Database Isolation By Scheduling

Kevin P. Gaffney
kpgaffney@wisc.edu
University of Wisconsin-Madison

Robert Claus
robert@datachat.ai
DataChat Inc.

Jignesh M. Patel
jignesh@cs.wisc.edu
University of Wisconsin-Madison

ABSTRACT

Transaction isolation is conventionally achieved by restricting access to the physical items in a database. To maximize performance, isolation functionality is often packaged with recovery, I/O, and data access methods in a monolithic transactional storage manager. While this design has historically afforded high performance in online transaction processing systems, industry trends indicate a growing need for a new approach in which intertwined components of the transactional storage manager are disaggregated into modular services. This paper presents a new method to modularize the isolation component. Our work builds on predicate locking, an isolation mechanism that enables this modularization by locking logical rather than physical items in a database. Predicate locking is rarely used as the core isolation mechanism because of its high theoretical complexity and perceived overhead. However, we show that this overhead can be substantially reduced in practice by optimizing for common predicate structures.

We present DIBS, a transaction scheduler that employs our predicate locking optimizations to guarantee isolation as a modular service. We evaluate the performance of DIBS as the sole isolation mechanism in a data processing system. In this setting, DIBS scales up to 10.5 million transactions per second on a TATP workload. We also explore how DIBS can be applied to existing database systems to increase transaction throughput. DIBS reduces per-transaction file system writes by 90% on TATP in SQLite, resulting in a 3X improvement in throughput. Finally, DIBS reduces row contention on YCSB in MySQL, providing serializable isolation with a 1.4X improvement in throughput.

PVLDB Reference Format:
Kevin P. Gaffney, Robert Claus, and Jignesh M. Patel. Database Isolation By Scheduling. PVLDB, 14(9): 1467 - 1480, 2021.
doi:10.14778/3461535.3461537

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/UWHustle/dibs.

1 INTRODUCTION

In a conventional database system architecture, concurrency control is packaged with recovery, I/O, and data access methods in a monolithic transactional storage manager [20]. Together, these components of the storage manager provide the ACID properties on which applications rely. While each ACID property can be conceptually regarded as a separate guarantee, the components that provide these guarantees are often deeply intertwined. For example, the recovery manager may depend on the concurrency control mechanism to ensure that modifications to the database can be safely undone when a transaction is rolled back.

While this design has historically afforded high performance in online transaction processing (OLTP) systems, the interdependence between components in the storage manager may result in complex software that is difficult to maintain. This complexity may result in subtle bugs that go undetected, even in rigorous testing. For example, the Elle transaction isolation checker [24] recently detected G2-item anomaly, a violation of serializability, in the serializable isolation level of PostgreSQL 12.3 [23]. Moreover, the complexity of intertwined system components may introduce unexpected and undesirable performance behavior. For example, for some workloads on MySQL with InnoDB, switching the transaction isolation level from serializable to read uncommitted *decreases* performance, contrary to expectations. We quantify this behavior in Figure 1. In this experiment, a YCSB variant workload consisting of 50% select and 50% update was executed against a table of 10,000 rows, each consisting of an integer primary key and one 100-character field. Each request scans the entire table looking for a single row that satisfies a predicate, then either updates or returns it. Surprisingly, read uncommitted performance degrades more easily than serializable. At 96 connections, read uncommitted isolation yields only 66% of the throughput of serializable. The complexity of the concurrency control mechanism makes it difficult to determine the reason for this behavior. One possible explanation is that in read uncommitted isolation, record locks are released for *every* nonmatching row immediately after evaluating the `WHERE` condition. This may incur additional overhead compared to serializable isolation, in which record locks are held for the duration of the transaction.

The complexity of monolithic transactional storage management also increases the development time necessary to add transactional support to a new system. As noted in the "one size does not fit all" approach [43], different applications may need very different query processing and storage systems. To meet these varied demands, there has recently been an explosion of new data processing platforms that address a wide range of problems. Indeed, as of December 2020, 360 different database systems were tracked by DB-Engines [9], and 5 out of the top 10 were not present at the turn of the century. Developers of new systems may be dissuaded by the effort required to implement bundled concurrency control, recovery, and other transactional functionality. A notable recent example is RocksDB, in which transactions were initially unsupported. Transactions with both pessimistic and optimistic concurrency control protocols have since been added [39]. Furthermore, new systems may initially compromise performance in favor of simplifying transactional support. For example, prior to its 2.2 release, MongoDB

Proceedings of the VLDB Endowment, Vol. 14, No. 9 ISSN 2150-8097.
doi:10.14778/3461535.3461537

used a global database lock to serialize modifications to the database [32].

In this paper, we propose a new approach to extract transaction isolation functionality from the other components of the transactional storage manager. Our approach provides isolation guarantees as a modular service that is agnostic to the contents of the database and the implementation of the other data processing components. This design reduces the complexity of transactional storage management and facilitates adding transactional support to new systems. We refer to this design as *isolation by scheduling*. In this paradigm, a transactional request is allowed to proceed only if it is guaranteed not to conflict with a concurrent request, regardless of the state of the database. The database system may then execute these requests without additional isolation mechanisms, such as locking and versioning.

To modularize isolation, we build on *predicate locking* [14], an often overlooked isolation mechanism. Under the predicate locking protocol, each lock is associated with a predicate. Predicates are then analyzed to determine whether locks conflict. Each lock then refers to a *logical* subset of the database. Contrast this to conventional isolation mechanisms, which deal with *physical* items in a database such as tables and rows. The distinction between logical and physical isolation allows predicate locking to be implemented as a modular component, while other isolation mechanisms are usually intertwined with other components in the transactional storage manager. Unfortunately, general predicate locking is NP-complete as it can be reduced to the boolean satisfiability problem. Consequently, it has received relatively little attention in the database community. To the best of our knowledge, general predicate locking is not used as the *sole* isolation mechanism in any mainstream database system. However, in this paper, we present several predicate locking optimizations that substantially improve its performance and scalability, making it a viable isolation mechanism for our approach.

Motivated by the need for modular isolation, in this paper we propose a transaction scheduling system that uses optimized predicate locking to provide isolation as a service. Specifically, this paper makes the following contributions.

(1) A set of novel predicate locking optimizations that offer reduced overhead and improved scalability compared to a naive implementation of predicate locking.
(2) A transaction scheduling system, which we call Database Isolation By Scheduling (DIBS), that combines these optimizations with a transaction scheduler to provide serializable isolation as a modular service.
(3) Multiple applications for DIBS that have the potential to reduce file system writes and increase concurrency, improving throughput on transactional workloads.
(4) An evaluation of our approach on a variety of data settings that demonstrates the effectiveness of our approach.

The remainder of this paper is organized as follows. In Section 2, we describe our predicate locking optimizations. In Section 3, we present a transaction scheduling system that combines our optimizations to provide modular isolation. In Section 4, we identify and evaluate three key applications for our system. In Section 5, we discuss the implications and assumptions associated with providing transaction isolation as a service. Related work is presented in Section 6, and our concluding remarks are in Section 7.

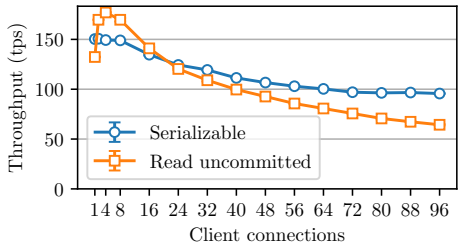


Figure 1: InnoDB read uncommitted performance can drop below serializable for write-intensive YCSB.

2 OPTIMIZED PREDICATE LOCKING

Predicate locking has the potential to provide the benefits of modularity and simplicity described in Section 1, but has received relatively little attention due to its high computational complexity. To build on the potential of predicate locking, we propose several optimizations that reduce its overhead and improve its scalability. In this section, we provide a brief overview of the predicate locking task. We then present our optimizations and describe how each addresses a specific source of overhead in naive predicate locking.

2.1 Naive predicate locking

Predicate locking is a form of transaction isolation that was originally proposed as a solution to the phantom read problem [14]. It is well-known that the locking of physical data items provides serializability only if the set of data items is fixed. Data items that do not yet exist cannot be locked. Hence, concurrent scan and insert operations may lead to a violation of serializability. In contrast, a predicate locking system locks a logical rather than physical subset of the database. This prevents the insertion of phantom data items into the set accessed by a transaction, and thus provides full serializability. While prior work has discussed predicate locking as a solution to the phantom read problem, we focus on an additional benefit in this paper: its ability to provide isolation as an independent service.

Under the predicate locking protocol, the system maintains a list of active predicate locks associated with in-progress transactions. A predicate lock is defined as a tuple $L = (R, P, a)$ where R is a relation, P is a predicate that can be evaluated on a tuple t in R, and a is the access mode ($read$ or $write$). Two predicate locks $L = (R, P, a)$ and $L' = (R', P', a')$ are said to conflict if all of the following conditions are true.

