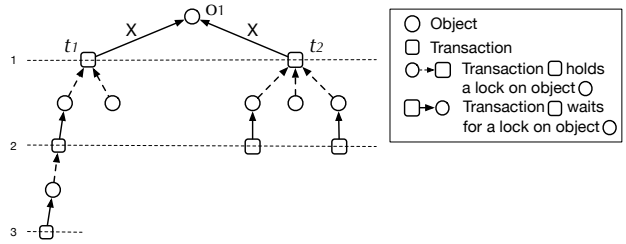

Figure 3: Transaction t_1 has a deeper dependency subgraph, but granting the lock to t_2 will unblock more transactions which can run concurrently.

Later, in Section 6.4, we empirically evaluate these heuristics. While none of these heuristics alone are able to guarantee an optimal lock scheduling strategy, they offer valuable insight in understanding the relationship between scheduling and overall contention. In particular, the first two heuristics focus on what we call *horizontal contention*, whereby a transaction holds locks on many objects directly needed by other transactions. In contrast, the third heuristic focuses on reducing *vertical contention*, whereby a chain of dependencies causes a series of transactions to block each other. Next, we present an algorithm which is capable of resolving both horizontal and vertical aspects of contention.

3.2 Largest-Dependency-Set-First

In this section, we propose an algorithm, called Largest-Dependency-Set-First (LDSF), which provides formal guarantees on the expected mean latency.

Consider two transactions t_1 and t_2 in the system. If there is a path from t_1 to t_2 in the dependency graph, we say that t_1 is dependent on t_2 (i.e., t_1 depends on t_2's completion/abortion for at least one of its required locks). We define the dependency set of t, denoted by $g(t)$, as the set of all transactions that are dependent on t (i.e., the set of transactions in t's dependency subgraph). Our LDSF algorithm uses the size of the dependency sets of different transactions to decide which one(s) to schedule first. For example, in Figure 4, there are five transactions in the dependency set of transaction t_1 (including t_1 itself) while there are four transactions in t_2's dependency set. Thus, in a situation where both t_1 and t_2 have requested an exclusive lock on object o_1, LDSF grants the lock to t_1 (instead of t_2) as soon as o_1 becomes available.

Now, we can formally present our LDSF algorithm. Suppose an object o becomes available (i.e., all previous locks on o are released), and there are $m + n$ transactions currently waiting for a lock on o: m transactions $t_1^i, t_2^i, \cdots, t_m^i$ are requesting a shared lock o, and n transactions $t_1^x, t_2^x, \cdots, t_n^x$ are requesting an exclusive lock on object o. Our LDSF algorithm defines the priority of each transaction t_i^x requesting an exclusive lock as the size of its dependency set, $|g(t_i^x)|$. However, LDSF treats all transactions requesting a shared lock on o, namely $t_1^i, t_2^i, \cdots, t_m^i$, as a single transaction—if LDSF decides to grant a shared lock, it will be granted to all of them. The priority of the shared lock requests is thus defined as the size of the union of their dependency sets, $\left|\bigcup_{i=1}^{m} g(t_i^i)\right|$. LDSF then finds the transaction $\hat{t^x}$ with the highest priority among $t_1^x, t_2^x, \cdots, t_n^x$. If $\hat{t^x}$'s priority is higher than the collective priority of the transactions requesting a shared lock, LDSF grants the exclusive lock to $\hat{t^x}$. Otherwise, a shared lock is granted to all transactions

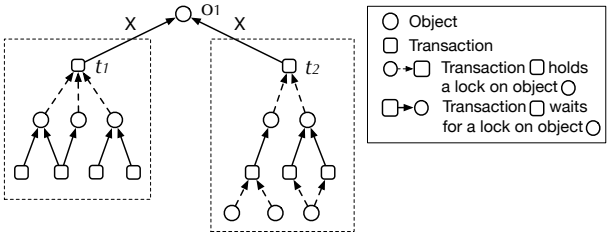

Figure 4: Lock scheduling based on the size of the dependency sets.

Input : The dependency graph of the system $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$, transaction t, object o, label $L \in \{X, S\}$
// meaning t has just released a lock of type L on o
Output: The set of transactions whose requested lock on o should be granted

if *there are other transactions still holding a lock on* o **then**
  return $\emptyset$;
Obtain the set of transactions waiting for a shared lock on o, $T^i \leftarrow \{t^i \in \mathcal{V} : (t^i, o) \in \mathcal{E} \text{ and } \mathcal{L}(t^i, o) = S\} = \{t^i_1, t^i_2, \cdots, t^i_m\}$;
Obtain the set of transactions waiting for an exclusive lock on o, $T^x \leftarrow \{t^x \in \mathcal{V} : (t^x, o) \in \mathcal{E} \text{ and } \mathcal{L}(t^x, o) = X\} = \{t^x_1, t^x_2, \cdots, t^x_n\}$;
Let $\tau(T^i) = \left|\bigcup_{i=1}^{n} g(t^i_i)\right|$;
Find a transaction $t^{\hat{x}} \in W$ s.t. $|g(t^{\hat{x}})| = \max_{t^x_i \in T^x} \left|g(t^x_i)\right|$;
if $\tau(T^i) < \left|g(t^{\hat{x}})\right|$ **then**
  return T^i;
else
  return $\{t^{\hat{x}}\}$;

Algorithm 1: *Largest-Dependency-Set-First Algorithm*

$t^i_1, t^i_2, \cdots, t^i_m$. The pseudo-code of the LDSF algorithm is provided in Algorithm 1.

Analysis — We do not make any assumptions about the future behavior of a transaction, as they may request various locks throughout their lifetime. Furthermore, since we cannot predict new transactions arriving in the future, in our analysis, we only consider the transactions that are already in the system. Since the system does not know the execution time of a transaction *a priori*, we model the execution time of each transaction as a memoryless random variable. That is, the time a transaction has already spent in execution does not necessarily reveal any information about the transaction's remaining execution time. We denote the remaining execution time as a random variable R with expectation $\bar{R}$. We also assume that the execution time of a transaction is not affected by the scheduling.[4] Transactions whose behavior depends on the actual wall-clock time (e.g., stop if run before 2pm, otherwise run for a long time) are also excluded from our discussion.

We first study a simplified scenario in which there are only exclusive locks in the system (we relax this assumption in Theorem 3). The following theorem states that LDSF minimizes the expected latency in this scenario.

THEOREM 2. *When there are only exclusive locks in the system, the LDSF algorithm is the optimal scheduling algorithm in terms of the expected latency.*

[4]For example, scheduling causes context switches, which may affect performance. For simplicity, in our formal analysis, we assume that their overall effect is not significant.

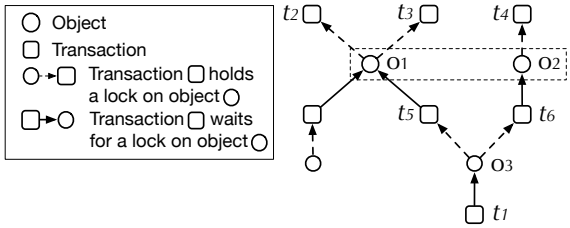

Figure 5: The critical objects of t_1 are o_1 and o_2, as they are locked by transactions t_2 and t_3. Note that, although o_3 is reachable from t_1, it is not a critical object of t_1 since it is locked by transactions that are not currently running, i.e., t_5 and t_6 which themselves are waiting for other locks.

The intuition here is that if a transaction t_1 is dependent on t_2, any progress in the execution of t_2 can also be considered as t_1's progress since t_1 cannot receive its lock unless t_2 finishes execution. Thus, by granting the lock to the transaction with the largest dependency set, LDSF allows the most transactions to make progress toward completion.