(1) $R = R'$
(2) $a = write$ or $a' = write$
(3) There exists some *feasible* tuple t such that $P(t) \wedge P'(t) = TRUE$.

While (1) and (2) can be determined in constant time, (3) equates to the boolean satisfiability problem which is generally NP-complete.

Using this definition of conflict, in Algorithm 1 we propose a naive algorithm to acquire a predicate lock.

The naive algorithm solves satisfiability of $P \wedge P'$ by converting to disjunctive normal form (DNF) in line with prior work [14]. The conjunction $P \wedge P'$ is unsatisfiable if and only if each term of its equivalent DNF predicate is a contradiction. Because each term is a conjunction, these contradictions can be easily identified. For example, suppose we have the predicates $P \equiv (a = 1 \vee b > 2)$ and $P' \equiv (a = 3 \wedge b = 4)$. Converting to DNF, $P \wedge P'$ is equivalent to the predicate

$$(a = 1 \wedge a = 3 \wedge b = 4) \vee (b > 2 \wedge a = 3 \wedge b = 4).$$

Clearly, the first term is unsatisfiable as no single tuple can satisfy both $a = 1$ and $a = 3$. However, the second term contains no such contradiction. Therefore, $P \wedge P'$ is satisfiable. While conceptually straightforward, converting an arbitrary formula to DNF can lead to worst-case exponential blowup in time and space.

Furthermore, in the naive algorithm, a thread that wishes to acquire a new predicate lock must check for conflict against all predicate locks held by other transactions in the system. In a critical section, the thread must save a snapshot of the active predicate lock set, then insert the new predicate lock into the set. This operation is denoted by the FetchInsert function call in Algorithm 1, where S' is the snapshot. Then, for each active predicate lock in the snapshot, the thread tests all the conditions described above. If any of them are true, the thread waits until the active predicate lock is released.

To summarize, in the naive algorithm, the complexity of acquiring a new predicate lock depends on both the number of existing predicate locks in the system and the size of the predicates in the predicate locks. For these reasons, the overhead of general predicate locking is deemed too high, and to the best of our knowledge it is not used as the *sole* isolation mechanism in high-performance database systems.

Algorithm 1: AcquirePredicateLock

```
Input: A desired predicate lock L and a set S of existing
       predicate locks
L_DNF ← ToDNF(L)
S' ← FetchInsert(S, L_DNF)
for L' ∈ S' do
    if L and L' conflict then
        wait until L' is released
    end
end
```

2.2 Optimized Predicate Locking

We now present our optimizations to the naive predicate locking algorithm and describe how each improves its performance and scalability.

2.2.1 Conjunct grouping. As stated earlier, converting an arbitrary boolean formula to DNF can result in exponential blowup in time and space. The *conjunct grouping* optimization aims to reduce the overhead of converting to DNF. Observe that each term of a DNF predicate must contain at least two references to the same column to be a possible contradiction. For example, the predicate $(a = v_1 \wedge a = v_2)$ is a contradiction if $v_1 \neq v_2$, whereas the predicate $(a = v_1 \wedge b = v_2)$ is satisfiable for all v_1 and v_2. Hence, we can avoid some unnecessary work by distributing the conjunction operator only over conjuncts that share common columns. The conjunct grouping optimization separates the conjuncts of P and P' into groups that access disjoint sets of columns. This can be accomplished in close to linear time with the use of a disjoint-set data structure [46]. Each group is then converted to DNF individually and tested for conflict as described in Section 2.1. If there are many such groups, the exponential blowup of the conversion to DNF may be greatly reduced. For example, consider the predicates

$$P \equiv ((a = 1 \vee a = 2) \wedge (b = 1 \vee b = 2))$$
$$P' \equiv ((a = 3 \vee a = 4) \wedge (b = 3 \vee b = 4)).$$

Converting $P \wedge P'$ to DNF results in a disjunction with 16 terms. By first applying conjunct grouping, we need only consider 2 disjunctions, each with 4 terms.

As a caveat, we note that in OLTP workloads, predicates are often *already* in DNF. In fact, three of the most widely used OLTP benchmarks, TPC-C [7], TATP [34], and YCSB [6], do not include any predicates with disjunctions. Rather, they include only predicates that are conjunctions of comparisons. For these predicates, we can test for conflict in linear time with respect to the number of conjuncts. In such OLTP workloads, the NP-completeness of boolean satisfiability is unlikely to be a significant performance consideration.

2.2.2 Prepared predicates. Applications commonly use *prepared statements*, which are parameterized declarative statements submitted to the data platform for parsing and optimization. The compiled statements can be later executed with values provided by the client. An application's use of prepared statements, as opposed to *ad hoc* statements, has two main advantages. First, the overhead of parsing a statement is incurred only once, rather than each time the statement is executed. If the query plan does not change depending on the parameter values, the same is true for the overhead of optimizing the statement. Second, prepared statements provide protection against SQL injection attacks, assuming that parameter values are passed safely to the statement [3].

In a similar manner, we analyze the predicates of prepared statements, which we refer to as *prepared predicates*, to reduce the overhead of determining whether two predicate locks conflict. The analysis makes use of a graph data structure, which we refer to as the *conflict graph*. For each prepared predicate, we include a vertex in the conflict graph. We connect vertices whose predicates satisfy both conditions (1) and (2) in Section 2.1, that is, they access the same relation and at least one is a write. At each edge, we store a *conflict predicate* that, when evaluated on the parameter values of the two prepared predicates, evaluates to true if the prepared predicates are satisfiable. For example, consider parameterized versions of the previously described predicates $P \equiv (a = v_1 \vee b > v_2)$ and $P' \equiv (a = v_3 \wedge b = v_4)$. At the edge connecting P and P' in the conflict graph, we store the conflict predicate

$$(v_1 = v_3 \wedge v_2 < v_4).$$

To solve satisfiability between two prepared predicates, we then evaluate the conflict predicate at the corresponding edge with the provided parameter values. Offline computation and simplification

of these conflict predicates shortens critical code paths and reduces overhead during online processing.

2.2.3 Column filtering. Notice that in Algorithm 1, there is a potential scalability bottleneck caused by concurrent access to a centralized set of predicate locks. As shown in the algorithm, the FetchInsert operation must atomically insert a predicate lock into the active set and return a snapshot of the set before the insertion. In practice, the active set of predicate locks is protected by a mutex to ensure correct behavior. Each thread must compete for ownership of this mutex, limiting scalability. As we will show later in a naive implementation of predicate locking, contention for the active set of predicate locks may lead to a degradation in overall throughput as more threads are added.

This bottleneck is especially pronounced in main-memory OLTP applications, where the relative overhead of concurrency control is greater compared to IO-bound OLTP or analytical applications [38]. Main-memory OLTP applications are characterized by statements that access only a small subset of the database, typically using fast index lookups. Because these statements can often be executed quickly, the system spends a greater portion of time ensuring isolation. Accordingly, high-performance OLTP systems take great care to avoid centralized scalability bottlenecks. Furthermore, we observe that in OLTP workloads, it is common for requests to include predicates on primary keys. In a typical OLTP system, the query planner analyzes each predicate to determine which indexes, if any, can be used to evaluate the predicate. The indexes are then used to find tuples that satisfy the predicate, avoiding the need for a full table scan. We use an analogous approach to test predicate lock conflicts.

The *column filtering* optimization aims to alleviate this bottleneck. Rather than maintain a single set of active predicate locks, we instead construct N buckets of active predicate locks, where N is a configurable parameter. We then define a function f mapping each predicate's values to a subset of the buckets such that, if two predicates conflict, there must exist some bucket to which both of them are mapped. To acquire some predicate lock $L = (R, P, a)$, we then call Algorithm 1 for each bucket in the mapping $f(P)$. The advantage of this approach is twofold. First, it may be possible to map two predicates to different buckets by only examining a small portion of each predicate, avoiding the need for a full comparison as before. Second, this mapping is done in parallel: each thread computes the mapping for its current predicate and then examines the corresponding buckets for other predicates that may conflict. For typical OLTP workloads, this optimization reduces contention for centralized resources and improves scalability.

2.2.4 Additional optimizations. Lastly, we describe two additional optimizations for naive predicate locking. First, we propose a modification to conditions (1) and (2) above. Rather than test whether two predicate locks access the same relation, we instead test whether two predicate locks access the same *column*, and at least one of them is a write. This optimization provides higher concurrency by allowing rows to be shared more freely among concurrent transactions, while maintaining serializability by ensuring data that is accessed by multiple transactions is only read and never written. As we will show in our results, this optimization increases throughput for workloads that access different columns of contended rows. The assumptions about database system behavior that enable this optimization are discussed in Section 5.

Second, we present a simple mechanism to avoid the rare worst-case scenario in which the time required to test whether two predicate locks conflict is greater than the time required to execute the two requests serially. Before converting to DNF, we first compute the number of terms in the result. This calculation can be carried out in linear time with respect to the size of the predicates using integer multiplication and addition. If the number of terms is greater than some threshold k, we fall back on full column locks. The threshold k can be specified on a per-request basis or as a global parameter. Conceivably, the system could tune k dynamically based on the latency of previous requests, though we leave this optimization to future work.

3 DATABASE ISOLATION BY SCHEDULING (DIBS)

Building on our predicate locking optimizations, we now present a scalable transaction scheduler that uses efficient predicate locking to provide isolation as a service. We refer to this system as Database Isolation By Scheduling (DIBS). We begin with an overview of the core modules of DIBS and how they interact, followed by a discussion of how DIBS combines each predicate locking optimization together.

3.1 Architecture

DIBS is designed to be a modular abstraction for a data platform, with respect to the discussion in Section 1. It is implemented as a pluggable layer between one or more client applications and a target data platform or query processing engine. The DIBS system consists of the modules shown in Figure 2. In this section, we describe each of these modules in detail.

The *client connector* and *database connector* modules are the interface between DIBS, client applications, and target data platforms. These modules expose the API functions shown in Figure 2 that can be implemented to provide isolation in a variety of contexts. The client connector API can be implemented to accept requests from a new application, such as a command line interface or a web application. Internally, client connectors maintain a queue of client requests to be processed by the DIBS system. The database connector API can be implemented to schedule requests on a new target data platform. Transactional control statements such as begin() and commit() are included in the database connector API so that the data platform can manage transaction context and provide atomicity and durability guarantees if desired. The data platform need not track transaction context for isolation between transactions, as the requests scheduled by DIBS are already isolated.

The *predicate lock manager* (PLM) maintains the set of active predicate locks in the system. Though analogous to a lock manager in a traditional database system, the PLM is distinct in that it stores predicate locks rather than references to physical data items in the database. This design allows it to remain independent of any abstraction and data structures used in the data platform. The goals of the PLM are to determine which predicate locks must to be acquired for a given request, and to acquire those locks with as little overhead as possible. To achieve these goals, the PLM employs each

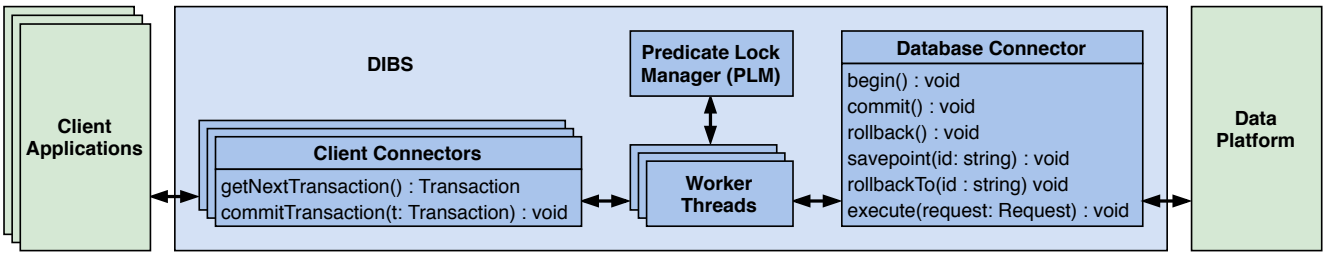


Figure 2: Architecture of the DIBS system.

of the predicate locking optimizations described earlier. Section 3.2 discusses how the PLM combines these optimizations.

The PLM is also responsible for managing deadlock. Our design offers the flexibility to choose the deadlock avoidance or detection method that best suits the workload. In our implementation and subsequent experiments, we use a timeout with random jitter when waiting for a predicate lock to be acquired. If the timeout expires, DIBS rolls back and restarts the current transaction. This strategy was chosen for its simplicity and its performance under high contention [5].

The *worker* is the main driver of the DIBS system and the mechanism by which DIBS scales up to provide higher transaction throughput. Each worker occupies a thread and executes the following operations in a loop. First, the worker calls the client connector API to receive a new transaction from the client connector's internal queue. Transactions include a list of requests, which can be submitted dynamically and are typically in the form of prepared SQL statements. Requests that are not prepared are parsed into their object representations. For each request, the worker calls the PLM to acquire the predicate locks that are needed to isolate the request from concurrent transactions. The worker thread may block as it waits for conflicting predicate locks to be released. The worker's attempt to acquire predicate locks results in one of the following two scenarios.

(1) The predicate locks are successfully acquired. The worker then calls the database connector API to execute the request on the target data platform.
(2) The predicate locks could not be acquired due to potential deadlock or another exception. In this case, the worker instructs the data platform to rollback the current transaction through the database connector API. We limit the scope of DIBS to isolation and leave the specific rollback implementation to the data platform or other modular components (e.g., approach outlined in [4]). The worker then calls the predicate manager to release all locks associated with the current transaction and restarts the transaction from the beginning.

After all requests have been completed, the worker instructs the data platform to commit the transaction. The data platform may use this commit hook to clean up transaction context and flush database changes to stable storage. The worker then calls the PLM to release all predicate locks associated with the current transaction. Both read and write predicate locks are released only after the commit hook is completed by the data platform to comply with the strong strict two-phase locking (SS2PL) protocol. Finally, the worker notifies the client application of the transaction commit.

3.2 Combining predicate locking optimizations

The DIBS PLM combines our predicate locking optimizations into a single service that efficiently acquires and releases predicate locks, allowing DIBS to provide low-overhead modular isolation. In this section, we describe how the PLM layers each of these optimizations together. We structure this discussion into two phases: initialization and online processing.

The input to the PLM during the initialization phase is a list of *request templates*. Request templates are comprised of a set of IDs representing the columns that the request reads, an additional set of IDs for the columns that the request writes, and a prepared (parameterized) predicate. This list may be a subset of the requests to be executed during online processing, or it may be empty. However, predicates that are not included in this list will be evaluated *ad hoc* and will not benefit from the prepared predicates optimization. The PLM iterates through the request templates, constructing the conflict graph described in Section 2.2.2. It also queries the target database schema to determine which columns to use for the column filtering optimization described in Section 2.2.3. In our evaluation, we empirically choose N, the number of buckets of predicate locks for each relation, to maximize throughput. We define the mapping function f to be a hash of the primary key of each relation modulo N.

During online processing, the PLM accepts both *prepared* and *ad hoc* predicate lock requests. For either type of request, the PLM attempts to compute the mapping f of the request's predicate. If the predicate cannot be mapped (i.e., it does not reference the primary key of the relation or requires a full scan), a new predicate lock is inserted into each of the N buckets. Otherwise, a predicate lock is only inserted into the buckets specified by the mapping. In both cases, the PLM uses calls to AcquirePredicateLock described in Algorithm 1 to insert predicate locks into buckets. Up to this point, prepared and ad hoc requests are handled similarly. However, the PLM uses different methods for each case to check whether two predicate locks conflict, which are discussed in the following paragraphs.

Prepared requests consist of an ID, which corresponds to a request template provided during initialization, and a list of parameter values. For each other predicate lock in the snapshot S' returned by FetchInsert in Algorithm 1, if the predicate lock originated from a prepared request, the PLM examines the edge in the conflict graph

connecting the two prepared predicates. The conflict predicate at that edge is then evaluated with the provided parameter values to determine whether the two predicate locks conflict. If the other predicate lock originated from an ad hoc request, the PLM treats both predicates as ad hoc, and uses the conjunct grouping optimization to determine whether they conflict.

Ad hoc requests consist of a request template (distinct from the request templates provided during the initialization phase) and a list of parameter values. For each other predicate lock in S', the PLM treats both predicates as ad hoc, regardless of whether the other predicate lock originated from a prepared request, and uses the conjunct grouping optimization to determine whether they conflict.

Eventually, predicate locks must be released due to transaction commit or rollback. To aid in the releasing of predicate locks, DIBS associates a *context* with each transaction. When the PLM inserts a predicate lock into a bucket using Algorithm 1, it also stores a reference to that bucket and a unique ID associated with the predicate lock in the transaction context. Then, when a worker instructs the PLM to release the predicate locks associated with a transaction, the PLM uses the transaction context to determine which predicate locks must be removed from which buckets.

3.3 Extensions to More Complex Predicates

Up to this point, we have only considered predicates of the form *column comparison value*, where *column* is a reference to a column in the database, *comparison* is any of the comparison operators in $\{<, \leq, >, \geq, =, \neq\}$, and *value* is a scalar literal. Here, we discuss how DIBS can be extended to support predicates with more complex expressions.