However, this does not necessarily hold true with the existence of shared locks. Even if transaction t_1 is dependent on t_2, the execution of t_2 does not necessarily contribute to t_1's progress. Specifically, consider the set of all objects that are reachable from t_1 in the dependency graph, but are locked (shared or exclusively) by *currently running* transactions. We call these objects the *critical objects* of t_1, and denote them as $C(t_1)$.[5] For example, in Figure 5, we have $C(t_1) = \{o_1, o_2\}$. Note that not all transactions that hold a lock on a critical object of t_1 contribute to t_1's progress. Rather, only the transaction that releases the last lock on that critical object allows for the progress of t_1. In the example of Figure 5, t_2's execution does not contribute to t_1's progress, unless t_3 releases the lock before t_2.

Nonetheless, when the number of transactions waiting for each shared lock is bounded, LDSF is a constant-factor approximation of the optimal scheduler.

THEOREM 3. *Let the maximum number of critical objects for any transaction in the system be c. Assume that the number of transactions waiting for a shared lock on the same object is bounded by u. The LDSF algorithm is a $(c \cdot u)$-approximation of the optimal scheduling (among strategies that grant all shared locks simultaneously) in terms of the expected latency.*

4. SPLITTING SHARED LOCKS

In the LDSF algorithm, when a shared lock is granted, it is granted to all transactions waiting for it. In Section 4.1, we show why this may not be the best strategy. Then, in Section 4.2, we propose a modification to our LDSF algorithm, called bLDSF, which improves upon LDSF by exploiting the idea of not granting all shared locks simultaneously.

4.1 The Benefits and Challenges

As noted earlier, when the LDSF algorithm grants a shared lock, it grants the lock to all transactions waiting for it. However, this may not be the optimal strategy. In general, granting a larger number of shared locks on the same object increases the probability that at least one of them will take

[5]Note that the critical objects of a transaction may change throughout its lifetime.

a long time before releasing the lock. Until the last transaction completes and releases its lock, no exclusive locks can be granted on that object. In other words, the expected duration that the slowest transaction holds a shared lock grows with the number of transactions sharing the lock. This is the well-known problem of *stragglers* [21, 27, 31, 63, 79], which is exacerbated as the number of independent processes grows.

To illustrate this more formally, consider the following example. Suppose that a set of m transactions, $t_1, \cdots, t_m$, are sharing a shared lock. Let $R_1^{rem}, R_2^{rem}, \cdots, R_m^{rem}$ be a set of random variables representing the remaining times of these transactions. Then, the time needed before an exclusive lock can be granted on the same object is the remaining time of the the slowest transaction, denoted as $R_{\max,m}^{rem} = \max\{R_1^{rem}, \cdots, R_m^{rem}\}$, which itself is a random variable. Let $\bar{R}_{\max,m}^{rem}$ be the expectation of $R_{\max,m}^{rem}$. As long as the R_i^{rem}'s have non-zero variance[6] (i.e., $\sigma_i^2 > 0$), $\bar{R}_{\max,m}^{rem}$ strictly increases with m, as stated next.

LEMMA 4. *Suppose that $R_1^{rem}, R_2^{rem}, \cdots$ are random variables with the same range of values. If $\sigma_{k+1}^2 > 0$, then $\bar{R}_{\max,k}^{rem} < \bar{R}_{\max,k+1}^{rem}$ for $1 \le k < m$.*

We define the delay factor as $f(m) = \dfrac{\bar{R}_{\max,m}^{rem}}{\bar{R}^{rem}}$. According to Lemma 4, $f(m)$ is strictly monotonically increasing with respect to m. The exact formula for $f(m)$ will depend on the specific distribution of R_i's. For example, if R_i's are exponentially distributed (i.e., a memoryless distribution) with mean $\bar{R}$, then their CDF is given by $F(x) = 1 - e^{-x/\bar{R}^{rem}}$. Then, $f(m)$ can be computed as $f(m) = \sum_{i=1}^{m} \dfrac{1}{i}$ However, regardless of the distribution of the latencies, $f(m)$ is guaranteed to satisfy the following three properties:

C1. $f(1) = 1$;

C2. $f(m) < f(m+1)$;

C3. $f(m) \le m$.

The first property is trivial: granting the lock to only one transaction at a time does not incur any delays. The second property is based on Lemma 4. The third is based on the fact that sharing a lock between a group of m transactions cannot be slower than granting the lock to them one after another and sequentially.

Since granting a shared lock to more transactions can delay the exclusive lock requests, it is conceivable that granting a shared lock to only a subset of the transactions waiting for it might reduce the overall latency in the system. Intuitively, when many transactions are waiting for the same shared lock, it would be better to grant the shared lock only to a few that have a higher priority (i.e., a larger dependency set), and leave the rest until the next time. This strategy can therefore reduce the time that other transactions have to wait for an exclusive lock, as illustrated in Figure 6. However, lock scheduling in this situation becomes extremely difficult. We have the following negative result.

THEOREM 5. *Let $\mathbb{A}_{\neg f}$ be the set of scheduling algorithms that do not use the knowledge of the delay factor $f(k)$ in their decisions. For any algorithm $\mathcal{A}_{\neg f} \in \mathbb{A}_{\neg f}$, there exists an algorithm $\mathcal{A}$, such that $\dfrac{\bar{w}(\mathcal{A}_{\neg f})}{\bar{w}(\mathcal{A})} = \omega(1)$ for some delay factor $f(k)$.*

[6]This assumption holds unless all instances of a transaction type take exactly the same time, which is unlikely.

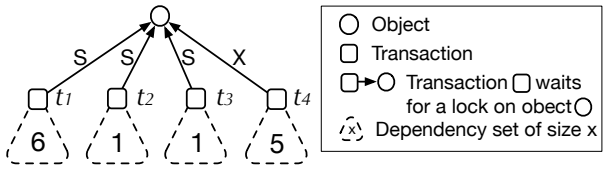

Figure 6: Assume that $f(2) = 1.5$ and $f(3) = 2$. If we first grant a shared lock to all of t_1, t_2, and t_3, all transactions in t_4's dependency set will wait for at least $2\bar{R}$. The total wait time will be $10\bar{R}$. However, if we only grant t_1's lock, then t_4's lock, and then grant t_2's and t_3's locks together, the transactions in t_4's dependency set will only wait $\bar{R}$, while those in t_2's and t_3's dependency sets will wait $2\bar{R}$. Thus, the total wait time in this case will be only $9\bar{R}$.

According to this theorem, any algorithm that does not rely on knowing the delay factor is not competitive: it performs arbitrarily poor, compared to the optimal scheduling. Thus, in the next section, we take the delay factor $f(k)$ as an input, and propose an algorithm that adopts the idea of granting shared locks only to a subset of the transactions requesting it. We also discuss the criteria for choosing delay factors that can yield good performance in practice.

4.2 The bLDSF Algorithm

In this section, we present a simple algorithm, called bLDSF, which inherits the intuition behind the LDSF algorithm, but also exploits the idea that a shared lock does not have to be granted to all transactions waiting for it.

While LDSF measures the progress enabled by different scheduling decisions, our bLDSF algorithm measures the *speed of progress*. If a transaction t^x waiting for an exclusive lock is scheduled, $|g(t^x)|$ transactions will make progress over the next $\bar{R}$ (expected) units of time. Thus, the speed of progress can be measured as $\dfrac{|g(t^x)|}{\bar{R}}$. On the other hand, by scheduling a batch of transactions $t_1^i, t_2^i, \cdots, t_k^i$ waiting for a shared lock together, $|\bigcup_{i=1}^{k} g(t_i^i)|$ transactions will make progress over the next $f(k) \cdot \bar{R}$ units of time. The speed of progress can then be measured as $\dfrac{|\bigcup_{i=1}^{k} g(t_i^i)|}{f(k)\bar{R}}$.