It is straightforward to add support for predicates of the form *column comparison expression*, where *expression* is an expression tree that may include arithmetic operators or functions that operate on scalar values, but does not include any column references. Predicates in this form are often called "sargable", because they can potentially take advantage of a database index to speed up search [41]. To acquire a predicate lock for a predicate in this form, DIBS would first issue a query to evaluate the expression in the data platform. Because the expression does not access any physical data, the data platform may evaluate the expression without concurrency control. The scalar result of the expression can then be used to acquire a predicate lock as before. To avoid redundant computation in the data platform, DIBS may internally rewrite the request, substituting the scalar result for the expression in the predicate.

DIBS could also be extended to support predicate locking for some join predicates. For example, consider the following query.

```
SELECT *
FROM T, U
WHERE T.a = 1 AND T.a = U.a;
```

Conceptually, this query could be restructured into two queries that access relations `T` and `U` separately with the predicate `a = 1`, followed by a cartesian product between the two resulting sets of tuples. This eliminates the join predicate, allowing DIBS to acquire two predicate locks as before. Importantly, the restructured query returns the same rows as the original query. Hence, the restructuring can be an internal DIBS operation and require no modification to the data platform's query plan.

For other operations such as natural joins without filter predicates, DIBS can always fall back on coarse granularity locks. In the worst case, an entire relation may be locked for the duration of a transaction. However, this may be the case for any isolation mechanism and, as we have shown, DIBS supports fine-granularity predicate locking for a broad range of predicate structures.

4 EVALUATION

In this section, we present an evaluation of DIBS in a variety of applications. Our evaluation is structured into two sections. First, we evaluate DIBS as the sole isolation mechanism for a prototype data processing engine built on Apache Arrow [16]. We cumulatively apply each predicate locking optimization and measure performance relative to naive predicate locking. Second, we demonstrate how fully-optimized DIBS can improve performance of OLTP workloads as an augmentation to existing database systems.

4.1 Workloads and Applications

We characterize the performance of our system on four benchmarks: TATP [34], SEATS [44], *SubscriberScan* (a benchmark that we developed specifically for this paper), and YCSB [6].

The TATP benchmark is designed to measure the performance of a transaction manager in a typical telecommunications application. We selected TATP because it has been extensively used in prior work in this area [21, 22, 25, 29, 35–37] and thoroughly compared to other transaction benchmarks [12]. In all TATP experiments, we use a database containing 1 million subscriber records and the recommended key distribution.

The SEATS benchmark models an airline ticketing system where customers search for flights and make reservations. The ticketing system is designed to allow customers to access the database using various credentials, such as frequent flyer number, customer account number, or login name. Thus, many of the transactions in SEATS use foreign-key joins and secondary indexes to identify a customer. The benchmark consists of a relatively write-intensive mix of 60 % read only transactions and 40 % transactions that update, insert, or delete records in the database. Compared to TATP, SEATS includes more complex predicates involving range searches and set membership. We used the benchmark driver provided by the developers of the OLTP-Bench framework [12].

SubscriberScan is a benchmark that we developed specifically to test the ability of DIBS to handle complex predicates. As noted earlier, TPC-C, TATP, and YCSB do not include predicates with disjunctions, but rather only predicates that are conjunctions of comparisons [6, 7, 34]. Because these predicates can be converted to DNF in linear time, these benchmarks are insufficient to fully evaluate our predicate locking optimizations. In contrast, SubscriberScan includes complex predicates with disjunctions and inequalities. SubscriberScan involves two transaction types, both of which involve a scan on the `Subscriber` table from TATP with 100 K rows. The transactions are executed in an 80%/20% mix. The first transaction type reads the data from each row that satisfies the predicate. The

second transaction type updates the `vlr_location` field to a random value for each row that satisfies the predicate. The predicate for both transaction types is of the following form.

```
WHERE ((byte2_1 BETWEEN ? AND ?)
       OR (byte2_1 BETWEEN ? AND ?))
  AND ((byte2_2 BETWEEN ? AND ?)
       OR (byte2_2 BETWEEN ? AND ?))
  AND ...;
```

Each `BETWEEN` interval is 16-wide and randomly chosen from the interval [0, 255]. In our experiments, we vary the number of conjuncts in the predicate from 1 to 8.

YCSB is commonly used for evaluating key-value systems, but has been adapted to transactional databases as well [28, 48]. We selected YCSB because it exposes configurable parameters that allow measurement of specific sources of overhead. In our YCSB evaluation, we varied the number of requests per transaction, the proportion of read requests to write requests, and the skew factor of the requested keys. Borrowing from existing work [28], we refer to the 95% select/5% update mix as *read-intensive*, and the 50% select/50% update mix as *write-intensive*. All YCSB experiments were run against a database containing 1 million records, each with an integer primary key and 10 fixed-width character fields of 100 bytes, for a total of about 1 GB of user data. Each YCSB request either queries one of these fields for a desired primary key, or updates the field with a new character string.

For TATP and YCSB, we choose $N = 1024$ buckets of predicate locks per relation. For SEATS, we use $N = 8$. For SubscriberScan, we use $N = 1$. These choices maximize throughput for each workload based on empirical tests. We omit the exponential blowup threshold k to demonstrate the overhead of converting the predicates in SubscriberScan to DNF.

We execute these benchmarks for three applications, which will be discussed in detail in the following sections. We provide configuration details for each application here. The first application uses a lock-free in-memory database engine developed with the Rust library of Apache Arrow version 3.0.0. As a baseline for the Arrow engine, we compare with SQL Server optimized for in-memory OLTP, often referred to as Hekaton [11]. All tables were set to memory-optimized and durability was set to `SCHEMA_ONLY`. Whenever supported, natively compiled stored procedures were used. Where appropriate, columnstore indexes were used. The second application uses SQLite version 3.33.0. We set SQLite to multithreaded mode and disabled the collection of memory allocation statistics. We also set the cache size to 8 GB, the journal mode to WAL, and the synchronization of the journal to FULL. All other SQLite settings were left as default. The third application uses MySQL version 8.0 with InnoDB backend. We set the buffer size to 8 GB and vary the isolation level depending on the experiment. All other MySQL and InnoDB settings were left as default.

4.2 Testing Setup

All experiments were run on a single machine with two Intel Xeon Silver 4114 processors with a clock speed of 2.2 GHz. Each processor has 10 cores each for a total of 20 cores. Each core has its own 32 KB L1 cache and 1024 KB L2 cache, with a 13.75 MB L3 cache shared between cores on a single processor. The machine has 192 GB of DDR4 SDRAM with 96 GB on each socket and one 480 GB Intel DC S3500 SATA SSD. We pinned DIBS worker threads to cores, utilizing the second socket only when greater than 10 workers were needed. The machine was provisioned using the CloudLab platform [13].

In all experiments, DIBS was run on the same machine as the client application and the target database engine. Workload generators were implemented as DIBS client connectors and included in the main event loop of each worker thread. In embedded settings, DIBS worker threads also perform the work of the requests using the API of the target database engines. In client-server settings, workers simply send requests to the database server for processing. In most experiments, we vary the number of worker threads from 1 to 20. In experiments that involve SQLite, we use up to 10 worker threads as system behavior generally remains steady beyond this point.

For each experimental configuration, we run five independent trials and plot the mean value, with the standard error of the mean shown as error bars. Each trial was run with a warmup duration of 10 seconds followed by a measurement duration of 60 seconds. We quantify throughput in transactions per second (tps). We quantify file system writes in SQLite by counting the number of calls to `fdatasync`, which SQLite uses to flush the binary log to stable storage, divided by the total number of transactions committed during the measurement duration.

4.3 Isolation in a Lock-Free Database

To demonstrate how DIBS provides serializable isolation as a service for new systems, we developed a lightweight, in-memory data processing engine that operates on Apache Arrow data [16]. Arrow is a cross-language data format specification that has enjoyed a recent surge in popularity due to its usefulness as an interface between query processing systems. In our implementation, tabular data is stored as a collection of Arrow arrays. We use Rust standard library data structures, such as `HashMap` and `BTreeMap` for indexing and `Mutex` for synchronization. The engine does not provide concurrency control beyond latching to protect data structures. Instead, DIBS ensures that any request admitted to the data processing engine does not violate a serializable schedule. This effectively decouples transaction isolation control flow from request execution, removing isolation overhead from the consideration of how best to execute a given request. We cumulatively apply each of our predicate locking optimizations and measure throughput relative to both naive predicate locking and SQL Server optimized for in-memory OLTP.