The bLDSF algorithm works as follows. First, it finds the transaction waiting for an exclusive lock with the largest dependency set, denoted as $\hat{t^x}$. Denote the size of its dependency set as $p = \left|g(\hat{t^x})\right|$. Then, bLDSF finds the batch of transactions, $\hat{t_1^i}, \hat{t_2^i}, \cdots, \hat{t_k^i}$, waiting for a shared lock such that $q = \dfrac{|\bigcup_{i=1}^{k} g(\hat{t_i^i})|}{f(k)}$ is maximized. When $q < p$, the system will make faster progress if $\hat{t^x}$ is scheduled first, in which case bLDSF will grant an exclusive lock to $\hat{t^x}$. Conversely, when $q > p$, the system will make faster progress if the batch of $\hat{t_1^i}, \hat{t_2^i}, \cdots, \hat{t_k^i}$ is scheduled first, in which case bLDSF will grant shared locks to $\hat{t_1^i}, \hat{t_2^i}, \cdots, \hat{t_k^i}$ simultaneously. When $q = p$, the speed of progress in the system will be the same under both scheduling decisions. In this case, bLDSF grants shared locks to the batch, in order to increase the overall degree of concurrency in the system. The pseudocode for bLDSF is provided in Algorithm 2.

We show that, when the number of transactions waiting for shared locks on the same object is bounded, the bLDSF algorithm is a constant factor approximation of the optimal scheduling algorithm in terms of the expected wait time.

Input : The dependency graph of the system $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$, transaction t, object o, label $L \in \{X, S\}$
// meaning t has just released a lock of type L on o
Output: The set of transactions whose requested lock on o should be granted

if *there are other transactions still holding a lock on* o **then**
  return $\emptyset$;
Obtain the set of transactions waiting for a shared lock on o, $T^i \leftarrow \{t^i \in \mathcal{V} : (t^i, o) \in \mathcal{E} \text{ and } \mathcal{L}(t^i, o) = S\} = \{t^i_1, t^i_2, \cdots, t^i_m\}$;
Obtain the set of transactions waiting for an exclusive lock on o, $T^x \leftarrow \{t^x \in \mathcal{V} : (t^x, o) \in \mathcal{E} \text{ and } \mathcal{L}(t^x, o) = X\} = \{t^x_1, t^x_2, \cdots, t^x_n\}$;
Let $t^{\hat{i}}_1, t^{\hat{i}}_2, \cdots, t^{\hat{i}}_k$ be the set of transactions in T^i such that $\frac{|\bigcup_{i=1}^{k} g(t^{\hat{i}}_i)|}{f(k)}$ is maximized ;
Let $t^{\hat{x}}$ be the transaction in T^x with the largest dependency set;
if $\left|g(t^{\hat{x}})\right| \cdot f(k) \leq \left|\bigcup_{i=1}^{k} g(t^{\hat{i}}_i)\right|$ **then**
  return $\{t^{\hat{i}}_1, t^{\hat{i}}_2, \cdots, t^{\hat{i}}_k\}$;
else
  return $t^{\hat{x}}$;

Algorithm 2: *The bLDSF Algorithm*

THEOREM 6. *Let the maximum number of critical objects for any transaction in the system be c. Assume that the number of transactions waiting for shared locks on the same object is bounded by v. Then, given a delay factor of $f(k)$, the bLDSF algorithm is an h-approximation of the optimal scheduling algorithm in terms of the expected wait time, where $h = cv^2 \cdot f(v)$.*

Unlike the LDSF algorithm, bLDSF requires a delay factor for its analysis. However, since the remaining times of transactions can be modeled as random variables, the exact form of the delay factor $f(k)$ will also depend on the distribution of these random variables. For example, the delay factor for exponential random variables is $f(k) = O(\log k)$ [22], for geometric random variables is $f(k) = O(\log k)$ [29], for Gaussian random variables is $f(k) = O(\sqrt{\log k})$ [47], and for power law random variables with exponent 3 is $f(k) = \sqrt{k}$.

In Section 6.7, we empirically show that bLDSF's performance is not sensitive to the specific choice of the delay factor, as long as it is a sub-linear function that grows monotonically with k (conditions C1, C2, and C3 from Section 4.1). This is because, when the batch size is small, the difference between all sub-linear functions is also small. For example, when $b = 10$, $\sqrt{b} \approx 3.16$ and $\log_2(1 + b) \approx 3.46$, leading to similar scheduling decisions. Even though $\sqrt{\log_2(1 + b)} \approx 1.86$ is smaller than the other two, it can still capture condition C2 quite well.

4.3 Discussion

In our analysis, we have assumed no additional information regarding a transaction's remaining execution time, or its lock access pattern. However, with the recent progress on incorporating machine learning models into DBMS technology [58, 11], one might be able to predict transaction latencies [78, 59] in the near future. When such information is available, a lock scheduling algorithm could take that into account when maximizing the *speed of progress*: a transaction that will take longer should be given less priority. The priority of a transaction would then be the size of its dependency set divided by its estimated execution time. Likewise, a transaction performing a table scan will request a large number of locks, and will not make any progress until all of its locks can be granted. Thus, knowing a transaction's lock pattern in advance would also be beneficial. We leave such extensions of our algorithms (e.g., to hybrid workloads [60]) to future work.

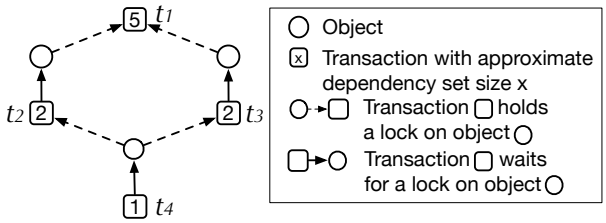

Figure 7: The effective size of t_1's dependency set is 5. But its exact size is only 4.

5. IMPLEMENTATION

We have implemented our scheduling algorithm in MySQL. Similar to all major DBMSs, the default lock scheduling policy in MySQL was FIFO.[7] Specifically, all pending lock requests on an object are placed in a queue. A lock request is granted immediately upon its arrival only if one of these two conditions holds: (i) there are no other locks currently held on the object, or (ii) the requested lock type is compatible with all of the locks currently held on the object and there are no incompatible requests ahead of it waiting in the queue. Similarly, whenever a lock is released on an object, MySQL's scheduler scans the entire queue from beginning to the end. It grants any waiting requests as long as one of these conditions holds. As soon as the scheduler encounters the first lock request that cannot be granted, it stops scanning the rest of the queue.

One challenge in implementing LDSF and bLDSF is keeping track of the sizes of the dependency sets. Exact calculation would require either (i) searching down the reverse edges in the dependency graph in real-time, whenever a scheduling decision is to be made, or (ii) storing the dependency sets for all transactions and maintaining them each time a transaction is blocked or a lock is granted. Both options are relatively costly. Therefore, in our implementation, we rely on an approximation of the sizes of the dependency sets, rather than computing their exact values. When a transaction t holds no locks that block other transactions, $|g(t)| = 1$. Otherwise, let T_t be the set of transactions waiting for an object currently held by transaction t. Then, $|g(t)| \approx \sum_{t' \in T_t} |g(t')| + 1$. The reason this method is only an approximation of $|g(t)|$ is that the dependency graph is a DAG (but not necessarily a tree), which means the dependency sets of different transactions may overlap. Figure 7 illustrates an example, where the dependency set of t_1 is $\{t_1, t_2, t_3, t_4\}$ and is therefore of size 4. However, its effective size is calculated as one plus the sum of the effective sizes of t_2 and t_3's dependency sets, resulting in 5. To ensure that transactions appearing on multiple paths will not be updated multiple times, we also keep track of those that have already been updated.

Another implementation challenge lies in the difficulty of finding the desired batch of transactions in bLDSF. Calculating the size of the union of several dependency sets

[7]Now, our LDSF algorithm is the default (MySQL 8.0.3+).

requires detailed information about the elements in each dependency set (since the dependency sets may overlap due to shared locks). Therefore, we rely on the following approximation in our implementation. We first sort all transactions waiting for a shared lock in the decreasing order of their dependency set sizes. Then, for $k = 1, 2, \cdots$, we calculate the q value (see Section 4.2) for the first k transactions. Here, we approximate the size of the union of the dependency sets as the sum of their individual sizes. Let k^* be the k value that maximizes q. We then take the first k^* transactions as our batch, which we consider for granting a shared lock to.

In Section 6, we show that, despite using these approximations in our implementation, our algorithms remain quite effective in practice.

Starvation Avoidance — In MySQL's implementation of FIFO, when there is an exclusive lock request in the queue, it serves as a conceptual barrier: later requests for shared locks cannot be granted, even if they are compatible with the currently held locks on the object. This mechanism prevents starvation when using FIFO. In our algorithms, starvation is prevented using a similar mechanism. We place a barrier at the end of the current wait queue. Lock requests that arrive later are placed behind this barrier and are not considered for scheduling. In other words, the only requests that are considered are those that are ahead of the barrier. Once all such requests are granted, this barrier is lifted, and a new barrier is added to the end of the current queue, i.e., those requests that were previously behind a barrier are now ahead of one. This mechanism prevents a transaction with a small dependency set from waiting indefinitely behind an infinite stream of newly arrived transactions with larger dependency sets. An alternative strategy to avoid starvation is to simply add a fraction of the transaction's age to its dependency set size when making scheduling decisions. A third strategy is to replace a transaction's dependency set size with a sufficiently large number once its wait time has exceeded a certain timeout threshold.

Space Complexity — Given the approximation methods mentioned above, both LDSF and bLDSF only require maintaining the approximate size of the dependency set of each transaction. Therefore, the overall space overhead of our algorithms is only $O(|T|)$.

Time Complexity — In MySQL, all lock requests on an object (either granted or not) are stored in a linked list. Whenever a transaction releases a lock on the object, the scheduler scans this list for requests that are not granted yet. For each of these requests, the scheduler scans the list again to check compatibility with granted requests. If the request is found compatible with all existing locks, it is granted, and the scheduler checks the compatibility of the next request. Otherwise, the request is not granted, and the scheduler stops granting further locks. Let N be the number of lock requests on an object (either granted or not). Then, FIFO takes $O(N^2)$ time in the worst case. LDSF and bLDSF both use the same procedure as FIFO to find compatible requests that are not granted yet, which takes $O(N^2)$ time. For bLDSF, we also sort all transactions waiting for a shared lock by the size of their dependency sets, which takes $O(N \log N)$ time. Thus, the time complexity of LDSF and bLDSF is still $O(N^2)$.

6. EXPERIMENTS

Our experiments aim to answer several key questions:

- How do our scheduling algorithms (LDSF and bLDSF) affect the overall throughput of the system?
- How do our algorithms compare against FIFO (the default policy in nearly all databases) and VATS (recently adopted by MySQL), in terms of reducing average and tail transaction latencies?
- How do our scheduling algorithms compare against various heuristics?
- How much overhead do our algorithms incur, compared to the latency of a transaction?
- How does the effectiveness of our algorithms vary with different levels of contention?
- What is the impact of the choice of delay factor on the effectiveness of bLDSF?
- What is the impact of approximating the dependency sets (Section 5) on reducing the overhead?

In summary, our experiments show the following:

1. By resolving contention much more effectively than FIFO and VATS, bLDSF improves throughput by up to 6.5x (by 4.5x on average) over FIFO, and by up to 2x (1.5x on average) over VATS. (Section 6.2)
2. bLDSF can reduce mean transaction latencies by up to 300x and 80x (30x and 3.5x, on average) compared to FIFO and VATS, respectively. It also reduces the 99th percentile latency by up to 190x and 16x, compared to FIFO and VATS, respectively. (Section 6.3)
3. Both bLDSF and LDSF outperform various heuristics by 2.5x in terms of throughput, and by up to 100x (8x on avg.) in terms of transaction latency. (Section 6.4)
4. Our algorithms reduce queue length by reducing contention, and thus incur much less overhead than FIFO. However, their overhead is larger than VATS. (Section 6.5)
5. As the degree of contention rises in the system, bLDSF's improvement over both FIFO and VATS increases. (Section 6.6)
6. bLDSF is not sensitive to the specific choice of delay factor, as long as it is chosen to be an increasing and sublinear function. (Section 6.7)
7. Our approximation technique reduces scheduling overhead by up to 80x.

6.1 Experimental Setup

Hardware & Software — All experiments were performed using a 5 GB buffer pool on a Linux server with 16 Intel(R) Xeon(R) CPU E5-2450 processors and 2.10GHz cores. The clients were run on a separate machine, submitting transactions to MySQL 5.7 running on the server.

Methodology — We used the OLTP-Bench tool [24] to run the TPC-C workload. We also modified this tool to run a microbenchmark (explained below). OLTP-Bench generated transactions at a specified rate, and client threads issued these transactions to MySQL. The latency of each transaction was calculated as the time from when it was issued until it finished. In all experiments, we controlled the number of transactions issued per second within a safe range to prevent MySQL from falling into a thrashing regime. We also noticed that the number of deadlocks was negligible compared

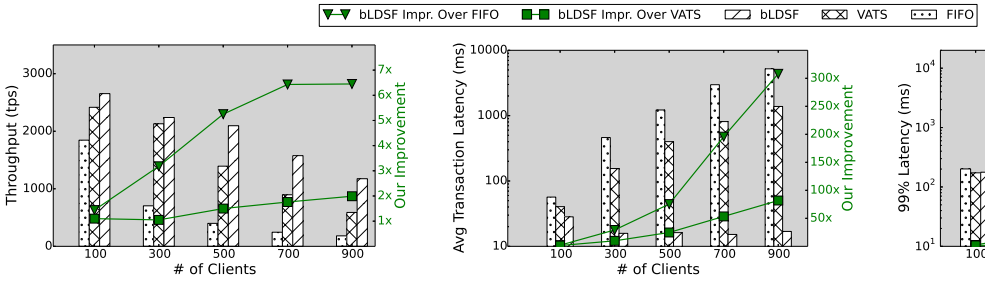

Figure 8: Throughput improvement with bLDSF (TPC-C).

Figure 9: Avg. latency improvement with bLDSF (under the same TPC-C transactions per second).

Figure 10: Tail latency improvement w/ bLDSF (under the same number of TPC-C transactions per second).

to the total number of transactions, across all experiments and algorithms.

TPC-C Workload — We used a 32-warehouse configuration for the TPC-C benchmark. To simulate a system with different levels of contention, we relied on changing the following two parameters: (i) number of clients, and (ii) number of submitted transactions per second (a.k.a. throughput). Each of our client threads issued a new transaction as soon as its previous transaction finished. Thus, by creating a specified number of client threads, we effectively controlled the number of in-flight transactions. To control the system throughput, we created client threads that issued transactions at a specific rate.

Microbenchmark — We created a microbenchmark for a more thorough evaluation of our algorithm under different degrees of contention. Specifically, we created a database with only one table that had 20,000 records in it. The clients would send transactions to the server, each comprised of 5 queries. Each query was randomly chosen to be either a "SELECT" query (acquiring a shared lock) or an "UPDATE" query (acquiring an exclusive lock). The records in the table were accessed by the queries according to a Zipfian distribution. To generate different levels of contention, we varied the following two parameters in our microbenchmark:

1. skew of the access pattern (the parameter θ of the Zipfian distribution)
2. fraction of exclusive locks (probability of "UPDATE" queries).

Baselines — We compared the performance of our bLDSF algorithm (with $f(k)$=$\log_2(1+k)$ as default) against the following baselines:

1. **First In First Out (FIFO).** FIFO is the default scheduler in MySQL and nearly all other DBMSs. When an object becomes available, FIFO grants the lock to the transaction that has waited the longest.
2. **Variance-Aware Transaction Scheduling (VATS).** This is the strategy proposed by Huang et al. [44]. When an object becomes available, VATS grants the lock to the eldest transaction in the queue.
3. **Largest Dependency Set First (LDSF).** This is the strategy described in Algorithm 1, which is equivalent to bLDSF with $b = \inf$, and $f(k) = 1$.
4. **Most Locks First (MLF).** When an object becomes available, grant a lock on it to the transaction that holds the most locks (introduced in Section 3.1).
5. **Most Blocking Locks First (MBLF).** When an object becomes available, grant a lock on it to the transaction that holds the most locks which block at least one other transaction (introduced in Section 3.1).
6. **Deepest Dependency First (DDF).** When an object becomes available, grant a lock on it to the transaction with the deepest dependency subgraph (Section 3.1).