Column filtering provides the most significant performance improvement for TATP as shown in Figure 3a. The throughput resulting from the first two optimizations, conjunct grouping and prepared predicates, is nearly indiscernible from the naive implementation for this workload. This is due to the simplicity of TATP's predicates: each is a small conjunction of comparisons, or even a single comparison. Because the predicates do not include disjunctions, conjunct grouping has no effect. Additionally, because the predicates are small, ad hoc conflict evaluation is comparable to the evaluation of prepared predicates. However, due to contention for

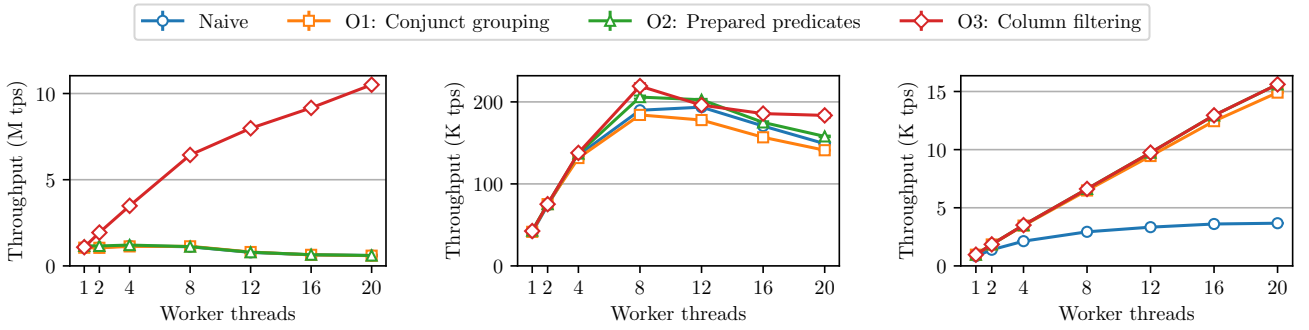


Figure 3: Throughput on OLTP benchmarks in the Arrow engine for each cumulative predicate locking optimization.

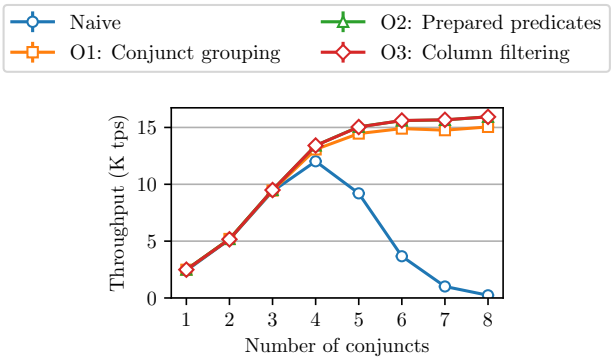


Figure 4: Throughput on SubscriberScan in the Arrow engine with 20 worker threads, varying number of conjuncts per predicate.

the centralized set of active predicate locks, the naive implementation does not scale. By removing this bottleneck, column filtering provides a substantial improvement in scalability. At 20 worker threads, fully-optimized DIBS executes 10.5 M tps, an approximate 10X improvement over the maximum throughput of the naive implementation. Memory-optimized SQL Server achieves a maximum of 58 K tps on this workload. We observed similar behavior for YCSB, but omit these results for brevity.

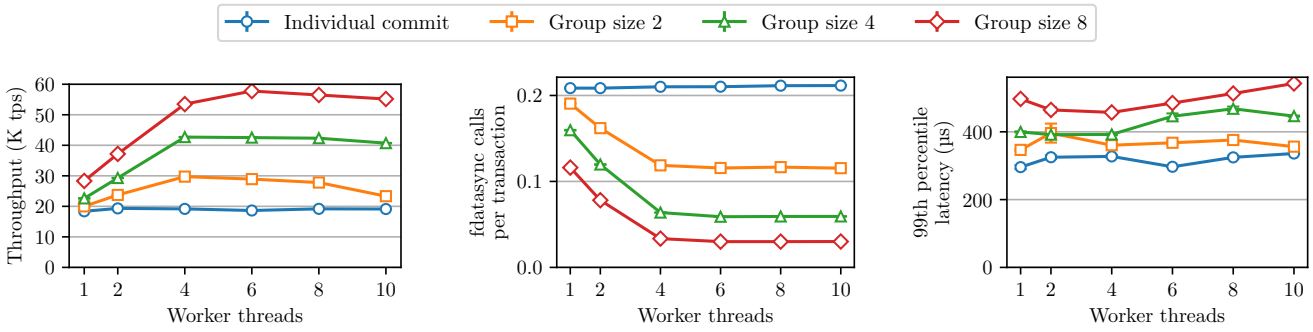
Both column filtering and prepared predicates provide modest improvements over naive on SEATS as shown in Figure 3b. Conjunct grouping incurs a slight performance penalty from the separation of predicates into groups. However, this penalty is countered by the benefit of the additional optimizations. At maximum, fully-optimized DIBS executes 220 K tps, an approximate 14% improvement over the maximum throughput of the naive implementation. The SEATS benchmark driver involves significant client-side processing, such as caching the results of a transaction for use in a later transaction and accessing latch-protected data structures. CPU profiling reveals that the majority of execution time is spent on client-side code, with fully-optimized DIBS incurring less than 5% overhead. Memory-optimized SQL Server achieves a maximum of 900 tps on this workload.

Conjunct grouping provides the most significant performance improvement for SubscriberScan. In Figure 3c, we fix the number of conjuncts per predicate at 6 and vary the number of worker threads. The naive implementation scales modestly, but tapers off quickly, achieving a maximum throughput of approximately 3700 tps. The optimized implementation scales much more rapidly, achieving a maximum throughput of approximately 15 K tps, a 4X improvement over the naive implementation. Figure 4 offers an explanation of this behavior. In this experiment, we fix the number of worker threads at 20 and vary the number of conjuncts per predicate. The naive implementation offers comparable performance to the optimized implementation up to about 4 conjuncts per predicate. After this point, the exponential complexity of converting to DNF begins to dominate the running time of the worker threads, which results in a substantial decrease in throughput. Note that as we increase the number of conjuncts, the predicate becomes more selective and hence less likely to conflict with other predicates, explaining the upward trend in throughput. At 6 conjuncts per predicate, memory-optimized SQL Server achieves a maximum throughput of 3 K tps.

4.4 Transaction Merging

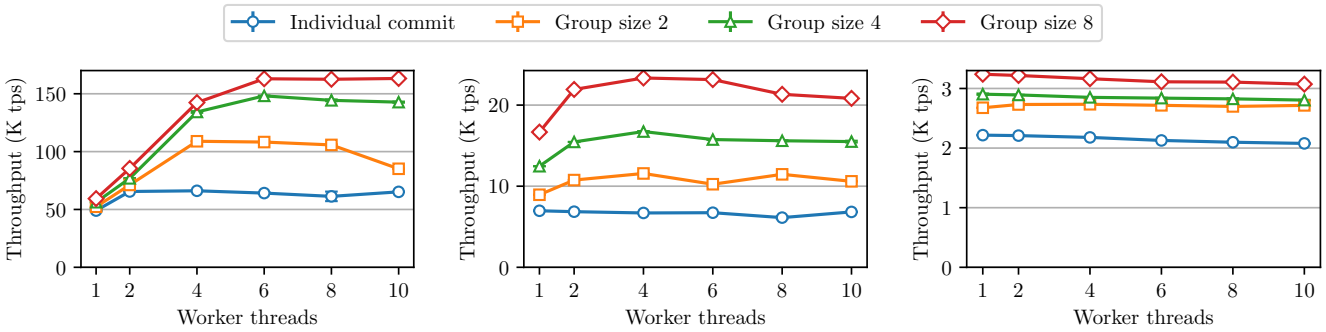
SQLite is the most widely deployed database engine to date [42]. As an embedded system, SQLite allows unlimited concurrent read transactions across multiple processes, but serializes write transactions through file-level locking. A new write transaction may not begin until the changes caused by the previous write transaction are flushed to persistent storage. This design becomes a significant bottleneck and limits scaleup even in read-intensive workloads.

Here, we apply DIBS to increase transaction throughput on SQLite by reducing the number of file system calls per transaction. The modularity of DIBS allows us to achieve this with little effort: we make a small modification to the DIBS worker loop, delaying the commit() command on the database until the requests from several client transactions have been issued to the SQLite database engine or a timeout expires. SQLite has no knowledge of which requests belong to which client transactions and hence



(a) Throughput **(b) File writes per client transaction** **(c) 99th percentile latency**

Figure 5: Transaction merging with DIBS increases throughput on TATP workloads by reducing the number of file system calls per client transaction in SQLite.