For MLF, MBLF, and DDF, if a shared lock is granted, all shared locks on that object are granted. For LDSF and bLDSF, we use the barriers explained in Section 5 to prevent starvation. For FIFO and VATS, if a shared lock is granted, they continue to grant shared locks to other transactions waiting in the queue until they encounter an exclusive lock, at which point they stop granting more locks.

6.2 Throughput

We compared the system throughput when using FIFO and VATS versus bLDSF, given an equal number of clients (i.e., in-flight transactions). We varied the number of clients from 100 to 900. The results of this experiment for TPC-C are presented in Figure 8.

In both cases, the throughput dropped as the number of clients increased. This is expected, as more transactions in the system lead to more objects being locked. Thus, when a transaction requests a lock, it is more likely to be blocked. In other words, the number of transactions that can make progress decreases, which leads to a decrease in throughput.

However, the throughput decreased more rapidly when using FIFO or VATS than bLDSF. For example, when there were only 100 clients, bLDSF outperformed FIFO by only 1.4x and VATS by 1.1x. However, with 900 clients, bLDSF achieved 6.5x higher throughput than FIFO and 2x higher throughput than VATS. As discussed in Section 4.2, bLDSF always schedules transactions that maximize the speed of progress in the system. This is why it allows for more transactions to be processed in a certain amount of time.

6.3 Average and Tail Transaction Latency

We compared transaction latencies of FIFO, VATS, and bLDSF under an equal number of transactions per second (i.e, throughput). We varied the number of clients (and hence, the number of in-flight transactions) from 100 to 900 for FIFO and VATS, and then ran bLDSF at the same throughput as VATS, which is higher than the throughput of FIFO. This means that we compare bLDSF with FIFO at a higher throughput. The result is shown in Figure 9. Our bLDSF algorithm dramatically outperformed FIFO by a factor of up to 300x and VATS by 80x. This outstanding improvement confirms our Theorems 3 and 6, as our algorithm is designed to minimize average transaction latencies.

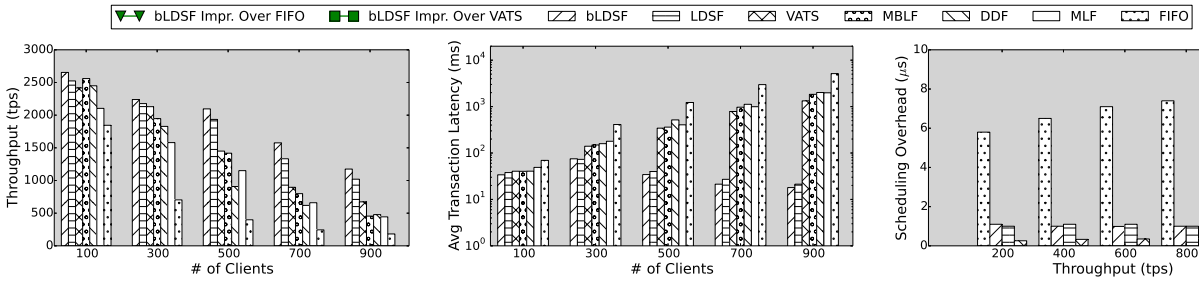

Figure 11: Maximum throughput under various algorithms (TPC-C).

Figure 12: Transaction latency under various algorithms (TPC-C).

Figure 13: Scheduling overhead of various algorithms (TPC-C).

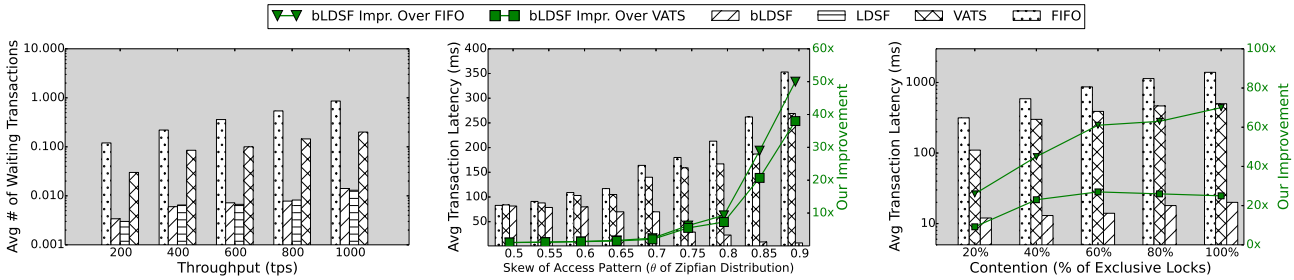

Figure 14: Average number of transactions waiting in the queue under various algorithms (TPC-C).

Figure 15: Average transaction latency for different degrees of skewness (microbenchmark).

Figure 16: Average latency for different numbers of exclusive locks (microbenchmark).

We also report the 99th percentile latencies in Figure 10. Here, bLDSF outperformed FIFO by up to 190x. Interestingly, bLDSF outperformed VATS too (by up to 16x), even though the latter is specifically designed to reduce tail latencies. This is because bLDSF causes all transactions to finish faster on average, and thus, those transactions waiting at the end of the queue will also wait less, leading to lower tail latencies.

6.4 Comparison with Other Heuristics

In this section, we report our comparison of both bLDSF and LDSF algorithms against the heuristic methods introduced in Section 3, i.e., MLF, MBLF, and DDF. Moreover, we compare our algorithms with VATS too.

First, we compared their throughput given an equal number of clients. We varied the number of clients from 100 to 900. The results are shown in Figure 11. LDSF and bLDSF achieve up to 2x and 2.5x improvement over the other heuristics in terms of throughput, respectively.

We also measured transaction latencies under an equal number of transactions per second (i.e, throughput). We varied the number of clients from 100 to 900 for the heuristics, and then ran bLDSF and LDSF at the maximum throughput achieved by any of the heuristics. For those heuristics which were not able to achieve this throughput, we compared our algorithms at a higher throughput than they achieved. The results are shown in Figure 12, indicating that MLF, MBLF, and DDF outperformed FIFO by almost 2.5x in terms of average latency, while our algorithms achieved up to 100x improvement over the best heuristics (MBLF with 900 transactions). Furthermore, bLDSF was better than LDSF by a small margin.

6.5 Scheduling Overhead

We also compared the overhead of our algorithms (LDSF and bLDSF) against both FIFO and VATS: the overhead of a scheduling algorithm is the time needed by the algorithm to *decide* which lock(s) to grant.

In this experiment, we fixed the number of clients to 100 while varying throughput from 200 to 1000. The result is shown in Figure 13. We can see that, although all three algorithms have the same time complexity in terms of the queue length (Section 5), ours resulted in much less overhead than FIFO because they led to much shorter queues for the same throughput. This is because our algorithms effectively resolve contention, and thus, reduce the number of waiting transactions in the queue. To illustrate this, we also measured the average number of waiting transactions whenever an object becomes available. As shown in Figure 14, this number was much smaller for LDSF and bLDSF. However, VATS incurred less overhead than LDSF and bLDSF, despite having longer queues. This is because VATS does not compute the sizes of the dependency sets.

6.6 Studying Different Levels of Contention

In this section, we study the impact of different levels of contention on the effectiveness of our bLDSF algorithm. Contention in a workload is a result of two factors: (i) skew in the data access pattern (e.g., popular tuples), and (ii) a large number of exclusive locks. There is more contention when the pattern is more skewed, as transactions will request a lock on the same records more often. Likewise, exclusive lock requests cause more contention, as they cannot be granted together and result in blocking more transactions. We studied the effectiveness of our algorithm under different degrees of contention by varying these two factors using our microbenchmark:

1. We fixed the fraction of exclusive locks to be 60% of all lock requests, and varied the θ parameter of the Zipfian distribution of our access distribution between 0.5 and 0.9 (larger θ, more skew).