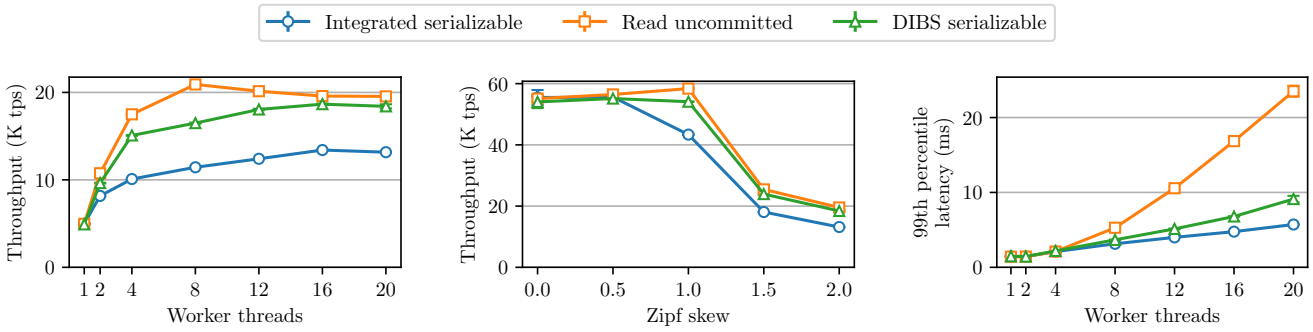
(a) Read intensive, 1 request per client transaction **(b) Write intensive, 1 request per client transaction** **(c) Write intensive, 8 requests per client transaction**

Figure 6: Transaction merging increases single-threaded performance and multi-threaded scalability on YCSB in SQLite.

provides no isolation between them. DIBS is then necessary to enforce a serial ordering. Clients are notified of a successful commit only when the outer transaction commits. Savepoints are placed at the end of each inner transaction to minimize the undoing of useful work in the event of an abort. As a result, SQLite can often flush the changes of multiple client transactions with a single file system call. This application of DIBS provides the same isolation at considerably higher performance without having to rewrite the core SQLite transaction processing components.

This technique is analogous to group commit, in which changes to the database are flushed from memory to persistent storage as a batch, rather than at the end of each transaction. However, the key difference between group commit and transaction merging with DIBS is that group commit is implemented in the storage manager of a database engine, whereas our technique can be applied as a modular layer over an existing database engine without modifying the source code or restarting the server. In addition, transaction merging with DIBS does not require a centralized buffer manager to coordinate the flushing of changes.

Figure 5 shows results from the evaluation of transaction merging with DIBS for the TATP benchmark on SQLite. SQLite's integrated individual commit mechanism does not scale well on TATP's 80% read, 20% write workload mix as the cost of writing to the file system dominates transaction processing. However, with the addition of transaction merging with DIBS, the system scales up to 57 K tps at 6 threads, an approximate 3X improvement over SQLite's individual commit. The driving factor behind this improvement is a reduction in file system writes per client transaction, as shown in Figure 5b. This reduction increases the throughput of write transactions, allowing the system to benefit from the addition of more threads to execute read-only transactions. Regardless of the number of threads, SQLite's individual commit mechanism averages just above 0.2 calls to `fdatasync` per client transaction. This behavior is expected, as TATP consists of 20% write transactions, each typically requiring one `fdatasync` call. In contrast, DIBS allows multiple write transactions to be committed in a group, often with just one `fdatasync` call per group. As the system scales up to 6 threads, the number of file system writes per client transaction is further reduced in all group sizes except individual commit.



(a) Scaleup for skew = 2.0 **(b) 20 worker threads, varying skew** **(c) 99th percentile latency, skew = 2.0**

Figure 7: DIBS scales better than MySQL integrated serializable isolation on YCSB when accessing different columns of contended rows.

Transaction latency is an important consideration in this setting because each transaction may not complete until the final transaction in its group is committed. We show the effect of group size on transaction latency in Figure 5c. As expected, the transaction latency increases as the number of transactions per group increases. However, at maximum throughput, 99% of transactions complete in less than 0.5 ms. As mentioned earlier, for latency-sensitive applications, DIBS may trigger the commit of a group when a timeout expires, rather than wait for additional transactions to arrive.

Figure 6 shows the results from an evaluation of transaction merging with DIBS on a suite of YCSB configurations. We include these results to highlight the workload characteristics for which transaction merging with DIBS provides substantial performance gains. For read-intensive YCSB, we observe only modest improvements from transaction merging on a single thread because most transactions do not write to the file system. However, throughput improves dramatically with the addition of more worker threads, reaching up to 160 K tps with 1 request per client transaction at group size 8, a 2.5X speedup compared to individual commit as shown in Figure 6a. For write-intensive YCSB with 1 request per client transaction shown in Figure 6b, transaction merging with DIBS achieves higher performance gains on a single thread. With group size 8 on a single thread, throughput reaches about 17 K tps, a 2.4X speedup from individual commit. At maximum, transaction merging with DIBS achieves 23 K tps, a 3X speedup from individual commit. For write-intensive YCSB with 8 requests per client transaction shown in Figure 6c, increasing the group size provides a modest performance improvement across all worker thread configurations. Additional experiments show that further increasing group size to 16 or more yields small performance improvements, but we omit these results for brevity.

These results indicate that transaction merging with DIBS improves throughput under a wide variety of workload characteristics, but has the greatest multithreaded scaleup for read-intensive workloads and the most single-threaded improvement for workloads with many small write transactions.

4.5 Fine-Granularity Locking

For some transaction workloads, DIBS provides higher concurrency than a traditional locking isolation mechanism with a resource hierarchy that terminates at the row level. This behavior occurs when transactions reference different columns within the same row. In traditional locking mechanisms, an entire row is generally locked, and hence cannot be shared. While it is known that this isn't always necessary [49], it is a common simplification. In contrast, DIBS locks at the finest granularity possible given the declarative information in transaction requests. Hence, a row can be shared freely among write transactions, provided that accesses to intersecting columns are read-only. This benefit is most apparent in workloads consisting of transactions that tend to access different columns of highly contended rows. Assumptions about database system behavior that allow this are further discussed in Section 5.

We characterize the performance gains of fine-granularity locking with DIBS on MySQL, which uses a row-level locking isolation mechanism. While MySQL provides no default isolation level that would allow transactions to concurrently write to the same row, its read uncommitted isolation level allows an unlimited number of readers *and* one writer per row. For high-contention, read-intensive transaction workloads, read uncommitted isolation can provide substantial performance gains compared to serializable isolation. We developed a system consisting of DIBS, which provides serializable isolation by default, on top of MySQL set to read uncommitted isolation level. We compare this system to MySQL without DIBS set to both serializable and read uncommitted isolation levels. We evaluate these configurations on a read-intensive YCSB workload consisting of 1 request per transaction. Each request uniformly accesses one of 10 fields in a row that is chosen at random from a Zipf distribution. We vary the skew of the row distribution to modulate contention.

Figure 7 shows the results of this evaluation. In Figure 7a, we fix the Zipf skew to 2.0 and vary the number of worker threads. As we scale beyond a single thread, MySQL's read uncommitted isolation provides substantially better throughput compared to MySQL's serializable isolation. Serializable isolation with DIBS scales between read uncommitted and integrated serializable. At 20 worker threads,

DIBS provides serializable isolation with 1.4X higher throughput compared to MySQL's integrated mechanism and achieves approximately 70% of the performance gain of switching from serializable to read uncommitted isolation. In Figure 7b, we fix the number of worker threads at 20 and vary the Zipf skew between 0.0 and 2.0. Beyond Zipf skew of 0.5, the throughput of MySQL's integrated serializable isolation begins to degrade. Although MySQL's read uncommitted isolation mechanism shows a similar degradation, it retains higher performance at all values of Zipf skew beyond 0.5. Serializable isolation with DIBS falls between MySQL's read uncommitted and serializable isolation levels and is able to maintain high throughput despite the row contention. Lastly, in Figure 7c, we measure the 99th percentile transaction latency across thread configurations for Zipf skew 2.0. As the system scales, transactions run with serializable isolation provided by DIBS have slightly higher latency than those run with MySQL's integrated mechanism. Interestingly, DIBS serializable transactions have considerably lower latency compared to MySQL read uncommitted transactions. With DIBS as a layer above MySQL, conflicting transactions are detected by optimized predicate locking in DIBS, lessening the burden on the concurrency control mechanism in MySQL. These experiments show that DIBS provides high-throughput isolation when implemented as a layer above an existing transactional database system and scales better than MySQL's integrated serializable isolation mechanism for high-contention workloads.

5 DISCUSSION

5.1 Database System Assumptions

To move the isolation mechanism outside of the database system, we must make some specific assumptions about how the database system handles reads and writes. These assumptions allow us to provide isolation based on predicates rather than physical tuples. In existing database systems, we generally find that these assumptions are covered by existing mechanisms such as latches and concurrent data structures. However, for new database systems that rely solely on DIBS as an external isolation provider, such as ours described in Section 4.3, it is useful to describe the minimal responsibilities that such a system must retain to guarantee correctness.

First, physical writes and reads must enforce ACID properties to individual values within a tuple. Our system only provides isolation across transactions, and hence it assumes that the database system itself will read and save individual values correctly. Second, the database system must synchronize data structures that are used to fulfill a request but not explicitly referenced in the request. For example, if there is a global transaction counter that every transaction increments without being referenced in a request, the DIBS mechanism would be unaware of it and hence not lock it appropriately. For such cases, additional synchronization may be necessary to ensure correctness. For example, the global transaction counter may be updated using an atomic operation. Third, transactions should not access values within a tuple outside of the columns the request indicates are necessary. In some systems, this may be an issue if they update entire tuples in place: read entire tuples, make changes locally, and then write over the entire tuple [17]. This could cause two transactions to overwrite each other's data by re-writing an old value in a column they did not actually update.