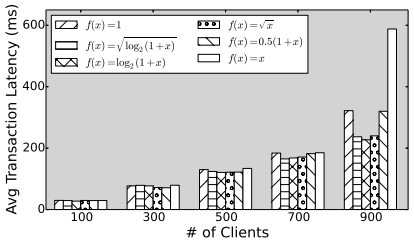

Figure 17: The impact of delay factor on average latency.

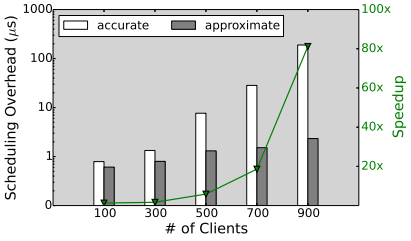

Figure 18: Scheduling overhead with and without our approximation heuristic for choosing a batch.

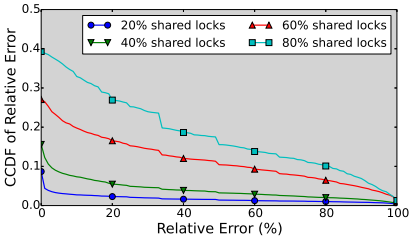

Figure 19: CCDF of the relative error of the approximation of the sizes of the dependency sets.

2. We fixed the θ parameter to be 0.8 and varied the probability of an "UPDATE" query in our microbenchmark between 20% and 100%. The larger this probability, the larger the fraction of exclusive locks.

First, we ran FIFO using 300 clients, and then ran both VATS and bLDSF at the same throughput as FIFO. The results of these experiments are shown in Figures 15 and 16.

Figure 15 shows that when there is no skew, there is no contention, and thus most queues are either empty or only have a single transaction waiting. Since there is no scheduling decision to be made in this situation, FIFO, VATS and bLDSF become equivalent and exhibit a similar performance. However, the gap between bLDSF and the other two algorithms widens as skew (and thereby contention) increases. For example, when the data access is highly skewed (θ = 0.9), bLDSF outperforms FIFO by more than 50x and VATS by 38x. Figure 16 reveals a similar trend: as more exclusive locks are requested, bLDSF achieves greater improvement. Specifically, when 20% of the lock requests are exclusive, bLDSF outperforms FIFO by 20x and VATS by 9x. However, when all the locks are exclusive, the improvement is even more dramatic, i.e., 70x over FIFO and 25x over VATS. Note that, although VATS guarantees optimality when there are only exclusive locks [44], it fails to account for transaction dependencies in its analysis (see Section 7 for a discussion of the assumptions made in VATS versus bLDSF). In summary, when there is no contention in the system, there are no scheduling decisions to be made, and all scheduling algorithms are equivalent. However, as contention rises, so does the need for better scheduling decisions, and so does the gap between bLDSF and other algorithms.

6.7 Choice of Delay Factor

To better understand the impact of delay factors on bLDSF, we experimented with several functions of different growth rates, ranging from the lower bound of all functions that satisfy conditions C1, C2, and C3 (i.e., $f(k) = 1$) to their upper bound (i.e., $f(k) = k$). Specifically, we used each of the following delay factors in our bLDSF algorithm, and measured the average transaction latency:

- $f_1(k) = 1$;
- $f_2(k) = \sqrt{\log_2(1+k)}$;
- $f_3(k) = \log_2(1+k)$;
- $f_4(k) = \sqrt{k}$;
- $f_5(k) = 0.5(1+k)$;
- $f_6(k) = k$.

The results are shown in Figure 17. We can see that all sublinear functions (i.e., f_2, f_3, and f_4) performed comparably, and that they performed better than the other functions. Understandably, f_1 did not perform well, as it did not satisfy condition C2 from Section 4.1. Functions f_5 and f_6 did not perform well either, since linear functions overestimate the delay. For example, two transactions running concurrently take less time than if they ran one after another.

6.8 Approximating Sizes of Dependency Sets

We studied the effectiveness of our approximation heuristic from Section 5 for choosing a batch of shared requests. Computing the optimal batch accurately was costly, and significantly lowered the throughput. However, we measured the scheduling overhead, and compared it to when we used an approximation. We ran TPC-C, and varied the number of clients from 100 to 900. As shown in Figure 18, our approximation reduced the scheduling overhead by up to 80x.

We also measured the error of our approximation technique for estimating the dependency set sizes—the deviation from the actual sizes of the dependency sets—for varying ratios of shared locks in the workload. Figure 19 shows the complementary cumulative distribution function (CCDF) of the relative error of approximating the dependency set sizes. The error grew with the ratio of shared locks; this was expected, as shared locks are the cause of error in our approximation. However, the errors remained within a reasonable range, e.g., even with 80% shared locks, we observed a 2-approximation of the exact sizes in 99% of the cases.

7. RELATED WORK

In short, the large body of work on traditional job scheduling is unsuitable in a database context due to the unique requirements of locking protocols deployed in databases. Although there is some work on lock scheduling for real-time databases, they aim at supporting explicit deadlines rather than minimizing the mean latency of transactions.

Job Scheduling — Outside the database community, there has been extensive research on scheduling problems in general. Here, the duration (and sometimes the weight and arrival time) of each task is known *a priori*, and a typical goal is to minimize (i) the sum of (weighted) completion times (SCT) [64, 41, 39], (ii) the latest completion time [20, 34, 67], (iii) the completion time variance (CTV) [14, 17, 76, 49], or even (iv) the waiting time variance (WTV) [28]. The offline SCT problem can be optimally solved using a Shortest-Weighted-Execution-Time approach, whereby jobs are scheduled in the non-decreasing order of their ratio of execution time to weight [70], if they all arrive at the same time. However, when the jobs arrive at different times, the scheduling becomes NP-hard [52].

None of these results are applicable to our setting, mainly because of their assumption that each processor/worker can

be used by only one job at a time, whereas in a database, locks can be held in shared and exclusive modes. Moreover, they assume the execution time of each job is known, which is not the case in a database (i.e., the database does not know when the application/user will commit and release its locks). Finally, with the exception of [64, 41], prior work on scheduling either assumes that all tasks are available at the beginning, or that their arrival time is known. In a database, however, such information is unavailable.

Dependency-based Scheduling — Scheduling tasks with dependencies among them has been studied for both single machines [69, 42] and multiprocessors [25, 26, 30, 61]. Here, each job only needs one processor and once scheduled, it will not be blocked again. However, in a database, a transaction can request many locks, and thus, can be blocked even after it is granted some locks.

Real-time Databases (RTDB) — There is some work on lock scheduling in the context of RTDBs, where transactions are scheduled to meet a set of user-given deadlines [75, 7, 55, 71, 77, 13, 36, 74, 73, 38, 33, 8, 53, 66, 19, 40]. It is shown that the First-In-First-Out (FIFO) policy performs poorly in this setting [33, 7, 8, 53], compared to the Earliest-Deadline-First policy [55, 71, 77], which is also used in practice [13].

Unfortunately, the work in this area is not applicable to general-purpose database systems. First, in an RTDB, each transaction comes with a pre-specified deadline, while in a general-purpose database such deadlines are not provided. Second, a key assumption in this line of work is that the execution time of each transaction is known in advance, whereas in a general database the execution time of a transaction is only known once it is finished. Finally, the scheduling goal in an RTDB is to minimize the total tardiness or the number of missed deadlines. In other words, as long as a transaction meets its deadline, RTDBs do not care whether it finishes right before the deadline or much earlier. In contrast, general databases aim to execute transactions as fast as possible.

Scheduling in Existing DBMS — For simplicity and fairness [12], the First-In-First-Out (FIFO) policy and its variants are the most widely adopted scheduling policies in many of today's databases [9], operating systems [16], and communication networks [51]. FIFO is the default lock scheduling policy in MySQL [3], MS SQL Server [6], Postgres [5], Teradata [4], and DB2 [2]. Despite its popularity, FIFO does not provide any guarantees in terms of average or percentile latencies. Huang et al. [44] propose a scheduling algorithm, called Variance-Aware Transaction Scheduling (VATS), which aims at minimizing the variance of transaction latencies, and its optimality holds only when there are no shared locks in the system. In contrast, we focus on minimizing mean latency, and allow for both shared and exclusive locks. In short, designing optimal lock scheduling algorithms for databases has remained an open problem.