We make no further assumptions about abort, rollback, or other database functionality as long as our base assumptions are met within that functionality as well. For example, transaction rollback during an abort would be expected to roll back individual fields rather than a snapshot of the entire tuple. We note that DIBS does not attempt to provide guarantees of atomicity, consistency, or durability. Providing these guarantees as modular services is a related set of active research topics. There is promising work in this direction, particularly focused on transaction recovery methods [4], that could be leveraged to develop companion services to DIBS that manage other aspects of transaction processing. An exploration of this design space is part of future work.

5.2 Supporting additional isolation levels

While DIBS is designed to provide fully serializable transaction isolation by default, it is straightforward to add support for additional isolation levels such as repeatable read, read committed, and read uncommitted. As described in Section 2.1, two predicate locks are said to conflict if they access the same relation, at least one is a write, and the conjunction of their predicates is satisfiable. To support each additional isolation level, we allow certain types of conflicting locks that would otherwise be prohibited under the serializable isolation level or make small modifications to how locks are acquired and released.

To support the repeatable read isolation level, we allow inserts and deletes that conflict with predicate locks over ranges. All other conflicts, including those between inserts/deletes and equality predicate locks, are still prohibited as before. This modification effectively allows phantom reads to occur. To support the read committed isolation level, we acquire predicate locks for `SELECT` statements as before, but rather than hold the predicate locks for the duration of the transaction, we release them as soon as the `SELECT` statement has completed. This modification effectively allows non-repeatable reads to occur. All predicate lock conflicts are handled in the same manner as repeatable read. Lastly, to support the read uncommitted isolation level, we do not acquire predicate locks for `SELECT` statements at all. This modification effectively allows dirty reads to occur, in which uncommitted modifications to the database are visible to external transactions. All predicate lock conflicts are handled in the same manner as repeatable read.

5.3 Transaction Isolation-as-a-Service

The DIBS system demonstrated in this paper can be implemented directly in a data platform or as a standalone service. We argue that the same advantages we see in microservice architectures [1] apply to splitting transaction isolation into an external service. As noted in Section 1, DIBS reduces the complexity of transactional storage management and shortens the time required to add transactional support to a new system. However, some additional benefits may result from providing transaction isolation as a service. We describe these benefits here.

Making the isolation mechanism its own service allows it to be scaled independent of the database. This approach has a distinct advantage since the amount of work done to provide isolation guarantees can vary by task. Conventional database systems typically follow a monolithic design for concurrency control, logging, and

query processing [20]. With these database systems increasingly deployed onto cloud architectures and accessed as a service [2, 33], dynamic scaling of these databases is a problem [19]. DIBS can be scaled as necessary independent of the database. Most importantly, it can be scaled quickly with very little state to store. Minimizing the stored state is important because scaling dynamically is more difficult in a distributed or mostly main memory database since both transferring data between nodes and loading data back into memory from persistent storage is slow.

Isolation-as-a-service also has the advantage of making one set of isolation logic accessible to many different data platforms. This approach also allows the possibility of using multiple isolation mechanisms within the same database. Prior work has studied the problem of dynamically selecting the best performing isolation mechanism for a given workload [45]. Transaction isolation that is provided as an external service allows the possibility of swapping isolation functionality in and out as needed.

6 RELATED WORK

A number of recent publications have explored separating isolation mechanisms from transaction execution in database systems. Lomet et al. [30], building on previous work by Sears and Brewer [40], propose a design in which transaction component (TC) is disaggregated from the data component (DC). This design is evolved in [31] and later implemented in the Deuteronomy system [26, 27]. While Deuteronomy emphasizes separation of concerns between the TC and the DC, there are two key contrasts with our work. First, Deuteronomy uses timestamp ordering multi-version concurrency control as an isolation mechanism. The TC acts as a cache that associates timestamps with data items from the DC. This mechanism requires that data be transferred between the DC and the TC. In contrast, DIBS uses predicate analysis to provide isolation and hence requires no data movement to and from the target data platform, allowing it to be implemented as a completely independent service. Second, the TC in Deuteronomy is designed only for key-value data stores. DIBS places no restriction on the data model of the target data platform.

Das et al. [8] propose a "Key Group" abstraction and an associated transactional protocol, which enable multi-key transactional operations in a key-value store. The authors also present G-Store, an implementation of the protocol as a layer above a key-value store, and demonstrate its scalability on multi-key access workloads. Similar to the transaction component in Deuteronomy, G-Store is restricted to key-value stores, whereas DIBS has no such restriction.

Calvin [47] is another example of independent transaction isolation because it provides a transaction scheduling layer decoupled from its execution layer to provide isolation between transactions. Calvin transactions are defined for specific data store operations on logical resources. It prevents conflicts by maintaining a lock table at the scheduling layer and only admits operations to execute when there are no conflicting operations running. However, Calvin's scheduling algorithm still depends on the contents and operations of the data store. This means that within a declarative database management system, it would require knowledge of the logical query execution plan, which in most systems is optimized using details from the data store.

Similar to Calvin, Bohm [15] achieves state-of-the-art multi-version concurrency control by performing transaction ordering and data versioning prior to processing. This allows the actual processing to proceed without isolation control. This makes execution considerably simpler and more efficient. However, threads in Bohm require knowledge of the contents of the database to determine whether they should apply concurrency control logic to a transaction. In addition, the read and write sets of transactions must be available in advance. DIBS has no such restriction.

There has also been work focused on specific aspects of transaction functionality in the database kernel. For example, Arulraj et al. [4] evaluate how storage and transaction recovery methods will fare in a changing hardware landscape that includes non-volatile memory devices. Our work in modular transaction isolation is complementary to this research because modularity enables individual components to independently evolve to meet changing application demands and hardware capabilities. With DIBS, we take a first step in a broader design refactoring for all transactional components.

A great deal of literature is available on concurrent scheduling that could be leveraged for designing a stronger scheduling mechanism. However, most existing work focuses on scheduling operations within a single transaction efficiently, or optimizing the performance across operations in the database [10, 18]. Some work has focused on applying scheduling principles within the database to reduce conflicts and optimize performance. However, these approaches enhance the performance of existing isolation mechanisms, whereas DIBS provides complete isolation guarantees without external concurrency control.

Finally, there has been recent work on optimizing entire transaction flows from an application perspective [50, 51]. This work is closely aligned with our observations that how requests are admitted to the database system is very important. A key practical difference between previous work and ours is that they focused on optimizing transactions from the application side, whereas our work focuses on an application-agnostic database solution.

7 CONCLUSION

Industry trends indicate a growing demand for microservice-based data processing components [30]. This is at odds with popular isolation mechanisms such as locking and multi-versioning, which are generally tightly coupled with the database storage manager and intertwined with data processing code. In this paper, we developed and evaluated DIBS, a novel transaction scheduling system that uses optimized predicate locking to provide isolation as a service. We demonstrated that DIBS can provide high-throughput serializable isolation on a transaction-agnostic data processing system. We also identified key applications where DIBS improves throughput on established database systems. We hope that this work prompts new developments in modular transaction services.

ACKNOWLEDGMENTS

This work was supported in part by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program, sponsored by MARCO and DARPA. Additional support was provided by the National Science Foundation (NSF) under grant OAC-1835446.

REFERENCES

[1] Gojko Adzic and Robert Chatley. 2017. Serverless Computing: Economic and Architectural Impact. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 884–889. https://doi.org/10.1145/3106237.3117767

[2] Divyakant Agrawal, Sudipto Das, and Amr El Abbadi. 2011. Big Data and Cloud Computing: Current State and Future Opportunities. In *Proceedings of the 14th International Conference on Extending Database Technology* (Uppsala, Sweden) *(EDBT/ICDT '11)*. Association for Computing Machinery, New York, NY, USA, 530–533. https://doi.org/10.1145/1951365.1951432

[3] Chris Anley. 2002. *Advanced SQL Injection In SQL Server Applications*. Technical Report. Next Generation Security Software Ltd.

[4] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. 2015. Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 707–722. https://doi.org/10.1145/2723372.2749441

[5] Marc Brooker. 2015. *AWS Architecture Blog: Exponential Backoff And Jitter*. Amazon Web Services. https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/

[6] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[7] The Transaction Processing Council. 2020. TPC-C Benchmark (Version 5.11.0). http://www.tpc.org/tpcc/

[8] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2010. G-Store: A Scalable Data Store for Transactional Multi Key Access in the Cloud. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 163–174. https://doi.org/10.1145/1807128.1807157

[9] DB-Engines. 2020. *DB-Engines Ranking*. https://db-engines.com/en/ranking

[10] Harshad Deshmukh, Hakan Memisoglu, and Jignesh M. Patel. 2017. Adaptive Concurrent Query Execution Framework for an Analytical In-Memory Database System. In *2017 IEEE International Congress on Big Data (BigData Congress)*. 23–30.