VATS — Based on the findings of a new profiler, called VProfiler [45], we have previously proposed Variance-Aware Transaction Scheduling (VATS) [44]. VATS prioritizes transactions according to their arrival time in the system, as opposed to FIFO, which prioritizes them according to their arrival time in the current queue. Our prior work proves the optimality of VATS in terms of minimizing the L_p-norm of transaction latencies [44], when there are no shared locks. In contrast, the current paper proves the optimality of bLDSF in terms of minimizing mean latency. More importantly, our analysis of VATS uses a simplifying assumption that models the latency of a transaction t as $l(t) = A(t) + U(t) + R \cdot (N(t) + 1)$, where $A(t)$ is the age of t (i.e., time since arrival), $U(t)$ is the time since t arrives in the current queue until the lock becomes available, and $N(t)$ is the number of transactions in the current queue that will be scheduled before t. However, VATS does not account for the fact that $U(t)$ itself can be affected by the scheduling decision. In this paper, in our analysis of bLDSF, we have been able to remove both assumptions and hence, prove optimality under a more realistic setting. We consider both shared and exclusive locks, and account for the impact of our scheduling decision on the wait times of other transactions waiting for other objects in the system. Our experiments show that bLDSF's more realistic assumptions lead to better decisions (see Section 6).

Deadlock Resolution — The problem of *deadlock resolution* is about deciding which transaction(s) to abort in order to resolve a deadlock [65, 37, 56, 43, 32]. Typically, transactions with lower priority are aborted, in order to reduce the amount of work wasted. Here, transactions are prioritized based on their age [32, 10], deadline [48], or number of held locks [57]. Franaszek et al. [32] empirically show that an age-based priority improves concurrency, and reduces the amount of work wasted. Agrawal et al. [10] argue that choosing victims based on their age and number of held locks leads to fewer rollbacks, than (i) choosing a transaction randomly, or (ii) aborting the most recently blocked transaction. These proposals take contention into account only for deadlock resolution. In contrast, we focus on lock scheduling and show that contention-aware scheduling yields significant performance benefits in practice.

8. CONCLUSION

We study a fundamental (yet, surprisingly overlooked) problem: *lock scheduling* in a database system. Despite the massive body of work on transactional databases, the astonishing impact of lock scheduling on overall performance of a transactional system seems to have been largely underrecognized—to the extent that every DBMS to date has simply relied on FIFO. To our knowledge, we are the first to propose the idea of contention-aware lock scheduling, and present efficient algorithms that are guaranteed to reduce mean transaction latencies down to a constant-factor-approximation of the optimal scheduling. We also empirically confirm our theoretical analysis by modifying a real-world DBMS. Our extensive experiments show that our algorithms reduce transaction latencies by up to two orders of magnitude, while delivering 6.5x higher throughput. More importantly, our algorithm has already been adopted by MySQL, and has started to impact real world applications.

9. ACKNOWLEDGEMENT

This work is supported in part by National Science Foundation (grants 1544844, 1553169, and 1629397). The authors are grateful to Mark Callaghan, Jan Lindström, Sunny Bains, and Laurynas Biveinis (for integrating our algorithm into MySQL, MariaDB, and Percona Server), to Shengqi Wu (for his early contributions), and to Morgan Lovay (for her comments).

10. REFERENCES

[1] Contention-aware transaction scheduling arriving in InnoDB to boost performance. http://mysqlserverteam.com/contention-aware-transaction-scheduling-arriving-in-innodb-to-boost-performance/.

[2] Db2 documentation. https://www.ibm.com/support/knowledgecenter/en/SSEPEK_12.0.0/perf/src/tpc/db2z_lockcontention.html.

[3] Mysql source code. https://github.com/mysql/mysql-server/blob/5.7/storage/innobase/lock/lock0lock.cc.

[4] Overview of teradata database locking. http://info.teradata.com/HTMLPubs/DB_TTU_16_00/index.html#page/General_Reference%2FB035-1091-160K%2Fmtg1472241438567.html.

[5] Postgres documentation. https://github.com/postgres/postgres/blob/master/src/backend/storage/lmgr/README.

[6] Sql server, lock manager, and relaxed fifo. https://blogs.msdn.microsoft.com/psssql/2009/06/02/sql-server-lock-manager-and-relaxed-fifo/.

[7] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions. *SIGMOD Rec.*, pages 71–81, 1988.

[8] R. K. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. *TODS*, pages 513–560, 1992.

[9] B. Adelberg, B. Kao, and H. Garcia-Molina. Database support for efficiently maintaining derived data. In *EDBT*, pages 223–240, 1996.

[10] R. Agrawal, M. J. Carey, and L. W. McVoy. The performance of alternative strategies for dealing with deadlocks in database management systems. *TSE*, pages 1348–1363, 1987.

[11] D. V. Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In *SIGMOD*, pages 1009–1024, 2017.

[12] S. Altmeyer, S. M. Sundharam, and N. Navet. The case for fifo real-time scheduling. Technical report, 2016.

[13] R. F. Aranha, V. Ganti, S. Narayanan, C. Muthukrishnan, S. Prasad, and K. Ramamritham. Implementation of a real-time database system. *Information Systems*, pages 55–74, 1996.

[14] C. Bector, Y. P. Gupta, and M. C. Gupta. V-shape property of optimal sequence of jobs about a common due date on a single machine. *Computers & operations research*, pages 583–588, 1989.

[15] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, pages 185–221, 1981.

[16] D. P. Bovet and M. Cesati. *Understanding the Linux kernel*. 2005.

[17] X. Cai. V-shape property for job sequences that minimize the expected completion time variance. *EJOR*, 1996.

[18] M. J. Carey and M. Stonebraker. The performance of concurrency control algorithms for database management systems. In *VLDB*, pages 107–118, 1984.

[19] S. Chakravarthy, D.-K. Hong, and T. Johnson. Real-time transaction scheduling: A framework for synthesizing static and dynamic factors. *RTSS*, pages 135–170, 1998.

[20] B.-C. Choi, S.-H. Yoon, and S.-J. Chung. Minimizing maximum completion time in a proportionate flow shop with one machine of different speed. *EJOR*, pages 964–974, 2007.

[21] J. Cipar, Q. Ho, J. K. Kim, S. Lee, G. R. Ganger, G. Gibson, K. Keeton, and E. P. Xing. Solving the straggler problem with bounded staleness. In *HotOS*, pages 22–22, 2013.

[22] A. DasGupta. Finite sample theory of order statistics and extremes. In *Probability for Statistics and Machine Learning*, pages 221–248. 2011.

[23] L. Daynès, O. Gruber, and P. Valduriez. Locking in oodbms client supporting nested transactions. In *ICDE*, pages 316–323, 1995.

[24] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, 2013.

[25] J. Du and J. Y.-T. Leung. Scheduling tree-structured tasks with restricted execution times. *Information processing letters*, pages 183–188, 1988.

[26] J. Du and J. Y.-T. Leung. Scheduling tree-structured tasks on two processors to minimize schedule length. *SIDMA*, pages 176–196, 1989.

[27] A. C. Dusseau, R. H. Arpaci, and D. E. Culler. Effective distributed scheduling of parallel workloads. *ACM SIGMETRICS Performance Evaluation Review*, pages 25–36, 1996.

[28] S. Eilon and I. Chowdhury. Minimising waiting time variance in the single machine problem. *Management Science*, pages 567–575, 1977.

[29] B. Eisenberg. On the expectation of the maximum of iid geometric random variables. *Statistics & Probability Letters*, pages 135–143, 2008.

[30] A. Feldmann, M.-Y. Kao, J. Sgall, and S.-H. Teng. Optimal online scheduling of parallel jobs with dependencies. In *STOC*, pages 642–651, 1993.

[31] K. B. Ferreira, P. G. Bridges, R. Brightwell, and K. T. Pedretti. The impact of system design parameters on application noise sensitivity. *Cluster computing*, pages 117–129, 2013.