[11] Cristian Diaconu, Craig Freedman, Erik Ismert, Paul Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *ACM International Conference on Management of Data 2013 (SIGMOD '13)*. https://www.microsoft.com/en-us/research/publication/hekaton-sql-servers-memory-optimized-oltp-engine/

[12] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proceedings of the VLDB Endowment* 7, 4, 277–288. https://doi.org/10.14778/2732240.2732246

[13] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

[14] Kapali P. Eswaran, Jim N. Gray, Raymond A. Lorie, and Irving L. Traiger. 1976. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM* 19, 11 (Nov. 1976), 624–633. https://doi.org/10.1145/360363.360369

[15] Jose M. Faleiro and Daniel J. Abadi. 2015. Rethinking Serializable Multiversion Concurrency Control. *Proceedings of the VLDB Endowment* 8, 11 (July 2015), 1190–1201. https://doi.org/10.14778/2809974.2809981

[16] Apache Software Foundation. 2019. Apache Arrow: A cross-language development platform for in-memory data. https://arrow.apache.org

[17] Jim Gray. 1981. The Transaction Concept: Virtues and Limitations. *Proceedings of Seventh International Conference on Very Large Databases*, 144–154.

[18] Chetan Gupta, Abhay Mehta, Song Wang, and Umesh Dayal. 2009. Fair, Effective, Efficient and Differentiated Scheduling in an Enterprise Data Warehouse. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology* (Saint Petersburg, Russia) *(EDBT '09)*. Association for Computing Machinery, New York, NY, USA, 696–707. https://doi.org/10.1145/1516360.1516441

[19] Hakan Hacıgümüş, Bala Iyer, and Sharad Mehrotra. 2002. Providing Database as a Service. In *Proceedings of the 18th International Conference on Data Engineering*. 29–38. https://doi.org/10.1109/ICDE.2002.994695

[20] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. 2007. Architecture of a Database System. *Foundations and Trends in Databases* 1, 2 (2007), 141–259. https://doi.org/10.1561/1900000002

[21] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. 2014. NVRAM-Aware Logging in Transaction Systems. *Proceedings of the VLDB Endowment* 8, 4 (Dec. 2014), 389–400. https://doi.org/10.14778/2735496.2735502

[22] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. 2012. Scalability of Write-Ahead Logging on Multicore and Multisocket Hardware. *The VLDB Journal* 21, 2 (April 2012), 239–263. https://doi.org/10.1007/s00778-011-0260-8

[23] Kyle Kingsbury. 2020. *Jepsen: PostgreSQL 12.3*. https://jepsen.io/analyses/postgresql-12.3

[24] Kyle Kingsbury and Peter Alvaro. 2021. Elle: Inferring Isolation Anomalies from Experimental Observations. *PVLDB* 14, 3 (2021), 268–280. https://doi.org/10.14778/3430915.3430918

[25] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *Proceedings of the VLDB Endowment* 5, 4 (Dec. 2011), 298–309. https://doi.org/10.14778/2095686.2095689

[26] Justin Levandoski, David Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. 2015. High Performance Transactions in Deuteronomy. In *7th Biennial Conference on Innovative Data Systems Research (CIDR '15)*. https://www.microsoft.com/en-us/research/publication/high-performance-transactions-in-deuteronomy/

[27] Justin Levandoski, David Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. 2016. Multi-Version Range Concurrency Control in Deuteronomy. In *42nd International Conference on Very Large Data Bases (VLDB '16)*. https://www.microsoft.com/en-us/research/publication/multi-version-range-concurrency-control-deuteronomy/

[28] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 21–35. https://doi.org/10.1145/3035918.3064015

[29] Jan Lindström, Vilho Raatikka, Jarmo Ruuth, Petri Soini, and Katriina Vakkila. 2013. IBM solidDB: In-Memory Database Optimized for Extreme Speed and Availability. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 14–20.

[30] David B. Lomet, Alan Fekete, Gerhard Weikum, and Michael J. Zwilling. 2009. Unbundling Transaction Services in the Cloud. In *4th Biennial Conference on Innovative Data Systems Research (CIDR '09)*. www.cidrdb.org. https://www.microsoft.com/en-us/research/publication/unbundling-transaction-services-in-the-cloud/

[31] David B. Lomet and Mohamed F. Mokbel. 2009. Locking Key Ranges with Unbundled Transaction Services. In *Proceedings of the VLDB Endowment (VLDB '09)*. https://doi.org/10.14778/1687627.1687658

[32] MongoDB. 2012. *MongoDB 2.2 Released - MongoDB Blog*. https://www.mongodb.com/blog/post/mongodb-22-released

[33] Barzan Mozafari, Carlo Curino, and Samuel Madden. 2013. DBSeer: Resource and Performance Prediction for Building a Next Generation Database Cloud. In *6th Biennial Conference on Innovative Data Systems Research (CIDR '13)*. http://cidrdb.org/cidr2013/Papers/CIDR13_Paper52.pdf

[34] Simo Neuvonen, Antoni Wolski, Markku Manner, and Vilho Raatikka. 2020. Telecom Application Transaction Processing Benchmark. http://tatpbenchmark.sourceforge.net

[35] Ismail Oukid, Daniel Booss, Wolfgang Lehner, Peter Bumbulis, and Thomas Willhalm. 2014. SOFORT: a hybrid SCM-DRAM storage engine for fast data recovery. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware (DaMoN '14)*. https://doi.org/10.1145/2619228.2619236

[36] Ismail Oukid, Wolfgang Lehner, Thomas Kissinger, Thomas Willhalm, and Peter Bumbulis. 2015. Instant Recovery for Main-Memory Databases. In *7th Biennial Conference on Innovative Data Systems Research (CIDR '15)*. http://cidrdb.org/cidr2015/Papers/CIDR15_Paper13.pdf

[37] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi Bhuyan. 2011. No More Backstabbing... A Faithful Scheduling Policy for Multithreaded Programs. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. 12–21. https://doi.org/10.1109/PACT.2011.8

[38] Kun Ren, Alexander Thomson, and Daniel J. Abadi. 2012. Lightweight Locking for Main Memory Database Systems. *Proceedings of the VLDB Endowment* 6, 2 (Dec. 2012), 145–156. https://doi.org/10.14778/2535568.2448947

[39] RocksDB. 2020. *Transactions*. https://github.com/facebook/rocksdb/wiki/Transactions

[40] Russell Sears and Eric Brewer. 2006. Stasis: Flexible Transactional Storage. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington) *(OSDI '06)*. USENIX Association, USA, 29–44.

[41] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data* (Boston, Massachusetts) *(SIGMOD '79)*. Association for Computing Machinery, New York, NY, USA, 23–34. https://doi.org/10.1145/582095.582099

[42] SQLite. 2020. *Most Widely Deployed and Used Database Engine*. https://www.sqlite.org/mostdeployed.html

[43] Michael Stonebraker and Ugur Cetintemel. 2005. "One Size Fits All: An Idea Whose Time Has Come and Gone. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*. IEEE Computer Society, USA, 2–11. https://doi.org/10.1109/ICDE.2005.1

[44] Michael Stonebraker and Andrew Pavlo. 2012. *The SEATS Airline Ticketing Systems Benchmark*. http://hstore.cs.brown.edu/projects/seats

[45] Dixin Tang, Hao Jiang, and Aaron J. Elmore. 2017. Adaptive Concurrency Control: Despite the Looking Glass, One Concurrency Control Does Not Fit All. In *8th Biennial Conference on Innovative Data Systems Research (CIDR '17)*. http://cidrdb.org/cidr2017/papers/p63-tang-cidr17.pdf

[46] Robert Endre Tarjan. 1975. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM* 22, 2 (April 1975), 215–225. https://doi.org/10.1145/321879.321884

[47] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2213836.2213838

[48] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 18–32. https://doi.org/10.1145/2517349.2522713

[49] W. E. Weihl. 1988. Commutativity-Based Concurrency Control for Abstract Data Types. *IEEE Trans. Comput.* 37, 12 (Dec. 1988), 1488–1505. https://doi.org/10.1109/12.9728

[50] Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. 2018. How Not to Structure Your Database-Backed Web Applications: A Study of Performance Bugs in the Wild. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 800–810. https://doi.org/10.1145/3180155.3180194

[51] Ningnan Zhou, Xuan Zhou, Xiao Zhang, Xiaoyong Du, and Shan Wang. 2017. Reordering Transaction Execution to Boost High Frequency Trading Applications. In *Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint Conference on Web and Big Data*. 169–184. https://doi.org/10.1007/978-3-319-63564-4_14