[32] P. Franaszek and J. T. Robinson. Limitations of concurrency in transaction processing. *TODS*, 1985.

[33] L. George and P. Minet. A fifo worst case analysis for a hard real-time distributed problem with consistency constraints. In *ICDCS*, pages 441–448, 1997.

[34] A. Guinet and M. Solomon. Scheduling hybrid flowshops to minimize maximum tardiness or maximum completion time. *IJPR*, pages 1643–1654, 1996.

[35] L. A. Hall, D. B. Shmoys, and J. Wein. Scheduling to minimize average completion time: Off-line and on-line algorithms. In *SODA*, pages 142–151, 1996.

[36] J. R. Haritsa, M. J. Canrey, and M. Livny. Value-based scheduling in real-time database systems. *VLDBJ*, pages 117–152, 1993.

[37] J. R. Haritsa, M. J. Carey, and M. Livny. Data access scheduling in firm real-time database systems. *RTSS*, pages 203–241, 1992.

[38] J. R. Haritsa, M. Livny, and M. J. Carey. Earliest deadline scheduling for real-time database systems. In

RTSS, pages 232–242, 1991.

[39] C. He, J. Y.-T. Leung, K. Lee, and M. L. Pinedo. Improved algorithms for single machine scheduling with release dates and rejections. *4OR*, pages 41–55, 2016.

[40] D. Hong, T. Johnson, and S. Chakravarthy. *Real-time transaction scheduling: a cost conscious approach.* 1993.

[41] J. A. Hoogeveen and A. P. Vestjens. Optimal on-line algorithms for single-machine scheduling. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 404–414, 1996.

[42] W. Horn. Single-machine job sequencing with treelike precedence ordering and linear delay penalties. *SIAP*, pages 189–202, 1972.

[43] J. Huang. *Real-time transaction processing: design, implementation, and performance evaluation.* PhD thesis, University of Massachusetts, 1991.

[44] J. Huang, B. Mozafari, G. Schoenebeck, and T. Wenisch. A top-down approach to achieving performance predictability in database systems. In *SIGMOD*, 2017.

[45] J. Huang, B. Mozafari, and T. Wenisch. Statistical analysis of latency through semantic profiling. In *EuroSys*, 2017.

[46] T. Ibaraki, T. Kameda, and N. Katoh. Cautious transaction schedulers for database concurrency control. *TSE*, pages 997–1009, 1988.

[47] G. C. Kamath. Bounds on the expectation of the maximum of samples from a gaussian. *URL http://www.gautamkamath.com/writings/gaussian_max.pdf*, 2015.

[48] B. Kao and H. Garcia-Molina. An overview of real-time database systems. In *Real Time Computing*, pages 261–282. 1994.

[49] A. M. Krieger and M. Raghavachari. V-shape property for optimal schedules with monotone penalty functions. *Computers & operations research*, pages 533–534, 1992.

[50] J. Lee and S. H. Son. Performance of concurrency control algorithms for real-time database systems., 1996.

[51] J. P. Lehoczky. Scheduling communication networks carrying real-time traffic. In *RTSS*, pages 470–479, 1998.

[52] J. K. Lenstra, A. R. Kan, and P. Brucker. Complexity of machine scheduling problems. *Annals of discrete mathematics*, pages 343–362, 1977.

[53] H. Leontyev and J. H. Anderson. Tardiness bounds for fifo scheduling on multiprocessors. In *ECRTS*, pages 71–71, 2007.

[54] Q. Lin, P. Chang, G. Chen, B. C. Ooi, K.-L. Tan, and Z. Wang. Towards a non-2pc transaction management in distributed database systems. In *SIGMOD*, pages 1659–1674, 2016.

[55] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *JACM*, pages 46–61, 1973.

[56] P. P. Macri. Deadlock detection and resolution in a codasyl based data management system. In *SIGMOD*, pages 45–49, 1976.

[57] D. P. Mitchell and M. J. Merritt. A distributed algorithm for deadlock detection and resolution. In *PODC*, pages 282–284, 1984.

[58] B. Mozafari, C. Curino, A. Jindal, and S. Madden. Performance and resource modeling in highly-concurrent OLTP workloads. In *SIGMOD*, 2013.

[59] B. Mozafari, C. Curino, and S. Madden. DBSeer: Resource and performance prediction for building a next generation database cloud. In *CIDR*, 2013.

[60] B. Mozafari, J. Ramnarayan, S. Menon, Y. Mahajan, S. Chakraborty, H. Bhanawat, and K. Bachhav. Snappydata: A unified cluster for streaming, transactions, and interactive analytics. In *CIDR*, 2017.

[61] R. R. Muntz and E. G. Coffman Jr. Preemptive scheduling of real-time tasks on multiprocessor systems. *JACM*, pages 324–338, 1970.

[62] J. Pei, X. Liu, P. M. Pardalos, W. Fan, and S. Yang. Scheduling deteriorating jobs on a single serial-batching machine with multiple job types and sequence-dependent setup times. *Annals of Operations Research*, pages 175–195, 2017.

[63] F. Petrini, D. J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 55–55, 2003.

[64] C. Phillips, C. Stein, and J. Wein. Scheduling jobs that arrive over time. In *WADS*, pages 86–97, 1995.

[65] K. Ramamritham. Real-time databases. *Distributed and parallel databases*, pages 199–226, 1993.

[66] R. Rastogi, S. Seshadri, P. Bohannon, D. Leinbaugh, A. Silberschatz, and S. Sudarshan. Improving predictability of transaction execution times in real-time databases. *RTSS*, pages 283–302, 2000.

[67] A. J. Ruiz-Torres and G. Centeno. Scheduling with flexible resources in parallel workcenters to minimize maximum completion time. *Computers & operations research*, pages 48–69, 2007.

[68] L. Sha, R. Rajkumar, and J. P. Lehooczky. Concurrency control for distributed real-time databases. *SIGMOD Rec.*, 1988.

[69] J. B. Sidney. Decomposition algorithms for single-machine sequencing with precedence relations and deferral costs. *Operations Research*, pages 283–298, 1975.

[70] W. E. Smith. Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, pages 59–66, 1956.

[71] J. A. Stankovic, M. Spuri, K. Ramamritham, and G. C. Buttazzo. *Deadline scheduling for real-time systems: EDF and related algorithms.* 2012.

[72] B. Tian, J. Huang, B. Mozafari, and G. Schoenebeck. Contention-aware lock scheduling for transactional databases. *Technical Report*, https://web.eecs.umich.edu/~mozafari/php/data/uploads/lock-schd-report.pdf, 2017.

[73] D. Towsley and S. Panwar. On the optimality of minimum laxity and earliest deadline scheduling for real-time multiprocessors. In *Workshop on Real Time*, pages 17–24, 1990.

[74] S.-M. Tseng, Y.-H. Chin, and W.-P. Yang. Scheduling

real-time transactions with dynamic values: a performance evaluation. In *Workshop on Real-Time Computing Systems and Applications*, pages 60–67, 1995.

[75] Ö. Ulusoy and G. G. Belford. Real-time transaction scheduling in database systems. *Information Systems*, pages 559–580, 1993.

[76] V. Vani and M. Raghavachari. Deterministic and random single machine sequencing with variance minimization. *Operations Research*, pages 111–120, 1987.

[77] M. Xiong, Q. Wang, and K. Ramamritham. On earliest deadline first scheduling for temporal consistency maintenance. *RTSS*, pages 208–237, 2008.

[78] D. Y. Yoon, B. Mozafari, and D. P. Brown. Dbseer: Pain-free database administration through workload intelligence. *PVLDB*, 8(12):2036–2039, 2015.

[79] R. Zajcew, P. Roy, D. L. Black, C. Peak, P. Guedes, B. Kemp, J. LoVerso, M. Leibensperger, M. Barnett, F. Rabii, et al. An osf/1 unix for massively parallel multicomputers. In *USENIX Winter*, pages 449–468, 1993.